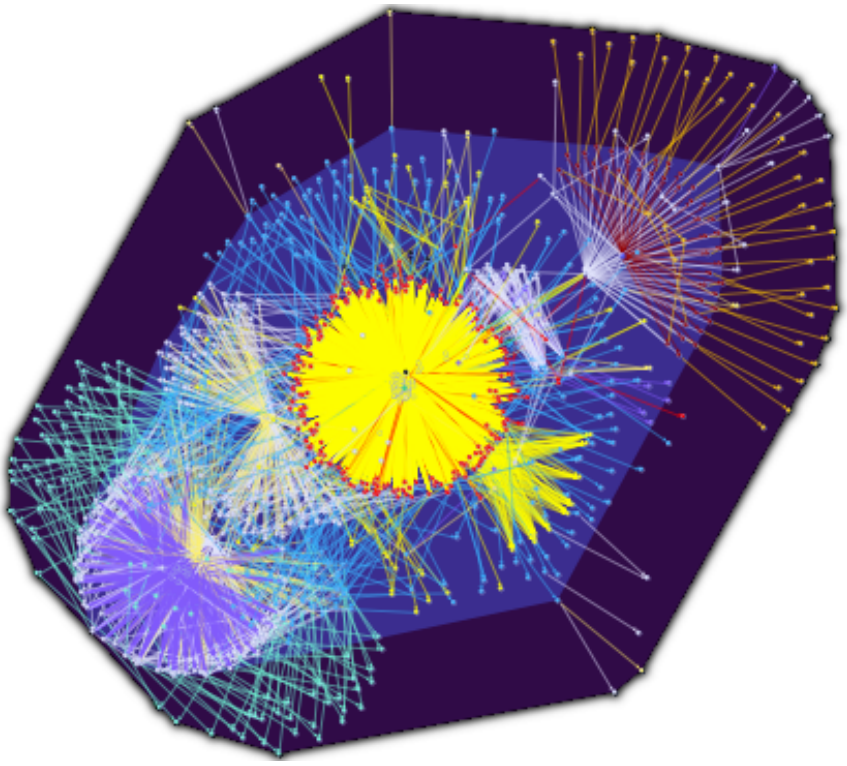
Universiteit Gent
Faculteit Ingenieurswetenschappen
Vakgroep Informatietechnologie

Co-evolutie van broncode en het bouwsysteem: impact op de introductie van AOSD in legacy systemen

Co-Evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems

Bram Adams



Proefschrift tot het bekomen van de graad van
Doctor in de Ingenieurswetenschappen:
Computerwetenschappen
Academiejaar 2007-2008

Universiteit Gent
Faculteit Ingenieurswetenschappen
Vakgroep Informatietechnologie

Promotoren: Prof. Dr. Ir. Herman Tromp
Prof. Dr. Wolfgang De Meuter

Universiteit Gent
Faculteit Ingenieurswetenschappen

Vakgroep Informatietechnologie
Sint-Pietersnieuwstraat 41, B-9000 Gent, België

Tel.: +32-9-264.33.18
Fax.: +32-9-264.35.93

Dit werk kwam tot stand in het kader van een BOF-beurs (Bijzonder Onderzoeks-Fonds) van de Universiteit Gent.

Proefschrift tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen:
Computerwetenschappen
Academiejaar 2007-2008

Preface

"Mathematics and programming!" was my typical answer in elementary and secondary school to other people's queries about my future. I liked mathematics and assumed that programming computers would also be cool. Eventually, I lived up to my bold prediction by pursuing a Master in Computer Science Engineering at Ghent University. At the age of eighteen, I wrote my first "Hello World" in Bart Dhoedt's course on Java. The transition from programming to software engineering followed a couple of years later, when Ghislain Hoffman encouraged me to migrate to the Linux operating system and to the Eclipse IDE. As these two systems spearheaded the open source movement, I became intrigued by design patterns, version control systems, test-driven development, continuous integration, build systems, refactoring, etc. This fascination led me to do my Master's thesis in Ghislain's group, with Herman Tromp, Koenraad Vandenborre and Kris De Schutter as my advisors. The thesis introduced me to AOSD and seduced me into doing a PhD. Ghislain and Herman agreed to become my advisors, and we managed to obtain a BOF grant from Ghent University. I am very grateful to them for this opportunity.

In October 2004, my journey into the world of research officially began, with as initial goal investigating AOSD technology for C. For this task, I could count on the other members of the group. Kris De Schutter was my mentor, who convinced me to think before programming, to use the right tool for the job and to switch to Apple computers. He was right about all of these. David Matthys taught me that there is more to life than just work, but I have been a bad student in this regard. I would also like to thank Hannes Verlinde, José Sampaio Faria, Andy Verkeyn, Koenraad Vandenborre, Jan Van Besien, Mieke Creve and Stijn Van Wonterghem for being themselves.

One given day in February, 2005, my research career got an unexpected boost when all of a sudden Ghislain offered me the chance to go to the AOSD 2005 conference in Chicago. In less than a week, I managed to write my first workshop paper (for SPLAT), together with Tom Tourwé. This paper was the first in a series in which Kris and I tried to solve challenges and issues concerning aspects for C identified by Tom and his colleagues at CWI in Amsterdam (The Netherlands). The SPLAT paper also represented the first real occasion to present my work in public, for which I am thankful to Tom and Kris Gybels, one of the organisers of

SPLAT who has been very influential on my work. Travelling together with him and Andy Kellens (both from the PROG lab at the VUB in Brussels) taught me that research could be fun, that Starbucks has delicious mocha and that Apple stores are cool. Above all, I noticed for the first time that being at a conference refuels ones ideas and desire for research.

Shortly after AOSD, Kris De Schutter and Andy Zaidman arranged a large case study with Kava, in the context of the Arriba project. Andy and I eventually spent several weeks at Kava. During that period, I came to know Andy as a serious researcher with clear ideas and opinions. It was (and still is) fun to work together with him. I also would like to thank Bernard De Ruyck, who graciously allowed us to apply our tools on his Kava system. Looking back, the impact of the Kava case on this dissertation has been significant. Andy and I encountered inexplicably many Makefile problems while introducing AOSD technology, which from the outside looked trivial but could only be solved using hacks. Yet, no tools could be found to help us, nor were there existing experience reports. As the reader will notice, this problem forms exactly the core topic of this dissertation.

By the summer of 2005, our group flourished as never before in terms of PhD students, projects and thesis students. But then, disaster stroke: Ghislain tragically passed away after having broken his leg on the parking lot. This came as a real shock to us, everyone was baffled. Although I have only worked for Ghislain for one year, I will not forget how he would make time to chat with anyone and how he always tried to reconcile both parties in case of conflicts. Without Ghislain, a dark and uncertain period gloomed. The group lost industrial contacts and courses, and some people had to leave. This kind of situation causes a lot of tension and makes people show their real faces. On the upside however, experiencing the slow death of a research group makes one appreciate the good people, who keep on believing in you and who remain supportive of your work.

For this, I first would like to thank Kris De Schutter and David Matthys, with whom it has always been a pleasure to work. In difficult circumstances, we managed to keep up the quality of education, research and thesis guidance. In the context of the latter, we have also encountered many enthusiastic thesis students, who brightened up our office at difficult times. Second, I want to thank Herman Tromp for taking over the role of advisor from Ghislain. He has done everything he could to let us do our work in the best possible circumstances. At the same time, he never let an opportunity slip to improve my cultural skills, especially regarding culture originating from Antwerp, but often to no avail. His weekly mocking about my favourite soccer team (SV Zulte-Waregem) was quite appreciated.

Third, I would like to thank the Belgian software engineering research community in general, and the PROG lab of Theo D'Hondt and Wolfgang ("Wolf") De Meuter in particular. PROG is a unique research group, in which thorough research is stimulated, but without sacrificing the pleasant things in life (drink anyone?). I have always been welcome at PROG (and SSEL), for which I am grateful. Theo encouraged me to co-operate with PROG members and, more generally, to think outside the box. At a crucial point in my PhD career (AOSD 2007), he introduced me to Wolf and this is without any doubt the best thing which happened to me

in the last three and a half years. Wolf is a very inspiring person who has a gift for explaining complex concepts in a simple, irresistible way. As my co-advisor, he taught me valuable things about writing papers and doing research. Thanks to him, this dissertation has a clear, explicit architecture with research questions, named criteria, meaningful titles, etc. I do not think there are many advisors who buy themselves a scanner solely to email their manually written feedback in the middle of the night. Thanks, Wolf!

Of the past three and a half years, 2007 has been the most exciting period, as I could reap the fruits of the MAKAO and Aspicere2 tools I had designed and developed in 2006. Apart from the work with Kris De Schutter, Andy and Tom, this has given me the opportunity to work together with a number of other people. I have had the fortune to be invited by Yvonne Coady and Michael Haupt to work with them on the VMADL project, together with Celina Gibbs and Stijn Timbermont. Similar to attending a conference, collaborating with internationally acclaimed researchers is highly inspiring and motivating, especially if each one works in a different domain. The VMADL collaboration has sparked other joint work, more in particular with Charlotte Herzeel and Kris Gybels, and with Yvonne, Celina, Chris Matthews and Bart Van Rompaey. I would like to thank all of these people for the enriching research experience, and I sincerely hope to continue these efforts.

When the time of writing this dissertation came (near the end of 2007), Yvonne Coady, Arie van Deursen, Koen De Bosschere, Bart Dhoedt and Filip De Turck accepted to be in my jury, chaired by Paul Kiekens. I would like to thank them for their constructive feedback, and for the courage to read this dissertation (the reader will probably guess why). During the writing process, I was fortunate to receive valuable feedback from Wolf, Kris De Schutter, Stijn Timbermont and Herman. This was not straightforward, given my voluminous writing style and the lack of structure in the initial versions. Various other people have made the conception and writing of this dissertation a bit easier. Slinger Jansen and Arie van Deursen pointed me to crucial related work, whereas Kai Germaschewski and Sam Ravnborg provided important feedback on the Linux kernel build system case study. Bruno De Fraine gave me a significantly time-saving latex-tip (the `\includeonly` macro). The secretaries An and Monic relieved me from various administrative tasks, and were always in for a chat. The morning commute was cheered up by the vibrant discussions with Piet, Dries, Nicolas, Lode and Tessa. At conferences, workshops, etc. I have met many people with whom I exchanged interesting comments and witty conversations. Thanks for this! Finally, I would like to thank the Smashing Pumpkins for giving me courage that one night before the final writing phase, and SV Zulte-Waregem for the weekly suspense.

Last but not least, I want to express my deepest gratitude to the people who have had to endure me every single day of my life, and who have been able to put into perspective my adventures with computers and work in general. I am referring to my father, mother, sister, family and two dogs. I owe everything to them. They have molded my character, have offered me the chance to study whatever I wanted (as long as I did the best I could), and have always supported me throughout my

studies and research. This was not straightforward, especially after the painful loss of my father almost two years ago, which has made a deep impact on our lives. I am really grateful for the opportunities my father and mother have given me, and I wish I could return all of their favours.

Ghent, March 2008
Bram Adams

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

ADL	architectural description language
AO	aspect orientation
AOP	aspect oriented programming
AOSD	aspect oriented software development
AST	abstract syntax tree

C

CBSE	component-based software engineering
CCC	crosscutting concern

D

DAG	directed acyclic graph

G

GBS	GNU Build System
GCC	GNU Compiler Collection
GNU	GNU is Not Unix

I

IR	intermediate representation

J

JIT	just-in-time compilation

M

MAKAO	Makefile Architecture Kernel featuring AOP

O

OO	object orientation

P

PCD	pointcut (declaration)

Nederlandse samenvatting –Summary in Dutch–

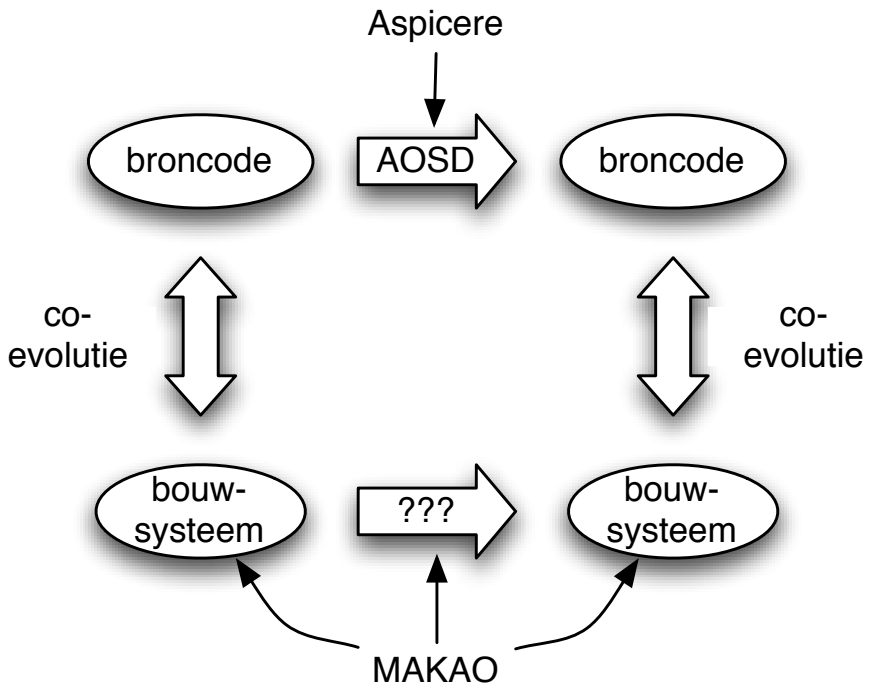
DIT doctoraatswerk onderzoekt het fenomeen van co-evolutie van broncode en het bouwsysteem, zowel conceptueel als experimenteel, en toont, opnieuw zowel conceptueel als experimenteel, hoe dit fenomeen een belangrijke impact heeft op de introductie van aspectgeoriënteerde software-ontwikkeling (AOSD)[1] technieken in zogenaamde legacy omgevingen. De komende secties vatten dit werk samen. Eerst stellen we de onderzoeksvragen voor die we onderzocht hebben. Daarna geven we een overzicht van onze aanpak voor het oplossen van elk van die vragen.

Probleemstelling

Dit doctoraatswerk stelt dat de introductie van AOP technologie in een legacy systeem gehinderd wordt door het fenomeen van co-evolutie van broncode en het bouwsysteem (Figuur 1). Legacy systemen zijn oude, bedrijfskritische software systemen die voortdurend omgevormd moeten worden om aan de schier oneindige stroom van nieuwe vereisten te voldoen [21, 58]. Het bouwsysteem is de infrastructuur die instaat voor het coördineren van compilers[2], preprocessors[3] en andere hulpmiddelen om uitvoerbare code te genereren uit broncode ("bouwlaag"), en om software-ontwikkelaars in staat te stellen om modules en functionaliteit te configureren die ingebouwd moeten worden in het uiteindelijk bouwresultaat ("configuratielaag"). Wanneer de broncode evolueert, moet het bouwsysteem mee evolueren om het effect van broncodewijzigingen te kunnen verifiëren en te corrigeren. Langs de andere kant zullen software-ontwikkelaars geneigd zijn om hun broncode minder optimaal te structureren indien het bouwsysteem hen niet toelaat om flexibel veranderingen aan te brengen. Initiële indicaties en experimentele aanwijzingen hiervoor werden reeds eerder in de literatuur vermeld [57, 58]. Dus, er bestaat een belangrijke spanning, meer bepaald co-evolutie [243, 232, 85, 128, 172], tussen broncode en het bouwsysteem.

De gevolgen van co-evolutie van broncode en het bouwsysteem komen vooral tot uiting in de context van de introductie van AOP technologie in legacy sys-

[1] Een alternatieve afkorting is "AOP", d.w.z. "aspectgeoriënteerd programmeren".
[2] Programma dat broncodebestanden vertaalt naar machinecode.
[3] Programma dat broncode in een andere vorm giet.



Figuur 1: Schematisch overzicht van co-evolutie van broncode en het bouwsysteem.

temen. AOSD [134] is voorgedragen als een geschikte technologie om legacy systemen om te vormen [51, 103, 142, 202, 170]. Een "aspect" encapsuleert de implementatie van een "overlappend facet" uit de "basiscode"[4] als een afzonderlijke module. Het kan meerdere "adviezen" bevatten die, in tegenstelling tot gewone functies, niet expliciet opgeroepen worden door de basiscode, maar daarentegen automatisch uitgevoerd worden wanneer aan een specifieke voorwaarde ("puntsnede") voldaan is op een bepaald moment tijdens uitvoering van het basisprogramma ("samenvloeiingspunt"). Evaluatie van de puntsnede ("het passen van samenvloeiingspunten") en het oproepen van het juiste advies is de verantwoordelijkheid van de "wever". In praktijk gebeurt het weven normaal tijdens het bouwproces, zodat zoveel mogelijk beslissingen voor het passen van samenvloeiingspunten statisch uitgevoerd worden. Statische wevers transformeren daarvoor broncode, bytecode of machinecode.

AOSD brengt een aanzienlijke verandering van programmeerparadigma teweeg in de broncode, d.w.z. dat één uiteinde van de co-evolutie relatie een drastische wijziging ondergaat. Omwille van co-evolutie van broncode en het bouwsysteem zijn compensatie-acties nodig in het bouwsysteem om beide uiteinden van

[4] Alternatieve naam: "basisprogramma".

de co-evolutie consistent te houden met elkaar (zie de horizontale pijlen op Figuur 1). Helaas hebben legacy systemen ook een legacy bouwsysteem, dat slecht gedocumenteerd is en moeilijk te begrijpen. Deze onzekerheid omtrent het gedrag en de structuur van het bouwsysteem verhindert veranderingen in het bouwsysteem en kan leiden tot compromissen voor de integratie van AOP technologie met het bouwsysteem. Deze compromissen tasten de semantiek van de aspecten in de broncode aan. Dit betekent dat onvoldoende middelen om met co-evolutie van broncode en het bouwsysteem om te gaan de introductie van AOP technologie in legacy systemen tegen houdt.

Dit doctoraatswerk geeft een antwoord op de volgende zes onderzoeksvragen:

1. Wat zijn de fundamentele verklaringen voor co-evolutie van broncode en het bouwsysteem?

2. Welke soort hulpmiddelen hebben we nodig om de co-evolutie fenomenen van broncode en het bouwsysteem te begrijpen en te controleren in legacy systemen?

3. Kunnen we onze hulpmiddelen gebruiken om de fundamentele verklaringen experimenteel te staven door ze toe te passen op externe legacy systemen?

4. Hoe valt de introductie van AOSD te rijmen met de fundamentele verklaringen voor co-evolutie van broncode en het bouwsysteem?

5. Hoe kunnen we AOSD technologie ontwerpen die op gepaste wijze omgaat met de co-evolutie indien ze toegepast wordt op legacy systemen?

6. Kunnen we deze technologie valideren door ze te gebruiken om co-evolutie fenomenen in bestaande legacy systemen te controleren?

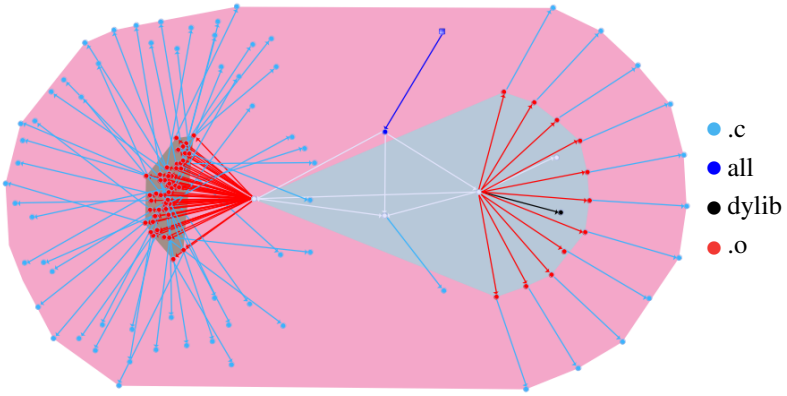
De volgende secties vatten samen hoe dit doctoraatswerk elk van deze vragen beantwoord heeft.

Wat is Co-evolutie van Broncode en het Bouwsysteem?

Co-evolutie [85] van broncode en het bouwsysteem komt overeen met “asynchrone evolutie van broncode en het bouwsysteem tijdens dewelke veranderingen aan één artifact een verticale impact heeft op een andere en omgekeerd”. We hebben een conceptuele verklaring van deze co-evolutie afgeleid onder de vorm van vier “bronnen van co-evolutie”:

RC1 Modulair redeneren in programmeertalen en compilatie-eenheden in bouwsystemen zijn sterk met elkaar verbonden.

RC2 Bouwafhankelijkheden worden toegepast om de architectuur van de broncode te reïfiëren.



Figuur 2: Voorbeeld van een bouwafhankelijkheidsgraaf.

RC3 Architecturale grensvlakken in de broncode worden niet altijd gerespecteerd door incrementele bouwprocessen.

RC4 De configuratielaag van een bouwsysteem wordt gebruikt als laag-technologische ondersteuning, geënt op productlijnen, voor variabiliteit in broncode.

De vier bronnen van co-evolutie vormen de basis voor het begrijpen van de co-evolutie van broncode en het bouwsysteem, en voor hulpmiddel- en aspecttaalondersteuning om ermee om te gaan.

Hulpmiddelen om Co-evolutie te begrijpen en te controleren

Uit de vier bronnen van co-evolutie hebben we vijf vereisten afgeleid voor hulpmiddelondersteuning om co-evolutie van broncode en het bouwsysteem te begrijpen en te controleren: visualisatie, vraagstelling, filteren, verificatie en omvorming. Deze vereisten volgen uit het doel T1 om klassieke bouwproblemen in legacy systemen op te lossen en het doel T2 om te helpen bij het begrijpen en controleren van de co-evolutie. MAKAO[5] (Makefile Architecture Kernel featuring Aspect Orientation) is een terugwinnings- en omvormingsraamwerk voor bouwsystemen, die aan deze vereisten voldoet. Het bouwsysteemmodel waarop MAKAO gebaseerd is, is de afhankelijkheidsgraaf van een concrete bouwuitvoering, aangevuld met de waarden van bouwvariabelen en statische informatie uit de bouwspecificaties. Figuur 2 toont een voorbeeld van zo'n bouwafhankelijkheidsgraaf. Knopen komen overeen met bouwdoelen, terwijl de takken van de graaf de afhankelijkheden tussen doelen weergeven. Kleuren bevatten informatie over het soort doel, zoals aangeduid in de legende.

[5] `http://users.ugent.be/~badams/makao/`

We hebben MAKAO's mogelijkheden om symptomen van co-evolutie van broncode en het bouwsysteem te identificeren gevalideerd in zes projecten, waarvan er vijf te maken hebben met AOP. Visualisatie en vraagstelling van bouwsystemen zijn nuttig gebleken in elk project. Ten tweede is het filteren van het bouwsysteem enkel van belang gebleken voor grote bouwsystemen met complexe bouwlogica. Het omvormen van het bouwsysteem is enkel toegepast voor ingrijpende, grote bouwspecificatiewijzigingen. Andere veranderingen konden manueel aangepakt worden, gebaseerd op de informatie die gehaald werd uit visualisatie en vraagstelling. We verwachten dat verificatie cruciaal is voor het valideren van het resultaat van MAKAO's omvormingsfunctionaliteit.

In het algemeen hebben we aangetoond dat de voorgestelde hulpmiddelondersteuning capabel is om het begrip en de controle van co-evolutie van broncode en het bouwsysteem gevoelig te verbeteren. Daarentegen zou MAKAO aangevuld moeten worden met een hulpmiddel om de configuratielaag te analyseren.

Experimenteel Bewijs van de Bronnen van Co-evolutie in Legacy Systemen

We hebben experimenteel bewijs verzameld van de geldigheid van de vier bronnen van co-evolutie doorheen de evolutie van het bouwsysteem van de Linux besturingssysteemkern. De mensen die dit bouwsysteem onderhouden, worstelen doorlopend met het vinden van manieren om de integratie van nieuwe broncodecomponenten te verbeteren (RC1) en om de bouwsysteemafhankelijkheden consistent te houden met de broncode-architectuur (RC2). We hebben expliciet bewijs gevonden van de niet aflatende afweging tussen het weglaten van broncodeafhankelijkheden tijdens het bouwen om hogere bouwsnelheden te bereiken, en het vrijwaren van de bouwcorrectheid (RC3). De evolutie van de configuratielaag wordt gedomineerd door de extreme eis om de geëxplodeerde configureerbaarheid van de broncode te controleren (RC4). Deze bevindingen valideren de vier bronnen van co-evolutie.

Wat is het Verband tussen de Introductie van AOSD en Co-evolutie?

De broncodewijzigingen geïntroduceerd door AOP technologie kunnen gecorreleerd worden met elk van de vier bronnen van co-evolutie. Aspecten voeren integraal redeneren in op het broncodeniveau [219, 162, 135] i.p.v. het traditionele modulair redeneren [187] (RC1). Omkering van afhankelijkheden [182, 160, 162], fijnmazige compositie van aspecten en de intensionele selectie van samenvloeiingspunten door puntsnedes verhinderen synchronisatie van broncode- en bouwsysteemafhankelijkheden (RC2). Incrementeel weven compromitteert de consistente compositie van aspecten omwille van integraal redeneren [135], fijnmazige

```
1 /*basiscode*/
2 PMC* pt_transfer_sub(Parrot_Interp d, Parrot_Interp s,
3                       PMC *sub){
4   return make_local_copy(d, s, sub);
5 }
6
7 /*advies*/
8 void debug_transfer(Parrot_Interp S, PMC* Sub) before Jp:
9  execution(Jp,``pt_transfer_sub'')
10  && args(Jp,[_,S,Sub])
11  && thread_debug(_){
12   PIO_eprintf(S, "copying over subroutine [%Ss]\n",
13               Parrot_full_sub_name(S, Sub));
14 }
```

Figuur 3: Voorbeeld van een Aspicere advies en de basiscode waarin het geweven wordt.

compositie en aspectinteractie [137, 113, 114, 74] (RC3). De verbeterde modulariteit van de broncode [219, 162], de invloed van de volgorde van weven [163, 137, 113, 114, 74] en het bestaan van afhankelijkheden tussen aspecten [75] vereisen betere communicatie tussen de broncode en de configuratielaag (RC4). Langs de andere kant kunnen aspecten ook de basiscode ontkoppelen van configuratielogica. De correlatie tussen AOSD en de vier bronnen van co-evolutie suggereren dat wanneer AOP technologie geïntroduceerd wordt in een legacy systeem, het bouwsysteem aangepast moet worden om consistentie tussen broncode en het bouwsysteem te bewaren.

AOSD Technologie om om te gaan met Co-evolutie

De hulpmiddelondersteuning die geïntroduceerd werd om co-evolutie van broncode en het bouwsysteem te begrijpen en te controleren, is zowel geldig indien AOP technologie gebruikt wordt, als wanneer dit niet zo is. Maar, indien AOP toegepast wordt, kunnen we van de bronnen van co-evolutie vereisten extraheren voor aspecttaalondersteuning om co-evolutie van broncode en het bouwsysteem te controleren (doel L2). RC4 suggereert om de bouwsysteemstructuur en -configuratie te integreren met de aspecttaal. Dit stelt ons in staat om toegang te krijgen tot configuratiebeslissingen en bouwafhankelijkheden vanuit puntsnedes en adviezen, zodat aspecten robuust gemaakt kunnen worden t.o.v. de bouwconfiguratie en we aspecten kunnen laten redeneren over de hoogniveau broncode-architectuur. Deze vereisten complementeren vereisten (doel L1) voor natuurlijke integratie van de aspecttaal in de basistaal, voor het specifiëren van robuuste puntsnedes, voor het ontwikkelen van generiek advies en voor het aanbieden van uitgebreide context aan adviezen.

Ondersteuning voor de L1 en L2 doelen is ingebouwd in Aspicere[6][7], d.i. onze aspecttaal voor C die gebaseerd is op de principes van logisch metaprogrammeren (LMP) [237, 30]. Figuur 3 toont een voorbeeldadvies (lijnen 8–14) die de uitvoering (lijn 8) van de `pt_transfer_sub` procedure (lijnen 2–5) adviseert. Aspicere's puntsnedetaal is gebaseerd op Prolog, d.w.z. dat puntsnedes uitgedrukt kunnen worden in termen van Prolog vraagstellingen (`execution` en `args`) en feiten (`thread_debug`). Logische feiten worden gebruikt om de structuur van de basiscode voor te stellen, maar kunnen ook aangewend worden om de structuur en de configuratie van het bouwsysteem voor te stellen (doel L2). Dit vereist enkel uitwisseling van deze informatie tussen het bouwsysteem en de logische feitendatabank. Lijn 11 van Figuur 3 controleert bijvoorbeeld of de `thread_debug` preprocessorvlag gedefinieerd is. Zodoende is het `debug_transfer` advies in staat om de conditionele compilatielogica uit de broncode te vervangen, d.w.z. om in de knoop geraakt en over de broncode verspreid preprocessorgebruik te verhuizen naar afzonderlijke modules (aspecten).

Validatie van AOSD Technologie om om te gaan met Co-evolutie

We hebben vijf projecten uitgevoerd waarin AOP technologie geïntroduceerd werd in een legacy C systeem, met als doel om extra experimenteel bewijs te verzamelen van co-evolutie van broncode en het bouwsysteem in de aanwezigheid van AOSD, en om de capaciteit van Aspicere te valideren om co-evolutie van broncode en bouwsysteem te controleren (doel L2) in legacy systemen (doel L1). Daarnaast hebben we deze projecten ook gebruikt om de evaluatieresultaten van MAKAO behaald tijdens de analyse van het bouwsysteem van de Linux besturingssysteemkern (doel T2) aan te vullen.

De hoofdoorzaak van het co-evolutieprobleem gelieerd aan RC1 en RC2 is het begrijpen en het definiëren van de notie van "integraliteit", d.w.z. het bereik van aspecten. Conceptueel worden aspecten doorheen de volledige basiscode toegepast, maar de grenzen van bibliotheken en uitvoerbare programma's, en de complexe interacties tussen deze bouwcomponenten maken deze notie minder helder. Het bouwsysteem heeft expliciete controle over het bereik van aspecten, en dus over de semantiek van het samengestelde systeem.

De huidige aspectwevers zijn veel trager dan basiscodecompilers, maar het bouwsysteem kan het weefproces niet significant versnellen op een veilige manier, omdat het niets afweet over de fijnmazige samenstelling aangeboden door aspecten. Incrementele weving inbouwen in wevers is echter niet voor de hand liggend, bijvoorbeeld omdat statische analyses om de geweven code te optimaliseren niet kunnen omgaan met incrementele wijzigingen aan de broncode. Er is nog altijd

[6] Van het Latijnse werkwoord "aspicere", dat "kijken naar" betekent. De kern van zijn voltooid deelwoord is "aspect-".

[7] `http://users.ugent.be/~badams/aspicere/`

veel werk te doen om om te kunnen gaan met de co-evolutieproblemen veroorzaakt door RC3.

Het verhoogde potentieel voor configureerbaarheid in een AOSD systeem vergt nauwe controle. Behalve het bepalen van het bereik van aspecten op hoog niveau ("integraliteit" van RC1), omhelst configuratie ook de selectie van een consistente verzameling aspecten om toe te passen op een systeem, en de associatie van aspecten met specifieke verzamelingen van basiscodemodules. Dit is een uitdagende taak, vooral wanneer de configuraties sterk fluctueren, bijvoorbeeld gedurende migratie naar een volledig omgevormd systeem.

Aspicere's ondersteuning voor integratie van informatie over de bouwstructuur en -configuratie in de logische feitenbasis (doel L2) is nuttig geweest in drie projecten. Bouwcomponentinformatie, de huidige bouwconfiguratie en de actieve configuratie-opties werden gebruikt om robuust advies samen te stellen en om de basiscode te ontkoppelen van de configuratielogica. We zijn ervan overtuigd dat de uitwisseling van bouwsysteeminformatie in het algemeen een effectief middel is voor een aspecttaal om om te gaan met problemen veroorzaakt door co-evolutie van broncode en het bouwsysteem. In het algemeen heeft Aspicere aangetoond dat het capabel is om om te gaan met legacy systemen (doel L1).

Conclusie

Dit doctoraatswerk onderzoekt co-evolutie van broncode en het bouwsysteem. We hebben conceptueel en experimenteel bewijs verzameld van het bestaan en de aard van deze co-evolutie, en we hebben aangetoond hoe de co-evolutie problemen veroorzaakt voor legacy systemen om om te gaan met broncodewijzigingen geïnitieerd door AOSD. Om co-evolutie van broncode en het bouwsysteem te begrijpen en te controleren, hebben we vereisten afgeleid voor hulpmiddel- en aspecttaalondersteuning. Deze ondersteuning is gevalideerd in zes projecten, waarvan vijf in de context van AOP in legacy systemen. Deze hebben aangetoond dat hulpmiddelondersteuning in staat is om te assisteren bij het begrijpen en omgaan met co-evolutie van broncode en het bouwsysteem.

English summary

THIS dissertation examines the phenomenon of co-evolution of source code and the build system, both conceptually and experimentally, and shows, again both conceptually and experimentally, how it has an important impact on the introduction of aspect oriented software development (AOSD[8]) techniques in legacy systems. We summarise this work in the coming sections. First, the research questions we have investigated are presented. Then, we give an outline of how we have addressed each question.

Problem Statement

This dissertation conjectures that the introduction of AOP technology in a legacy system is hampered by the phenomenon of co-evolution of source code and the build system (Figure 4). Legacy systems are old, mission-critical software systems which continually have to be re-engineered to deal with a constant stream of new requirements [21, 58]. The build system is the infrastructure which is responsible for co-ordinating compilers, preprocessors and other tools to generate executable code from source code ("build layer"), and to enable developers to configure the modules and features which have to be incorporated in the build product ("configuration layer"). When the source code evolves, the build system needs to be adapted to debug or test the effects of the source code changes. Conversely, if a build system does not allow flexible changes, developers might be tempted to write spaghetti code just to facilitate integration of new source code into the build process. Initial indications and early experimental evidence for this can be found in the literature [57, 58]. Hence, there is an important tension, i.e. co-evolution [243, 232, 85, 128, 172], between source code and the build system.

The consequences of co-evolution of source code and the build system especially become visible in the context of the introduction of AOP technology in legacy systems. AOSD [134] has been proposed as a suitable technology to re-engineer legacy systems [51, 103, 142, 202, 170]. An "aspect" extracts the implementation of a so-called "crosscutting concern" from the "base code" into a separate module. It can contain multiple "advice"s which, contrary to functions, are not explicitly invoked by the base code, but instead are automatically executed when a specific condition ("pointcut") is satisfied at some point during the

[8] An alternative acronym is "AOP", for "aspect oriented programming".

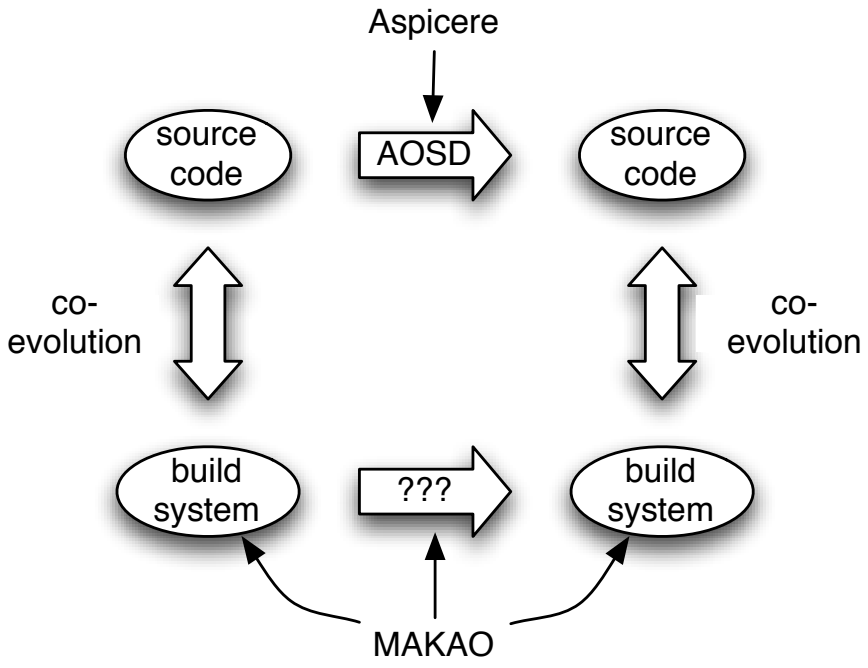


Figure 4: Schematic overview of co-evolution of source code and the build system.

program execution ("join point"). Evaluation of the pointcut ("join point matching") and invocation of the advice is the responsibility of the aspect "weaver". In practice, weaving is usually performed at compile-time, in which case as many join point matching decisions as possible are resolved statically. To do this, static weavers typically transform source code, bytecode or machine code.

AOSD brings with it a major paradigm shift in the source code, i.e. one end of the co-evolution relation undergoes a drastic change. Because of co-evolution of source code and the build system, compensating changes are necessary in the build system to keep both ends of the co-evolution consistent with each other (horizontal arrows on Figure 4). Unfortunately, legacy systems have a legacy build system, which is not well-understood. This uncertainty about the build system's behaviour and structure hinders build changes and may lead to compromises in the integration of AOP technology with the build system. These compromises affect the semantics of the aspects in the source code. Hence, insufficient means to deal with co-evolution of source code and the build system impedes the introduction of AOP technology in legacy systems.

This dissertation addresses the following six research questions:

1. What are the fundamental reasons for co-evolution of source code and the build system?

2. What kind of tools do we need to understand and manage the co-evolution phenomena of source code and the build system in legacy systems?

3. Can we use our tools to confirm the conceptual reasons experimentally by applying them to third-party legacy systems?

4. How does the introduction of AOSD add to the fundamental reasons for co-evolution of source code and the build system?

5. How do we design AOSD technology which adequately deals with the co-evolution when applied to legacy systems?

6. Can we validate this technology by using it to manage co-evolution phenomena in existing legacy systems?

The next sections summarise how this dissertation has addressed each of these questions.

What is Co-evolution of Source Code and the Build System?

Co-evolution [85] of source code and the build system corresponds to "asynchronous evolution of source code and the build system during which changes on one artifact have a vertical impact on the other one and vice versa". We have distilled a conceptual explanation of this co-evolution in the form of four "roots of co-evolution":

RC1 Modular reasoning in programming languages and compilation units in build systems are strongly related.

RC2 Build dependencies are used to reify the architectural structure of the source code.

RC3 Architectural interfaces in the source code are not always respected by incremental build processes.

RC4 Build system configuration layers are used as a poor man's support for product line-styled variability of source code.

The four roots of co-evolution form the basis for understanding the co-evolution of source code and the build system, and for tool and language support to deal with it.

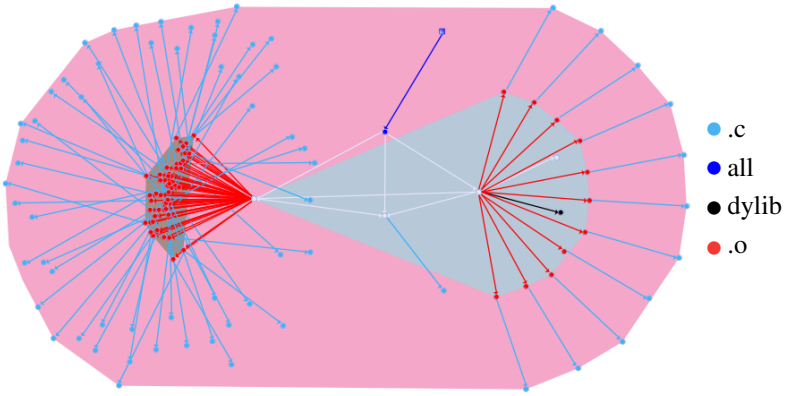


Figure 5: Example build dependency graph.

Tool Support to Understand and Manage Co-evolution

From the four roots of co-evolution, we have extracted five requirements for tool support to understand and manage co-evolution of source code and the build system: visualisation, querying, filtering, verification and re-engineering. These requirements follow from the goal T1 to solve traditional build problems in legacy systems and the goal T2 to assist in understanding and managing the co-evolution. MAKAO[9] (Makefile Architecture Kernel featuring Aspect Orientation) is a reverse- and re-engineering framework for build systems which satisfies these requirements. The build system model on which MAKAO is based is the dependency graph of a concrete build run, enhanced with the build-time values of variables and static information of the build scripts. Figure 5 shows an example build dependency graph. Nodes correspond to build targets, whereas edges display the dependencies between targets. Colors convey information about the type of target, as indicated by the legend.

We have validated MAKAO's ability to identify symptoms of co-evolution of source code and the build system on six case studies, of which five deal with AOP technology. Visualisation and querying of build systems have been useful for every case study. Second, filtering of the build system has only been needed for large build systems with complex build recursion. Re-engineering of the build system has exclusively been used for invasive, large build script changes. Other changes can be resolved manually, based on the information obtained via visualisation and querying. Verification is expected to be crucial for validating the result of MAKAO's re-engineering.

In general, we have shown that the proposed tool support is capable of improving the understanding and management of co-evolution of source code and the build system. However, MAKAO should be complemented by a tool for analysing the configuration layer.

[9] `http://users.ugent.be/~badams/makao/`

Experimental Evidence of the Roots of Co-evolution in Legacy Systems

We have collected experimental evidence of the validity of the four roots of co-evolution by analysing the evolution of the Linux kernel build system. The kernel build maintainers continuously have struggled to improve the integration of new source code components (RC1) and to keep the build system dependencies consistent with the source code architecture (RC2). Explicit evidence has been found about the recurrent trade-off between omitting source code dependencies during the build for higher build speed and maintaining build correctness (RC3). The evolution of the configuration layer is dominated by the extreme demands to manage the abundant configurability of the source code (RC4). These findings validate the four roots of co-evolution.

What is the Relation between the Introduction of AOSD and Co-evolution?

The source code changes introduced by AOP technology can be correlated to each of the four roots of co-evolution. Aspects introduce whole-program reasoning at the source code level [219, 162, 135] instead of traditional modular reasoning [187] (RC1). Inversion of dependencies [182, 160, 162], the fine-grained composition of aspects and the intensional selection of join points via pointcuts hinder synchronisation between source code and build system dependencies (RC2). Incremental weaving challenges the consistent composition of aspects because of whole-program reasoning [135], fine-grained composition and aspect interaction [137, 113, 114, 74] (RC3). The improved source code modularity [219, 162], the influence of weaving order [163, 137, 113, 114, 74] and the existence of dependencies between aspects [75] require better communication between the source code and the configuration layer (RC4). Aspects can on the other hand decouple the base code from configuration logic. The correlation of AOSD with the four roots of co-evolution suggests that when AOP technology is introduced in a legacy system, the build system needs to change to retain consistency between source code and the build system.

AOSD Technology to Deal with Co-evolution

The tool support introduced to understand and manage co-evolution of source code and the build system is valid whether or not AOP technology is used. However, if AOP is applied, we can extract from the four roots of co-evolution requirements for aspect language support to manage co-evolution of source code and the build system (goal L2). RC4 suggests to integrate the build system structure and configuration with the aspect weaver's weave-time meta data. This enables access to

```
1 /*base code*/
2 PMC* pt_transfer_sub(Parrot_Interp d, Parrot_Interp s,
3                       PMC *sub){
4   return make_local_copy(d, s, sub);
5 }
6
7 /*advice*/
8 void debug_transfer(Parrot_Interp S, PMC* Sub) before Jp:
9  execution(Jp,``pt_transfer_sub'')
10  && args(Jp,[_,S,Sub])
11  && thread_debug(_){
12   PIO_eprintf(S, "copying over subroutine [%Ss]\n",
13               Parrot_full_sub_name(S, Sub));
14 }
```

Figure 6: Example Aspicere advice and the base code it is woven into.

configuration decisions and build dependencies from pointcuts and advice to make aspects robust to the build configuration, and even to let them reason about the higher-level source code architecture. These requirements complement requirements of goal L1 for natural integration of the aspect language in the base language, for specification of robust pointcuts, for the development of generic advice and for access to a wealth of context to advice.

Support for the L1 and L2 goals has been incorporated in Aspicere[10][11], i.e. our aspect language for C which is based on the principles of logic meta-programming (LMP) [237, 30]. Figure 6 shows an example advice (lines 8–14) which advises the execution (line 9) of the `pt_transfer_sub` procedure (lines 2–5). Aspicere's pointcut language is based on Prolog, i.e. pointcuts are expressed in terms of Prolog queries (`execution` and `args`) and facts (`thread_debug`). Logic facts are used to represent the structure of the base code, but can also be used to model the build system structure and configuration (goal L2). This only requires exchange of this information between the build system and the weaver. Line 11 of Figure 6 e.g. checks whether the `thread_debug` preprocessor flag has been defined. Hence, the `debug_transfer` advice is able to replace conditional compilation logic in the source code, i.e. to move tangled and scattered preprocessor usage to separate modules (aspects).

[10] From the Latin verb "aspicere", which means "to look at". The root of its past participle is "aspect-".
[11] `http://users.ugent.be/~badams/aspicere/`

Validation of AOSD Technology to Deal with Co-evolution

We have performed five case studies in which AOP technology has been introduced in a legacy C system, with the aim of collecting additional experimental evidence of co-evolution of source code and the build system in the presence of AOSD, and of validating the ability of Aspicere to manage co-evolution of source code and the build system (goal L2) in legacy systems (goal L1). Moreover, these cases have been used to enhance the evaluation results of MAKAO obtained from the Linux kernel build system analysis (goal T2).

The main co-evolution problem attributed to RC1 and RC2 is the understanding and definition of the notion of "whole program", i.e. the scope of aspects. Theoretically, aspects apply across the whole base code, but boundaries of libraries and executables, and complex interactions between these build components blur this notion. The build system has explicit control over the scope of aspects, and hence the semantics of the composed system.

Current aspect weavers are much slower than base code compilers, but the build system cannot significantly accelerate the weaving process in a safe manner, because it knows nothing about the fine-grained composition provided by aspects. Incorporating incremental weaving into weavers is not straightforward however, e.g. because static analyses to optimise the woven code cannot cope with incremental changes to the base code. There is still much work left to be able to deal with the co-evolution problems caused by RC3.

The increased potential for configurability in an AOSD-based system requires tight control. Apart from determining the high-level scope of aspects ("whole program" of RC1), configuration entails selection of a consistent set of aspects to apply on a system, and the association of aspects to specific sets of base code modules. This is a challenging task, especially when the configurations heavily fluctuate, e.g. during migration to a fully re-engineered system.

Aspicere's support for integration of build structure and configuration information with the logic fact base (goal L2) has been useful in three case studies. Build component information, the current build configuration and the currently active configuration options have been used to write robust advice and to decouple the base code from configuration logic. We believe that exchange of build system information in general is an effective means for an aspect language to deal with problems caused by co-evolution of source code and the build system. In general, Aspicere has shown to be capable of dealing with legacy systems (goal L1).

Conclusion

This dissertation investigates co-evolution of source code and the build system. We have distilled conceptual and experimental evidence of the existence and nature of this co-evolution, and have shown how the co-evolution causes problems for legacy systems to deal with source code changes introduced by AOSD. To understand

and manage co-evolution of source code and the build system, we have distilled requirements for tool and aspect language support. This support has been validated on six case studies (of which five deal with AOP in legacy systems), which have shown that tool support is able to assist in understanding of and dealing with co-evolution of source code and the build system.

Software evolution is too often confused with program evolution. Software is much more than programs. [...] Languages, tools and programs evolve in parallel.

Jean-Marie Favre [85]

1 Introduction

1.1 Context of the Dissertation

LEGACY systems are old, mission-critical software which are still plagued by constantly changing requirements [21, 58]. To recover from the initial large investments and because of their crucial role, legacy systems need to serve far more years than originally foreseen. As such, they are typically implemented using a variety of old programming technologies. Over time however, the number of people familiar with the internals of these systems has decreased dramatically. The original design has eroded to enable quick hacks or to implement unforeseen features. Documentation, design information or high-level models of understanding are scarce and often there are quite some unsupported, vendor-specific language dialects involved. Still, the systems must somehow cope with new technologies like EAI (Enterprise Application Integration), SOA (Service Oriented Architecture), etc. If not, the owning company loses competitiveness. Paradoxically, developers tend to avoid changing the implementation for fear of breaking something. On the other hand, completely re-implementing the system using newer technologies is not a good idea either, because of the high cost, because of incomplete requirements and because of missing design information. To summarise, legacy systems have to be kept alive as is, but need to be updated as well to cope with any new requirements.

A number of researchers have claimed that “aspect oriented software develop-

ment"[1] (AOSD) can solve many of the problems of legacy systems [142, 202, 170]. AOSD is a relatively young paradigm [134] aiming to be a solution for modeling so-called "crosscutting concerns". Without AOSD, a concern like persistency is typically spread ("scattered") across various modules which implement the actual business logic of a system. In those modules, the implementation of the persistency concern is heavily mixed ("tangled") with the implementation of the business logic concern. Scattering and tangling are important indicators of potentially troublesome crosscutting between concerns. In our example, the persistency concern crosscuts the business logic concern, but it is not cleanly modularised in the system. Changes to the persistency implementation will have to propagate throughout the whole system because of scattering, whereas tangling negatively impacts understandability of the persistency concern and the other concerns it is intertwined with.

Crosscutting concerns, either functional or non-functional, are not just a recent phenomenon. They are inherent to the problem domain and quality attributes which are modeled. Hence, scattering and tangling are (in general) not the consequence of software design failure. The problems caused by crosscutting concerns primarily result from insufficient software development support for dealing with crosscutting concerns in a modular way. As legacy systems are typically implemented in compositionally less powerful languages and they have to cope with a stream of new requirements which could not have been foreseen when the system was designed, scattering and tangling hamper understanding and re-engineering of legacy systems. As AOSD not only raises awareness about the presence of crosscutting concerns, but also provides means to help in cleanly modularising them, it is a promising technology to help in re-engineering legacy systems [51, 103, 142, 202, 170].

At the programming language level, AOSD proposes to implement a crosscutting concern within a separate module, called an "aspect", which is isolated from the rest of the application, called the "base code"[2]. An aspect groups a number of "advice" constructs. These are similar to procedures, except that they are not explicitly invoked by the base code. Instead, their execution is automatically triggered when a run-time condition of the base code, their "pointcut", is satisfied. This condition can be expressed in terms of program structure, dynamic events, control flow, etc. The moments during program execution on which a pointcut is evaluated and possibly matches, are named "join points". The process of evaluating the pointcut and invoking triggered advice is named "weaving" and is the responsibility of the "weaver". Conceptually, weaving should be perceived as a

[1] A related term is "aspect oriented programming" (AOP), which is used in the context of programming languages and tools.

[2] Asymmetric AOP treats base code and aspects different, i.e. aspects are applied on top of the base code. Symmetric AOP [226, 225] on the other hand treats all modules as equal to each other, but this philosophy is less widespread in practice.

run-time activity, but for efficiency reasons most weaver implementations try to do as much as possible before the program actually runs, e.g. by transforming the source code, post-processing bytecode or weaving at class load-time. The net effect of AOP is that extracting crosscutting concern code into an aspect removes scattering. Furthermore, implicit invocation reduces tangling within the base code.

Although AOSD techniques have originated on the programming language level, in the meantime AOSD has been investigated on various levels: implementation, design (modeling), requirements or analysis ("early" aspects), etc. Most of these efforts have focused on forward-engineering [46]. As already mentioned, some researchers have also identified AOP as being important for re-engineering of legacy systems, i.e. reverse-engineering followed by forward-engineering. Mens et al. [170] argue that, besides economic relevance, the number of legacy systems is significantly larger than the number of newly constructed systems and that they are infected with badly modularised crosscutting concerns. They state that evaluation of AOSD in legacy systems is important for the adoption of AOSD, and that customised language and tool support is needed for this.

In this context, De Schutter proposes [202] AOP language technology to re-engineer legacy (Cobol) systems, i.e. for both reverse- and forward-engineering. Because one can only modify a system if there is enough information about the business rules and sub-components involved, AOP's first task is to help in mining (reverse-engineering) hidden knowledge from the legacy base code. This recovering can be seen as a kind of smart tracing, in which the requested information is gathered in a focused way. Being able to specify the mining logic separately from the base code and to declaratively express when information should be recorded are the two strengths of AOP applied here. Reuse of the recovery logic is a beneficial side-effect. After reverse-engineering, the next step is to restructure or integrate new features in the system, or to re-implement old features using the acquired knowledge. Here, AOP is favourable because crosscutting features can be modeled in separate modules, (ideally) without compromising the existing source code layout. To complement this, pointcuts express the desired composition of the aspects with the base code. To optimise this flux of recovered system knowledge to the forward-engineering phase, De Schutter proposes the application of Logic Meta-Programming (LMP) principles [237, 30] in the aspect language's design. A logic representation of the program, meta data and any other interesting tidbit of information can be used to compose a pointcut as a declarative expression. This allows to express very robust pointcuts based on the program structure or idioms in use instead of on syntactical details. Because the programming languages used in legacy systems are typically rather old and less well-defined, an LMP-based pointcut approach seems like a perfect fit to make AOP work in these environments.

In the following section, we explain how this dissertation continues along this tradition of applying AOP in the context of legacy systems. The focus of our

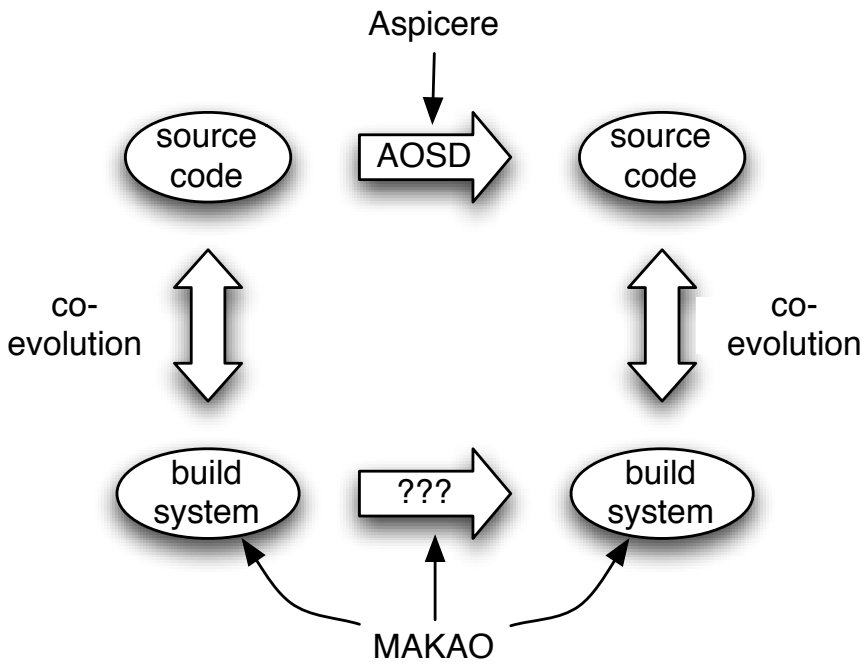


Figure 1.1: High-level overview of co-evolution of source code and the build system.

attention will be the relation between source code and the build system, and how this affects the introduction of AOP technology in legacy systems.

1.2 Problem Statement

As identified by various people, different software models and different levels of describing and thinking about software are causally connected to one another, i.e. they *co-evolve* [243, 232, 85, 128, 172]. If one description changes, this may have an impact on the other one and vice versa. Changes can (and will) occur asynchronously on both sides of the co-evolution relation, i.e. co-evolution is bi-directional. Various co-evolution relations have been studied before: architecture/implementation [243], product line/variability [232], program/language/-tool [85], design regularities/source code [128], source code/tests [172], etc. This is not an exhaustive list, as Favre has pointed out [85]. He conjectures that co-evolution can occur between any two or more levels or artifacts of a particular software development dimension, like e.g. the phases of software development or level-of-abstraction (instance, model, meta-model, etc.). A change on one level sooner or later induces changes on other ones. Introduction of AOSD into a system

to reverse-engineer the design of a system or to re-engineer crosscutting features into aspects is an example of such a change.

This dissertation focuses on one particular instance of co-evolution: co-evolution of source code and the build system (Figure 1.1). The build system is the infrastructure responsible for initiating and directing the construction of the software system from its basic modules. It is also responsible for driving tests in a fluid way or for automating all tedious deployment actions. Hence, it forms a common thread throughout the implementation phase of software development. There are some clear indications that a build system co-evolves with the source code. Intuitively, if developers add new source code files, move existing ones or remove redundant code, chances are high that they will need to modify their makefiles or change the project configuration in their Integrated Development Environment (IDE). A more scientific indication is given by Demeyer et al. [58]. Their "Do a mock installation" reverse-engineering pattern states that the investigation of build and installation process provides valuable insights into the source code architecture. They also note that long build times indicate that a system's internal organisation is too complex, a typical characteristic of legacy systems. A similar relation of the build system and the high-level structure of a software system is mentioned by Favre [84]. To boot, variability in the source code is primarily controlled by the build system and requires invasive, fine-grained communication between the source code and the build system. Given these observations and the fact that both build systems and source code (architecture) may vary, we claim that a build system and the source code co-evolve. This dissertation studies this conjecture and tries to uncover its impact in practice.

It is surprising that despite the importance of co-evolution of build system and source code, evolution and maintenance of the former has been largely ignored in research. Some specific symptoms of this co-evolution have been treated, like modularising the build system to exploit source code reusability [57] or speeding up the build process by restructuring source module dependencies [82], but a systematic account of problems associated with build systems and their evolution has not been made yet. As a consequence, little tool support exists to improve understanding or re-engineering of build systems, let alone deal with co-evolution. This situation becomes even more apparent in legacy systems, where build system concepts and terminology from the 1970s still rule and have been used for tasks they were not designed for [84]. To make up for the inability to scale, various workarounds and hacks have been imposed on top of existing build systems. These have complicated understanding, re-engineering and co-evolution of build systems even more. Tool support is indispensable to recover the design of build systems and to help maintaining them.

Why is the co-evolution of source code and the build system relevant to legacy systems and AOSD? Consider a legacy system into which AOP is introduced at the

source code level [202] (upper horizontal arrow on Figure 1.1). In the wordings of Favre [86], a change of programming paradigm is a change of meta-model. This is an invasive conceptual shift which brings with it other languages and tools [85], i.e. aspect technology has to be integrated into the build system. As mentioned above, build systems in legacy systems do not feature state-of-the-art technology for managing variability or expressing the structure of the system composition [84]. Moreover, only a select number of people understands a system's build [71] because of meagre tool support and a general lack of build documentation [228]. These are exactly the same reasons why one would apply AOP at the source code level in the first place, but their effects within the build system are less known. Just as with source code, one tries to avoid modifications to the build system as much as possible. This leads to a conflict where the introduction of AOP into a system induces drastic changes on the source code level and hence (through co-evolution) on the build system, but where the build system lacks the flexibility to apply these changes. The only way out is a sub-optimal build integration, but this may lead to compromises on the language level, e.g. no flexible (un)plugging mechanism for aspects. In other words, the introduction of AOP clearly requires the need to support co-evolution of source code and the build system, as the limits of traditional build system technology are gradually reached.

Introduction of AOSD is important for a legacy system's survival, but reimplementing or replacing a legacy build system is not straightforward. Hence, it is not sufficient to just point out the danger of co-evolution of source code and the build system on the use of aspects, and to rely on future AOP-aware build systems to dismantle it. AOP should be applied and evaluated within legacy systems now [170]. We present initial tool and aspect language support for managing co-evolution of source code and the build system. MAKAO is a dedicated framework for understanding and maintaining build systems. Aspicere is an aspect language for C with explicit provisions for interfacing with the build system configuration, which allows to extract source code variability controlled by the build system into aspects. Together, they enable developers to understand problems caused by co-evolution and to stimulate successful co-evolution. We evaluate the capability of MAKAO and Aspicere to deal with co-evolution of source code and the build system on a number of representative case studies. These show how co-evolution of source code and the build system represents a realistic challenge for AOSD in legacy systems. Explicit tool and language support is required to face this challenge.

To summarise, these are the six research questions addressed by this dissertation:

1. What are the fundamental reasons for co-evolution of source code and the build system?

2. What kind of tools do we need to understand and manage the co-evolution phenomena of source code and the build system in legacy systems?

3. Can we use our tools to confirm the conceptual reasons experimentally by applying them to third-party legacy systems?

4. How does the introduction of AOSD add to the fundamental reasons for co-evolution of source code and the build system?

5. How do we design AOSD technology which adequately deals with the co-evolution when applied to legacy systems?

6. Can we validate this technology by using it to manage co-evolution phenomena in existing legacy systems?

1.3 Contributions

To answer the research questions of the previous section, this dissertation makes the following contributions:

1. Conceptual evidence about the existence of co-evolution of source code and the build system is collected in the form of four postulated "roots of co-evolution". These explain how changes in the source code have repercussions on the build system and vice versa.

2. We propose a framework, MAKAO, for the reverse- and re-engineering of build systems. It is aimed at solving typical problems in legacy build systems (goal T1), and at enabling developers to investigate and understand practical instances of co-evolution of source code and the build system (goal T2).

3. The four roots of co-evolution and MAKAO are validated by collecting experimental evidence about the existence of co-evolution of source code and the build system in the Linux kernel build system.

4. The introduction of AOP at the source code level can be linked with each of the four roots of co-evolution. As such, the roots suggest that accompanying changes to the build system are required.

5. We propose an aspect language for C, Aspicere, which is capable of dealing with legacy systems (goal L1) and which can interface with the build system (goal L2). The integration of the build system with Aspicere facilitates the decoupling of the base code from tangled configuration logic.

6. The four roots of co-evolution, and the proposed tool and aspect language support are evaluated on two reverse-engineering and three re-engineering case studies in which AOP is introduced in legacy C systems.

Many examples and experiments in this dissertation are situated in C systems, because the C language strikes a good balance between a legacy procedural programming language and a still widely used programming language for systems software (operating systems, virtual machines, compilers, etc.). This makes it easier to find representative legacy systems to experiment with, and also to derive tools to process them. Nevertheless, the conceptual evidence about the existence of co-evolution of source code and the build system, and the goals and requirements for tool and aspect language support, are independent of the programming language used. Hence, we claim that our findings are applicable for legacy systems in general.

The next six sections elaborate on the contributions.

Roots of Co-evolution To be able to postulate the roots of the co-evolution of source code and the build system, we have looked at a comprehensive open source build system, the GNU Build System (GBS), and have investigated existing work on build system technology, smart recompilation strategies, programming language modularity, etc. GBS shows us the many responsibilities of a build system, the numerous ways in how it interacts with the source code and the important problems which hamper its understandability and scalability.

From existing work on build systems and programming languages, we distill a number of crucial links between source code and build system concepts, as depicted by the vertical arrows on Figure 1.1. Each of these links leads us to postulate a root of co-evolution between source code and the build system, i.e. a partial explanation for co-evolution and a potential source of co-evolution problems at the same time. The roots (RC1, RC2, RC3 and RC4) explain why source code changes induce changes in the build system and vice versa. The four roots of co-evolution lay the foundation for understanding co-evolution of source code and the build system, and for tool and language support to deal with it.

Missing Tool Support for Build Systems Despite the importance of the build system, the lack of dedicated tool support to deal with the build system understandability and scalability problems restricts reverse- and re-engineering efforts, and precludes understanding of co-evolution of source code and the build system. We distill five requirements for tool support to solve the aforementioned build system problems (goal T1), and to understand and manage the symptoms of co-evolution explained by the roots of co-evolution (goal T2). We use these five requirements to evaluate existing tools and techniques for supporting the build system. Because none of them satisfies the five requirements, we discuss the design

and implementation of a reverse- and re-engineering framework for build systems, MAKAO, which is explicitly based on the five requirements. MAKAO's ability to meet goals T1 and T2 have been validated on a number of representative case studies, either in the presence or absence of AOSD. We have found that tool support for understanding the build system is crucial to deal with co-evolution of source code and the build system.

Experimental Evidence of Co-evolution without AOSD We have used MAKAO to analyse the evolution of the Linux kernel build system from its inception until recent versions to investigate the practical consequences of co-evolution of source code and the build system in legacy systems. First, physical correlation between source code and the build system is measured via line and file count metrics. Second, the internal complexity of the Linux kernel build system is quantified by MAKAO, and can be linked to concrete build system modifications via freely available kernel developer resources. Third, important evolution steps can be identified from this data, one of which is analysed in detail with MAKAO. The analysis results provide insight into understandability and scalability problems which hamper co-evolution of source code and the build system. These findings enable us to validate the four roots of co-evolution, and MAKAO's ability to satisfy goals T1 and T2.

Co-evolution in the Presence of AOP Introduction of AOSD technology in a legacy system is an example of a big change at the source code level. By relating the introduction of AOP technology in legacy systems (horizontal arrow on top of Figure 1.1) for reverse- or re-engineering to the four roots of co-evolution, we can infer that this introduction also requires the build system to change in order to maintain the co-evolution relation (horizontal arrow at the bottom of Figure 1.1). In other words, co-evolution of source code and the build system in the presence of AOP technology is an important example of the general phenomenon of co-evolution of source code and the build system which was addressed by the previous research questions. Again, the roots of co-evolution contribute to the understanding of co-evolution of source code and the build system, and to the design of tool and language support for dealing with it.

Aspect Language for Legacy C Systems To study co-evolution of source code and the build system when AOP is introduced in a legacy system, we have distilled requirements for an aspect language for legacy systems (goal L1) which is able to improve the co-evolution of source code and the build system by interfacing with the build system (goal L2). To be able to perform case studies in legacy C systems, we have evaluated existing aspect languages for C w.r.t. the requirements specified by goals L1 and L2. Unfortunately, none of the existing AOP languages

for C satisfied all these requirements. We have designed and implemented a new aspect language for C, Aspicere, based on logic meta-programming LMP [237, 30, 202]. We discuss Aspicere's language design, two weaver implementations and two advanced language extensions. The requirements set out by goals L1 and L2 are validated by a comparison between Aspicere and a recently proposed industrial aspect language. Validation of Aspicere's ability to satisfy goals L1 and L2 is presented in five case studies.

Experimental Evidence of Co-evolution with AOSD We have applied Aspicere on two reverse-engineering and three re-engineering case studies to validate the four roots of co-evolution in the context of AOP, and to evaluate the ability of MAKAO (goal T2) and Aspicere (goal L2) to deal with co-evolution of source code and the build system. The reverse-engineering aspects support dynamic program analysis techniques, whereas the re-engineering aspects extract exception handling from the base code, support a declarative architectural description language (ADL) and extract conditionally compiled code into aspects. These case studies show us how the introduction of AOSD in the source code exercises the limits of traditional build system technology. Dedicated tools for understanding build systems and aspect language support for interfacing with the build system are promising techniques for understanding and managing co-evolution of source code and the build system.

1.4 Road Map

This section gives an overview of the focus of each chapter within this dissertation. The dependencies between the chapters are illustrated in Figure 1.2. This scheme also shows how the chapters work together to formulate or validate the roots of co-evolution (RC1 →RC4) and the goals for tool (T1 and T2) and aspect language (L1 and L2) support.

Chapter 2 explains the constituent parts of a build system and illustrates them by means of a realistic build system, i.e. the GNU Build System. Important understandability and scalability problems of GBS and build systems in general are identified which are not only harmful for understanding and re-engineering of a build system, but also for co-evolution of source code and the build system. We zoom in on the latter phenomenon by distilling and postulating four previously undocumented "roots of co-evolution" (RC1, RC2, RC3 and RC4), which correspond to links between source code and the build system distilled from existing work on programming languages and build systems. Finally, we consider the additional influence of introduction of AOSD technology on the four roots and how this predicts changes in the build system as a consequence.

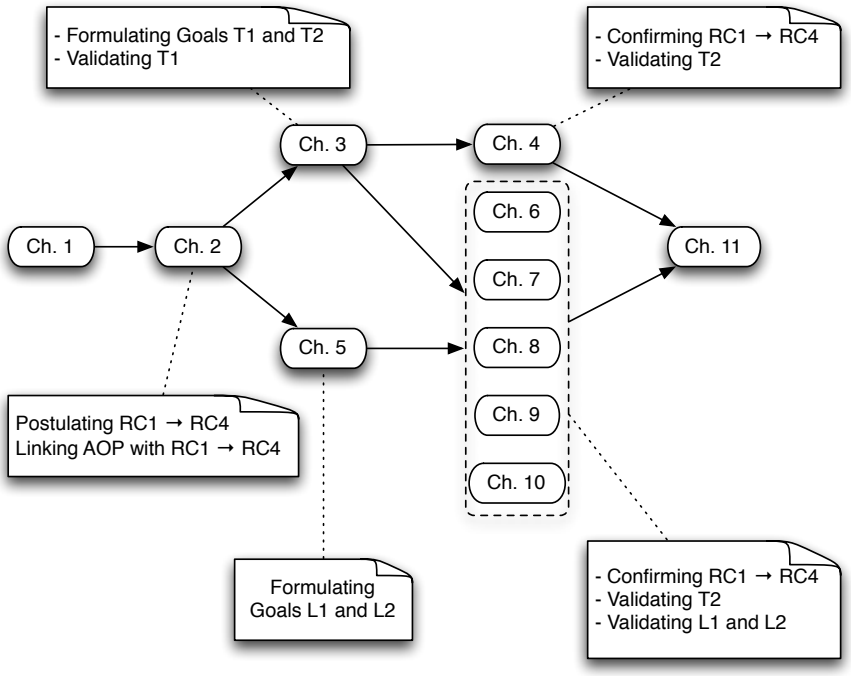


Figure 1.2: Dependencies between the chapters of this dissertation. Chapters are annotated with the role they play in formulating or validating the roots of co-evolution (RC1 →RC4) and the goals for tool (T1 and T2) and aspect language (L1 and L2) support. The five AOP case study chapters in the dashed rectangle can be read in parallel.

Chapter 3 proposes a number of requirements tools should fulfil to foster build system maintenance (goal T1) and to help in understanding and managing co-evolution of source code and the build system (goal T2). We show that existing tools do not satisfy these requirements and propose a new framework, MAKAO. Based on results from three case studies, we show how MAKAO is able to deal with build system problems (goal T1).

Chapter 4 applies MAKAO to find experimental evidence of the four roots of co-evolution in the Linux kernel build system, from the first kernel releases until now. We explicitly focus on the intrinsic complexity of the build process instead of on how the users interface with it. First, we calculate various simple, but relevant metrics of the build system. These enable us to identify important evolution steps in the build system. One of these steps, the migration from the Linux 2.4 to the 2.6 kernel, is investigated in detail. The evolution of the Linux kernel build system turns out to be driven by the need to overcome problems caused by co-evolution

of source code and the build system.

Chapter 5 distills requirements for an aspect language to deal with legacy systems (goal L1) and to facilitate the co-evolution of source code and the build system (goal L2). These requirements are used to evaluate existing aspect languages for C, followed by the design and implementation of Aspicere, our own aspect language for C. It is based on LMP [237, 30], and has explicit means to interface with the build system, i.e. the build system is integrated with the aspect language. We discuss Aspicere's aspect language and weaver features, and compare Aspicere with a recently proposed aspect language for C aimed at industrial application to validate our requirements (goals L1 and L2).

Chapter 6, Chapter 7, Chapter 8, Chapter 9 and Chapter 10 present five case studies with AOP in legacy systems. The first two cases (Kava and Quake 3) use aspects to reverse-engineer an existing legacy system. The other three re-engineer a legacy system, by extracting exception handling into aspects, by supporting the implementation of an architectural description language (ADL) [168] and by extracting conditional compilation into aspects. These cases are used to validate the four roots of co-evolution in the face of AOSD, as well as to validate the degree to which MAKAO and Aspicere satisfy goals T2 and L1/L2 respectively.

Chapter 11 discusses the degree to which we have been able to address the six research questions of this dissertation, and proposes interesting directions for future work.

Everything has a Makefile.

Bram Adams

2

Conceptual Evidence for Co-evolution of Source Code and the Build System

THIS chapter[1] introduces the notion of a build system as it has evolved throughout the years (Section 2.1). We illustrate the resulting build model by means of the open source GNU Build System. Section 2.2 then describes a wide range of problems attributed to build systems in general, and "make"-based systems in particular. The extent of these problems becomes apparent in Section 2.3, where we investigate traces of co-evolution of source code and the build system in existing research. We distill four links between source code and the build system, named "roots of co-evolution", which describe the ways in which source code and build system interact in traditional software systems and which together form conceptual evidence of co-evolution of source code and the build system. Afterwards, we introduce AOSD and show for each root of co-evolution how AOP technology theoretically engenders more co-evolution issues. The key message of this chapter is that source code and the build system co-evolve and that tool support is required to manage this.

2.1 The Build System and its Responsibilities

We first discuss the historical evolution of build systems (Section 2.1.1), before presenting the build system model we use throughout this dissertation (Section 2.1.2).

[1] Small parts of this chapter are based on [3].

To illustrate these concepts, we consider the comprehensive open source GNU Build System in Section 2.1.3 and we situate the build system in the broader context of software development (Section 2.1.4).

2.1.1 History of Build Systems

Conceptually, a "build system" is the collection of tools and prescriptions which give birth to an application, i.e. turn a set of text files and data into a running system. Before 1975 [84], most developers wrote their own ad hoc build and install scripts to automate the tedious work of repeatedly invoking compilers in the correct order with the right flags and input files, linking compiled files into the desired libraries and executables, and moving the generated files to the appropriate system locations or packaging them up for distribution. This scripted approach was not perfect either. First, the scripts, typically written in shell command languages, were not able to scale with the growing complexity of build specifications. Finding out where and when new compilations ought to be inserted proved to be a hassle, and many build errors sneaked in. On top of this, there was no straightforward way to provide incremental compilation, i.e. to only re-compile the files with changes.

In 1975, Feldman has introduced a dedicated build tool named "make" [89]. To express the sequence of commands needed to produce the desired build artifacts, "make" still attaches an imperative list of shell commands ("build recipe" or "command list") to each build product. However, in order to compose the various artifacts into higher-level ones in the right order, "make" has introduced a declarative, rule-based specification. "Rules" are responsible for explicitly capturing "dependencies" between "build targets"[2] (executables, object files, etc.), and they are listed in textual "makefiles". These provisions tackle the scalability problems of shell scripts. Incremental compilation is facilitated through a time stamp-based heuristic which ensures that a rule's target is only (re)built if the target does not yet exist or if at least one of its dependees is newer. "make"'s innovations have influenced lots of other build tools like e.g. Ant[3] or SCons[4]. Traditional "make" systems are still in wide use today.

Originally, a build system consisted solely of a build layer based on e.g. "make". This was insufficient to cope with the growing number of platforms and product variants modern software had to be capable of. Portability and configurability of software had become an important concern as well. The former manifests itself in different locations of system header files, names of compilers or compiler flags used. The source code itself contains some portions which should not always be compiled, like bug fixes for a particular platform or to work around limitations

[2] When we refer to "the" build target of a rule (or the "dependent"), we refer to the target which results from a build rule. The targets it depends on are "prerequisites" or "dependees".

[3] `http://ant.apache.org/`

[4] `http://www.scons.org/`

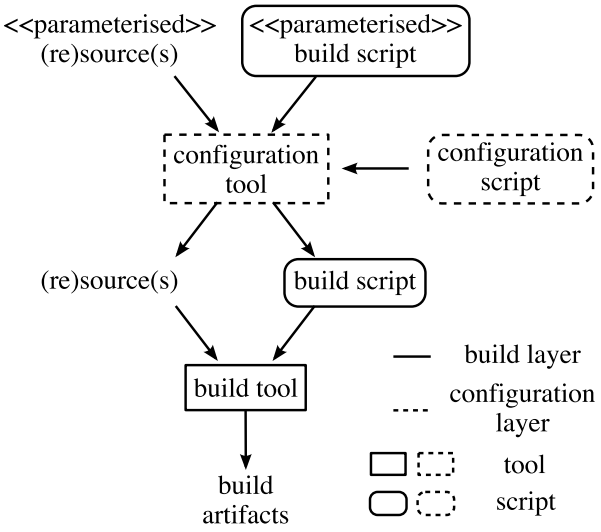


Figure 2.1: Build system architecture.

of an API. Hence, configurability has to be interpreted in its widest sense, ranging from selecting the set of source files to build, to the specific variant of an algorithm which can be conditionally selected within the source files. These functionalities have become the responsibility of dedicated configuration tools and scripts which together form the configuration layer, i.e. a higher-level layer on top of the build layer.

The next section discusses the resulting build model in detail.

2.1.2 The General Build System Model

Figure 2.1 shows the core architecture (often hidden behind an IDE) of modern build systems which we adopt throughout this dissertation. A build system takes care of two things (Figure 2.1):

- deciding which components should be built, selecting the desired features to compile and establishing any platform-dependent information needed to do so;
- incrementally building the system taking dependencies into account.

The first task is performed by the so-called “configuration layer”. A configuration tool takes possibly parametrised build scripts and resources like the source code or data files, and resolves all build parameters based on user input and automatic detection of build platform characteristics. The configuration script defines

the build parameters of interest and specifies how their values should be obtained. There are various ways through which the values of configuration parameters can be passed to the build layer and source code. Section 2.1.3.1 explains these in detail. Selection of the right files to compile may depend on the developer's orders, constraints between files, recently committed versions of the source code, etc. The goal of the configuration layer is to instantiate the resources and build scripts.

The "build layer" is responsible for the second task. The build tool does the ground work based on the build dependency specification within the build script. Influenced by "make" [89], build tools typically act like backward chainers. For a particular build goal, each dependee at a time is investigated recursively until a build rule is reached for which all dependees have been checked or which does not have any dependees. If either the build goal at that level does not exist or a heuristic based on time stamps or check sums is able to determine that at least one of its dependees is newer, the command list (or "build recipe") is executed to construct the desired build goal. If there is no build rule for the current build target, the file system is searched for this file instead. If this search fails, an error is generated. Conceptually, the fact whether or not a goal has been rebuilt is propagated up to the rule immediately depending on that goal. This process recursively continues until the user-defined build goal is either rebuilt, deemed up to date, or an error has occurred along the way. In the latter case, the build engineer usually can configure beforehand whether a build should try to continue after an error or give up immediately.

As Feldman shows [89], this backward chaining approach is based on a Directed Acyclic Graph (DAG) in which nodes correspond to build artifacts like files, executables, etc. and in which the edges correspond to build dependencies. There is no prescribed way to include command lists in the DAG, however. Also, the order of a given target's dependees in the build specification disappears in a DAG, unless edges can be labeled somehow, e.g. via time stamp information. Using the DAG metaphor, incremental compilation naturally maps onto a heuristic-based topological sort algorithm.

To conclude, the build system's configuration layer filters out irrelevant source code and build logic for the platform at hand, and then passes control to the build layer, which exploits the build dependencies to generate a working application in a correct and fast way. Some build systems make a very clear distinction between the configuration and the build layer, while others blend them together or even generate the build layer automatically from the configuration specification. The latter occurs within the GNU Build System, which we discuss in the next section as a prototypical example of the presented build system model.

2.1.3 The GNU Build System (GBS), an Archetypical Build System

The GNU Build System (GBS for short) is an amalgam of tools which together form a very powerful, but complex open source build system. Most of its components can be used in isolation, but there is good integration support between them. GBS's goals are:

- to make source code portable across as many platforms and operating systems as possible;
- to make it easier for developers and users to write and use build files by enforcing the GNU Coding standards[5];
- to provide convenient and safe packaging support;
- to offer platform-independent installation functionality.

Figure 2.2 gives a graphical overview[6] of GBS's major components (ellipse shapes), although libtool[7] and gettext[8] are not shown here (they are used in the example of Appendix A). The heart of GBS is formed by "autoconf"[9], which forms GBS's configuration layer. The build layer is automatically generated by "automake"[10]. The other three programs ("aclocal", "autoheader" and "autoscan") are support tools. Notice the dashed line in the middle of Figure 2.2. It emphasises the firm design decision within GBS to differentiate between build machinery and files needed by the source code developers (upper half), and the more modest build tools required by open source users to compile the software on their system. These clients do not necessarily need autoconf nor automake. Instead, they receive the generated Bash script "configure" and a couple of template files ("*.in") which together generate a fully instantiated build layer for the build platform at hand. This build layer consists of makefiles for the "GNU Make" build tool.

This section discusses the major components of GBS, i.e. autoconf, automake and GNU Make with the aim of highlighting the responsibilities of a build system and to show the complex interactions between source code and build systems. During our explanation, we refer to an example system presented in detail in Appendix A. It is a simple C system comprising of three source files (Figure A.1, Figure A.2 and Figure A.3) spread over two directories ("src" and "lib"). For the

[5] `http://www.gnu.org/prep/standards.html`

[6] This illustration originally has been made by René Nyffenegger and published at `http://www.adp-gmbh.ch/misc/tools/configure/files_used.html`. He has kindly given us permission to enhance and use his schematic overview in this dissertation.

[7] `http://www.gnu.org/software/libtool/`

[8] `http://www.gnu.org/software/gettext/manual/`

[9] `http://www.gnu.org/software/autoconf/manual/`

[10] `http://www.gnu.org/software/automake/manual/`

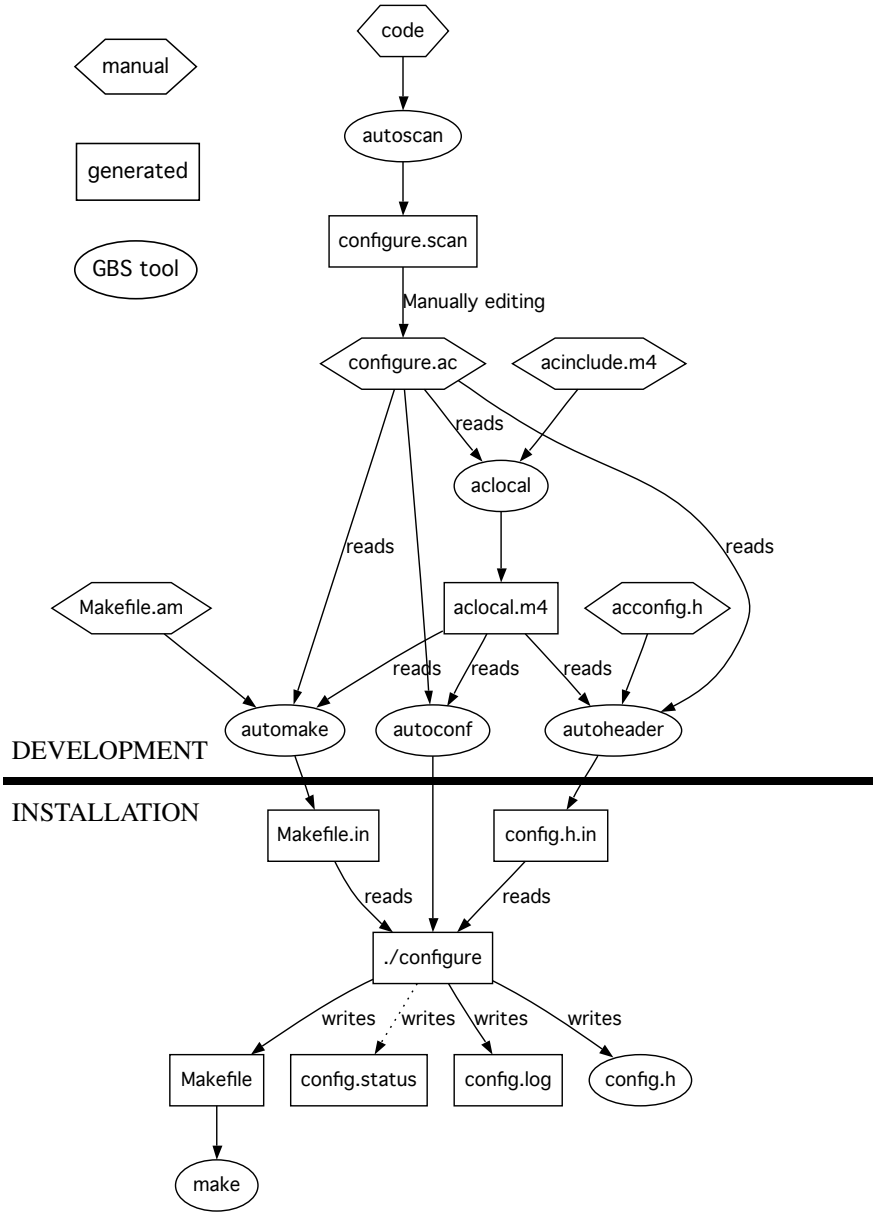


Figure 2.2: Schematic overview of the most important files used within GBS, reproduced and modified with permission by René Nyffenegger. Edges marked with "reads" represent secondary input files used alongside the main input specification (edges without label).

```
AC_INIT([amhello], [2.0], [bug-report@address])
AC_CONFIG_AUX_DIR([build-aux])
AM_INIT_AUTOMAKE([foreign])
AM_GNU_GETTEXT_VERSION([0.14.5])
AM_GNU_GETTEXT([external])
AM_CONDITIONAL(GETTEXT_INSTALLED, [test x$USE_NLS = xyes])
AC_SUBST([USE_NLS])
AC_PROG_LIBTOOL
AC_PROG_CC
AM_CONFIG_HEADER([config.h])
AC_CONFIG_FILES([Makefile lib/Makefile src/Makefile \
                 po/Makefile.in lib/say.c m4/Makefile])
AC_OUTPUT
```

Figure 2.3: Example configure.ac from the example system in Appendix A (this is a copy of Figure A.4 on page 320).

```
#include <config.h>
#include <stdio.h>

#ifdef HAVE_GETTEXT
#include "../src/gettext.h"
#define _(string) gettext (string)
#else
#define _(string) (string)
#endif

void say_hello (void){
  puts (_("Hello World (with%s gettext)!"),
        ("@USE_NLS@"=="yes"?_(""):_("out")));
  printf (_("This is %s.\n"), PACKAGE_STRING);
}
```

Figure 2.4: Source file template "lib/say.c.in". This is a copy of Figure A.3 on page 320, which illustrates the usage of the generated config.h.

reader's convenience, snippets of the example GBS files are duplicated whenever appropriate.

2.1.3.1 Configuration Layer: Autoconf

The original "autoconf" was created in 1991 by David J. Mackenzie [234]. Its core goal is to provide portability across various operating systems and platforms. To achieve this, all known compatibility problems of libraries, tools or APIs across many platforms have been collected and hidden behind a macro-based facade. The

idea is that developers specify their needs, e.g. a C compiler or a library providing some function, and that "autoconf" finds the particular tool or artifact installed on the current system. For this, it is vital that "autoconf" can identify the precise version, location and name of any tool, operating system or library of interest. Instead of merely looking at version numbers, the tools are invoked with mini-programs which exercise the features desired by the developer. This provides robust version identification, which often even works on unknown systems.

In Figure 2.2, we observe that "autoconf" expects two inputs: "configure.ac" and "aclocal.m4". The former contains the configuration specification. Usually, an initial version has been generated by "autoscan", which scans the source code for potentially non-portable logic. Only one "configure.ac" is needed for an entire project, and it can reference all "Makefile.in" build script templates of the build layer. Figure 2.3 shows a simple example[11]. Each line contains an expansion of a GNU "m4"[12] macro which checks for the existence of a tool, library, etc. Each system can provide its own macros on top of the standard ones by collecting them in a library file, "aclocal.m4", using "aclocal". To avoid name clashes, prefixes like "AM_" ("automake") or "AC_" ("autoconf") are used. The macro on line 1 contains some meta data about the program, e.g. the name of the system and the current version, while line 2 specifies that generated scripts for internal use have to be generated within a specific directory. Line 9 asks "autoconf" to look for a C compiler, whereas line 13 demarcates the end of the configuration specification.

A crucial question is how configuration choices from within "configure.ac" are communicated by the configuration tool to source code and build scripts. GBS offers five possibilities:

1. Passing conditional compilation flags via the invoked compiler commands.
2. Generating a header file during the build with the active conditional flags, combined with conditional compilation in the source code.
3. Using conditional build logic in the build scripts (or templates).
4. Parametrising source code and/or build scripts with build-time parameters.
5. Using environment variables to override certain paths or commands.

The first option is the simplest one, but complicates readability of the command line and may on some systems even exceed the maximum length of shell command invocations. The second option solves this. "config.h.in" is a generated header file (line 10 on Figure 2.3) which initially undefines all occurring compile-time constants (see Figure A.8 on page 322). This header file is automatically generated by the "autoheader" tool based on "configure.ac" and may possibly be

[11] A backslash at the end of a line is used to continue the current statement on the next line.
[12] `http://www.gnu.org/software/m4/manual/`

```
1 AM_CPPFLAGS = -DLOCALEDIR=\"$(localedir)\"
2 lib_LTLIBRARIES = libhello.la
3 libhello_la_SOURCES = say.c say.h
4 libhello_la_LDFLAGS = -version-info 0:0:0
5
6 localedir = $(datadir)/locale
7 DEFS = -DLOCALEDIR=\"$(localedir)\" @DEFS@
```

Figure 2.5: Automake build script template lib/Makefile.am. This is a copy of Figure A.6 on page 321.

customised via an "acconfig.h". When executing "configure" on the client machine, the relevant flags are defined in the resulting "config.h". Any source file interested in this configuration information just needs to include this header file[13] and use it in conditional compilation checks. This is illustrated in Figure 2.4 on lines 4–9. If the internationalisation tools (`HAVE_GETTEXT`) are not installed, the specific `gettext` function is not used (line 6). A similar mechanism can be used inside build scripts (see lines 6–10 in Figure A.7 on page 321). The fourth method has also been applied in Figure 2.4. This code snippet shows the "lib/say.c.in" file, i.e. a source file template (as its name ends in ".in") which is parametrised with "autoconf" parameters like `@USE_NLS@` on line 13. The substitution macro on line 7 of Figure 2.3 instantiates a "lib/say.c" file in which `@USE_NLS@` will either be replaced by `yes` or `no`. The compiler will likely be able to optimise away the resulting string comparison. The fifth and final technique passes configuration data to the build scripts by overriding certain commands and paths via environment or makefile variables like `CC`, `CPPFLAGS`, etc.

The goal of "autoconf" is to generate a Bash shell script ("configure") at the client site. This takes the source code and build script templates and instantiates them based on the configuration choices. Besides this, "configure" also generates a script "config.status" and a log file. "config.status" can be seen as a partially evaluated version of "configure" with all configuration choices hardcoded. The instantiated build scripts can then be processed by GNU Make (see Section 2.1.3.3). Before explaining GNU Make, we first consider the high-level specification of build dependencies using "automake".

2.1.3.2 Build Layer Generation: Automake

"automake" is a tool set started by David J. Mackenzie in 1994 which offers a high-level, domain-specific language to specify build layer dependencies and compilation tools [167]. It automatically translates high-level specifications ("Make-

[13] Because compilation happens completely after configuration has ended (see the "make" node on Figure 2.2), the compiler will always find this header file.

```
1 make_OBJECTS = ar.o arscan.o \
2                commands.o dir.o ... hash.o
3 make$(EXEEXT): $(make_OBJECTS)
4         @rm -f make$(EXEEXT)
5         $(LINK) $(make_LDFLAGS) \
6                 $(make_OBJECTS) \
7                 $(make_LDADD) $(LIBS)
8 ...
```

Figure 2.6: Example GNU Make makefile.

file.am") into portable, standardised GNU Make (Section 2.1.3.3) build scripts, similar to e.g. the work of Buffenbarger et al. [37]. When using the full GBS, the output of "automake" usually is a parametrised makefile ("Makefile.in"). In the build specification, it suffices to express what the desired build output should be (binary, library, header file, script or data) and which source code files are needed. One can also specify compiler flags, add custom build rules which need to be transferred as is to the generated makefiles, and specify distribution and installation requirements. Source file dependencies related to `#include` statements are automatically extracted based on a common trick (see 4.5.3.2), such that one only needs to specify the high-level dependencies.

The sample GBS build system in Appendix A contains three "Makefile.am" files, one per directory. Figure 2.5 shows "lib/Makefile.am". Line 2 specifies that the result of this directory's build will be a "libtool" library named "libhello.la". It consists of the code of "say.c" and "say.h" (line 3) and is versioned in a default way (line 4). Notice that name clashes are avoided by tying the `SOURCES` and `LDFLAGS` variables specifically to `libhello.la` instead of declaring them globally. The "Makefile.in" which will be generated by "automake" is parametrised by `@DEFS@` on line 7 and hence needs to be processed by "autoconf" to obtain a GNU Make build script. We discuss GNU Make in the next section.

2.1.3.3 Build Layer: GNU Make

GNU Make [218, 167] currently is the most popular "make" implementation and, in fact, build tool, both in industry and in the open source community. More powerful build tools have failed to overthrow "make". It is generally believed that this is because of "make"'s conceptual simplicity [89], despite the flaws we discuss in Section 2.2.

Figure 2.6 shows an example makefile snippet to compile GNU Make itself[14].

[14] GNU Make is implemented in C and needs an older version of itself to bootstrap.

On lines 1–2, a makefile variable named `make_OBJECTS` is defined as a list[15] of several object files, i.e. compiled versions of a C or C++ implementation module. Lines 3–7 represent a build rule which specifies that the target called `make$(EXEEXT)` depends on the list of object files denoted by variable `make_OBJECTS`. To build `make$(EXEEXT)`, the command list (build recipe) on lines 4–7 should be executed. Note that variables like `$(EXEEXT)` and `$(LINK)` correspond to the fifth way of passing information from the configuration layer. On Windows systems, `$(EXEEXT)` will resolve to "exe", but on Linux or OSX platforms to the empty string instead.

GNU Make interprets makefiles based on Feldman's time stamp-based heuristic. Target `make$(EXEEXT)` only needs to be (re)built by its recipe if it does not yet exist or if at least one of its dependees is newer. This mechanism is transitive: for each dependee D of `make$(EXEEXT)` (i.e. the files listed on lines 1–2), D should either be a physical file or the build target of another rule. Each line in the command list is executed in a separate shell.

GNU makefiles can contain definitions, expansions and invocations of macros and functions. As such, GNU Make contains a powerful, but complex string processing facility. The `eval` function e.g. allows to evaluate any valid makefile shell command. GNU Make provides sophisticated control of variables, as it makes a distinction between eager ("simple" variables) and lazy assignments ("recursive" variables). If the := operator is used, the right hand side is evaluated immediately and the resulting value assigned to the simple variable on the left hand side. In the case of =, the right hand side is stored as a string. Only when the value of the recursive variable on the left hand side is explicitly asked for, the stored string is evaluated using the by then established environment. This can be applied to implement highly sophisticated build logic (see Section 4.5.3.2) and also leads naturally to a top-down approach for specifying a build script. High-level targets are described on top of the makefile and gradually more low-level ones can be added below. Finally, makefile snippets spread across different directories can be included literally into a given makefile, similar to C's `#include` statement. This is often used in combination with conditional definition of rules (cf. conditional compilation in C).

To technically support these advanced features, the execution of GNU Make is divided in two phases:

1. the makefile and all its included makefiles are processed to load variables and rules into "make"'s internal database[16] and to create the build DAG;

2. the DAG stored in the database is analysed, required targets are determined and command lists are executed.

[15] A backslash at the end of a line can be used to continue the current statement on the next line.

[16] Simple variables have been assigned a value, while recursive ones contain the corresponding string definition.

These two phases actually correspond to two builds, one at the meta-level and one at the base level. The latter one does the actual construction work, as described in Figure 2.6, whereas the former needs to bring the makefile itself up-to-date. The included makefiles are its dependees. This means that if there is a build rule specified for one of those dependees, it will be used to bring the dependee and (transitively) the main makefile up-to-date. If this happens, the internal "make" database is cleared and the first phase is re-executed until all dependencies of the main makefile are up-to-date (or an infinite loop occurs if the dependees are rebuilt on each re-execution). Only then, the second build phase will execute.

Before we finish the discussion of GBS, we briefly consider alternatives of "make" and the areas in which they are used.

Alternatives for GNU Make This section first discusses the applicability of "make" for modern Java systems. Then, we consider various build tools which improve in some way on "make".

Many alternatives for "make" and GNU Make have been proposed, but none of them have been able to overthrow the original, except in the case of Java applications. There are two reasons why "make" is not very well-suited for Java. First, most Java compilers are much more coarse-grained than C or C++ compilers. Because they cache symbols extracted from previously compiled, imported classes instead of preprocessing source code [167] and they also defer most optimisations to the JIT compiler, they are fast enough to build an entire project with a single execution of the Java compiler. By way of comparison, Mecklenburg [167] claims (backed by Ammons [13]) that in one case compiling 500 Java files took only 14 seconds compared to a build time of five seconds for one Java file. A traditional "make" approach would require 500 compiler invocations instead of one, which means that the speedup of skipping certain compilations based on time stamp data is completely lost. The second reason for "make"'s unsuitability for Java applications is the fact that one Java file not necessarily leads to one class file, because nested and inner classes yield extra class files. This fundamentally clashes with "make"'s one-target-per-rule principle, as pointed out by Ammons [13].

For these reasons and also for the drawbacks listed later in Section 2.2, Apache Ant[17] (Another Neat Tool) is the de facto build tool in the Java world [181]. It was released in 2000 by James Duncan Davidson as an extensible, platform-independent build tool for Java. Build scripts are XML documents which describe dependencies between targets. However, contrary to "make" these dependencies only specify a pre-order tree traversal. There is no time stamp-based or other heuristic attached to them. Hence, Ant targets do not map onto files, but rather represent a collection of tasks or GNU Make phony targets [13] (see 4.5.3.2). These tasks are implemented by Java classes. The `javac` task e.g. compiles a set of Java

[17] `http://ant.apache.org/`

source files if they are newer than their class files or the latter do not exist.

Other build tools each improve in one or more ways on "make". Javamake [61] is a Java build tool based on a project database with relevant dependency information extracted from the compiled classes. Based on the semantics of changes, it determines whether recompilation of source files is needed or not. SCons[18] uses Python programs as build scripts and MD5 check sums instead of time stamps. Builder [207] persists the source code and derived artifacts in a database. The resulting build DAG can be partitioned to prevent unnecessary recompiling and a set of generic, enforceable constraints can be used to check the reliability of the compilation and incremental linking. DERIVE [206] is built on top of a Prolog engine, file system and relational database. Its main objective is to manage the enormous variability within the build process. Source files and their attributes are modeled as facts, configurations as rules and compilers and other tools as functions over byte strings. Rules are referential transparent, as new versions of components are added to the knowledge base instead of destroying old ones. Memoisation, partial evaluation and abstract interpretation are used to automatically detect derivation dependencies and potential concurrency between build actions.

Other build tools have more semantic checks. Odin [47] is based on a store of persistent, typed software objects (files, strings or a list) and a set of typed tool specifications. One can specify object derivations between types of objects at once instead of on a file by file basis. The directory structure does not play a role anymore, as the forest of derived objects maintains (possibly parametrised) links between derived objects and atomic ones. Implicit build dependencies within parametrised commands can be uncovered such that intermediate results can be reused more effectively during derivation. The Odin Request Language is an imperative OO language to interactively query the store. Even error messages or warnings are reified as objects available for further querying. Hence, Odin enables developers to program with derived objects, i.e. programming-in-the-large [60] (see Section 2.3.4.2).

There are also some tools which try to improve on GNU Make's support for parallel builds, i.e. running multiple commands at once. The main bottleneck for this is finding an optimal, safe scheduling of build tasks across the machines, taking into account dependencies between artifacts and cache support. The best-known tools for this are "distcc" and "ccache", both invented by the open source Samba project[19]. "distcc" uses a compiler on the local machine to preprocess each output and send the preprocessed code to any available machine for compilation [167]. "ccache" caches object files and can be easily integrated in a build by replacing the compiler by `ccache compilername`. This even speeds up non-distributed builds.

[18] `http://www.scons.org/`
[19] `http://www.samba.org/`

2.1.3.4 Additional GBS Components

To close the discussion of GBS, we consider two other components which are often used with GBS: “libtool” and “gettext”. The former was designed by Gordon Matzigkeit in 1996 [234]. It abstracts over the way how shared libraries are created and used on various operating systems. Instead of using a platform-specific extension, “libtool” works with “.lo” and “.la” files which contain meta data, versioning info and a link to the actual object files or libraries. The required configuration logic for using “libtool” is shown on Figure 2.3 (line 8) and Figure 2.5 (line 2).

“gettext” is responsible for enabling integration of native translations and support for locale concepts like currency and date format into the build system (internationalisation), as well as for insertion of the translations themselves (localisation). These functionalities have to be integrated in the configuration logic (lines 4–5 on Figure 2.3 and the locale and `LIBINTL` code in Figure 2.5 and Figure A.7 on page 321) and the source code. As described above, a specific API needs to be included in the code base and called for every string appearance (lines 5–6 in Figure 2.4). “gettext” makes a build system robust against cultural variability.

2.1.4 Roles Played by the Build System

The explanation of the GNU Build System gives a more concrete idea of the core functionalities of a build system. However, a build system interacts with various other facets of software development. This section briefly reflects on these interactions in order to be able to situate the build system better in the software development process and at the same time to stress its important role. We consider continuous integration, IDEs, configuration management, deployment, release management and variability management.

2.1.4.1 Continuous Integration

There is a close connection between the build system and test suites [181]. Development processes like Extreme Programming heavily rely on extensive test suites which are ideally exercised on each change to the source code. As long as developers are not forced somehow to run tests, test-driven development may be compromised in tough situations (when it actually is needed the most). The build system solves this, as it is the ideal place to enforce unit testing during development and to make it easily repeatable.

This idea has been taken to another level with the advent of continuous integration servers. These systems regularly check a source code repository to detect new changes to the source code. On a change, the system is built and the test suite exercised. If a test fails, developers are automatically notified, otherwise this ver-

sion of the source code can be tagged and possibly considered as a new release of the software. The build system plays a crucial role in this scenario.

2.1.4.2 Integrated Development Environments

To increase developer productivity and standardise the development tools, vendors have combined compilation tools, editors, documentation browsers, unit test frameworks, build systems, etc. into one "Integrated Development Environment" (IDE) [88, 199]. Nowadays, almost everyone uses an IDE for day-to-day development in C++ or Java, whereas developers of legacy systems are often limited to command-line environments. Although details of build and configuration scripts are hidden behind user-friendly forms, IDEs typically have their own build system implementation. Often, this is heavily integrated with the compiler and other tools like debuggers, as illustrated by Smalltalk-80 [101], Borland's Turbo Pascal[20] or IBM's Eclipse[21]. This facilitates language-specific optimisations of build speed and ease of specification, but may pin the developers on these tools [95]. More importantly, Mecklenburg [167] states that IDEs are good for small-scale or localised development, but do not scale for bigger systems and are not practical for unattended builds.

2.1.4.3 Software Configuration Management (SCM)

Software Configuration Management (SCM) takes care of "the control of the evolution of complex systems" [81]. This goes far beyond the build system model we have presented in Section 2.1.2, as the latter assumes that there is exactly one version of each source code file available to the build system. In reality, many developers contribute to the code base of a system and some kind of version control system [196] is used to manage this. This maintains a history of all changes to source code files (these are named "revisions"), enables multiple independent branches of development, provides means to merge and contrast changes, keeps meta data about the intent and owner of modifications the changes, etc. As a consequence, developers often need to mix source code files which belong to different revisions, e.g. because the latest versions of some files are not stable. On the other hand, the results of the build process, i.e. released binaries or libraries need to be stored in the versioning system too for maintenance and bug fixing purposes. These activities are sometimes aptly called "programming-in-the-many" [81].

A lot of research has been done involving SCM systems [81], often closely related to the development of IDEs (see Section 2.1.4.2), but adoption of these techniques has been slow to catch up. Many different SCM techniques have been proposed [227, 42, 143, 147, 165, 47], but SCM systems have very long ignored

[20] `http://en.wikipedia.org/wiki/Turbo_Pascal/`
[21] `http://www.eclipse.org/`

these developments, and have kept on relying on low-level abstractions like files, directories or makefiles [84, 81]. Although this has lowered the barrier for adoption of SCM, these systems have not immediately brought the improvements they were intrinsically capable of, nor have they clearly established their advantages over the build system model of Section 2.1.2. As a consequence, legacy systems often contain custom SCM systems using low-level technology.

2.1.4.4 Software Deployment

Deployment consists of packaging and installing compiled software artifacts, i.e. it is one of the possible phases following a standard build. Packaging is concerned with wrapping source code and/or compiled artifacts into some kind of abstraction with a clear notion of identity, configuration interface and dependencies on (versions of) other packages [56]. Installing entails resolving missing dependencies, applying on-site configuration and (if needed) building package elements which have been distributed as source code. Hence, there is a clear link between software deployment and the build system.

The relation between deployment and build systems has been investigated by various researchers [62, 64, 66, 63, 65, 117, 47]. Dolstra et al. e.g. have designed a build manager named "Maak"[22] based on a lazy functional specification language for build variants. A powerful, parametrised import construct enables to express configuration dependencies via an abstract name. This enables Maak to determine whether the referenced package already exists in binary form (in cache) or whether (re)build is necessary. As such, a binary package in fact corresponds to a partially evaluated source distribution with respect to the target build platform [206]. Maak's successor, Nix[23], improves on this by using cryptographic hashes as identifiers of dependencies [66] to prevent unresolved references and reduce interference between different dependencies. Files within components are scanned for these special strings to recursively detect all dependencies of components. The principle of "maximal sharing", i.e. reuse of common dependencies based on their identifier, enables seamless transition between binary and source distribution. Finally, Nix is based on a "store" [66, 63] which contains all installed components together with any intermediate build results, the source code and declarative store expressions [65] which describe how the build results were obtained. Execution of build actions is transactional. User environments are trees of symbolic links to activated components for a given user's configuration. These links are the roots of a garbage collector pass which on demand deletes unused paths from the store. This extra level of indirection enables atomic upgrades, rollbacks and leads to deterministic and reproducible configurations. This work shows how packaging and installing,

[22] `http://people.cs.uu.nl/eelco/maak/`
[23] `http://nix.cs.uu.nl/`

although closely related to the build system, requires a lot of extra provisions on top of the build model of Section 2.1.2.

2.1.4.5 Release Management

Deployment is closely related to release management, which consists of configuring products in a flexible way, keeping track of which version of which source module has been packaged in which release ("bill-of-materials") and minimising the number of needed upgrade actions on a new release [229, 230, 231]. This requires thorough integration with the build system, e.g. to manage product variability [229] (see Section 2.1.4.6) or to verify whether new releases of a component or its dependencies have been made and they pass all their tests [230]. Because our build system model is not capable of this [231], van der Storm has built a continuous integration system (Sisyphus) which polls the version control system for changes, updates dependency and configuration information of releases which pass all integration tests in a Release Knowledge Base (RKB), uploads the release to an update server accessible by clients and enables incremental upgrades by only distributing changes.

2.1.4.6 Variability Management

Both packaging and release management rely on flexible configuration specifications which are able to manage the often overwhelming variability in software. This is studied within the product line research community[24]. Again, there is a clear relation between build systems and product lines [227]. In fact, a product line can be considered as a very advanced configuration layer (more on this in Section 2.3.4.4) with a high-level specification language able to express any possible source of variability, feasible compositions, dependencies between various variables and more powerful means to enforce a desired configuration. The latter refers to the fact that besides build-time composition, many product lines also require run-time adaptation [55, 233, 56, 64]. Unfortunately, there is no standard configuration interface for variability across all phases of a program's lifetime. A combination of configuration-, build- and run-time technologies is needed to implement a product line, as illustrated by van Deursen et al. [233, 56] or by the Linux kernel build system [55], which uses rather low-level technology.

[24] `http://www.softwareproductlines.com/`

```
1 all: A.class app.jar
2 A.class: A.java
3   javac A.java
4 app.jar:
5   jar cf $@ A.class `ls ../classes/*.class`
```

Figure 2.7: Makefile with implicit dependencies.

2.2 Understandability and Scalability Problems of Build Systems

This section discusses an extensive number of understandability and scalability problems associated with "make" (Section 2.2.1), GNU Make (Section 2.2.2), GBS (Section 2.2.3) and the application of build systems in general (Section 2.2.4). The first three sections only briefly highlight various problem areas, whereas the fourth section goes into more detail. These build problems show that tool support is needed to understand and maintain build systems. In Chapter 3, the build problems are used to distill a number of requirements for dedicated tool support for design recovery and maintenance of build systems.

2.2.1 Problems with "make"

This section discusses two understandability problems associated with "make", and four scalability issues.

2.2.1.1 Understandability

The existence of understandability problems seems at odds with the conceptual simplicity of "make", but this group of problems rather corresponds to omissions or limitations.

Implicit Dependencies Consider the three rules in the makefile of Figure 2.7. The first one (line 1) states that target "all" requires both a file named "A.class" as well as an archive "app.jar". The former is compiled by "javac" (line 3) from a Java source file named "A.java" (line 2). The rule for the archive (lines 4–5), however, does not explicitly specify its dependees behind the colon on line 4. Instead, the dependees are only mentioned in the rule's command list (line 5). In this case, the dependency of "app.jar" on "A.class" is named an *implicit* dependency and we call "A.class" an "implicit target". Even worse, "app.jar" has a number of implicit dependees which can only be resolved at build-time as they are the result of a shell command (between back quotes).

Implicit dependencies inevitably lead to inconsistent and incorrect builds. E.g. if the "app.jar"-target would be built directly, "make" will not try to remake "A.class", and will bundle the existing, possibly outdated version in the archive. Unfortunately, enforcing developers to explicitly list all dependees is not feasible, so tool support should assist developers in tracking the actual dependees down. Note that implicit dependencies are closely related to Singleton's [206] concerns about the presence of side-effects in the non-declarative command lists of build rules, i.e. effects which cannot necessarily be specified as dependent or dependee.

Time Stamps "make"'s time stamp-based updating algorithm cannot handle situations where a dependee is replaced by an older version, e.g. by reverting to an older revision. The dependent will not be incrementally remade, except when a full build is made. Other heuristics based on check sums or code formatters do not suffer from this problem.

2.2.1.2 Scalability

This group of problems is related to the (ab)use of "make" outside the "medium sized systems" [89] it originally was conceived for [84, 81].

External Dependency Extraction One of the biggest problems of "make" for scalability is the synchronisation between source code dependencies (e.g. from C `#include`-statements) and build dependencies. This is a problem of synchronisation between on the one hand source-level dependencies and on the other hand dependencies between build artifacts. "make" does not prescribe a way to deal with this. Manually updating build dependencies each time the source code has been edited, is not maintainable. Several tools have been designed — often integrated with compilers — to emit build script snippets based on the actual dependencies in the source code, but their integration with "make" often leads to redundant work. The myriad of dependency management tools and techniques, and its importance on developer productivity, build correctness and build speed nevertheless forms an important indication towards the existence of a co-evolution relation between source code and the build system.

Costly Incremental Compilation Heydon et al. [117] claim that the cost of incremental compilation using "make" is too high because it starts dependency analysis from the leaves of the DAG instead of pruning subgraphs via a pre-order traversal. Hence, the cost of this analysis is proportional to the total number of source files in the build instead of the number of changed ones. This is harmful for scaling up to big systems.

One-dimensional Dependencies Lamb [141] argues that "make" focuses on only one kind of dependency relation between dependent and dependees. As a result, this one kind of dependencies is used to express any kind of build relation such as "executables are built from object files", "source files need declarations of header files", "header files extend each other", etc. Unraveling these relations is necessary to avoid the presence of circular dependencies.

Traceability, Versioning and Variants "make" does not have provisions for tracing back from build products to the sources they were derived from, nor is there a means to manage different versions or variants of build artifacts [89, 206, 37]. Various descendants of "make" have been built to treat these problems, but often this has increased their complexity too much compared to "make". Hence, other mechanisms should be used as a complement to "make" to manage traceability, versioning and variants.

2.2.2 Problems with GNU Make

Modern implementations of "make" like GNU Make offer a number of workarounds for the aforementioned problems of "make", but in doing this a lot of understandability problems have been introduced.

2.2.2.1 Understandability

Syntax has Semantics GNU Make still relies on syntactical details like tabs, spaces (behind the colon) or dollar signs to convey semantics. This is extremely error-prone, but unfortunately the resulting error messages are very cryptic.

Advanced Language Features Advanced features like simple/recursive variables, the `eval`-command and the two-phase execution model make GNU Make a lot more flexible. As we will see in Section 4.5, macros and other features are even capable of building abstractions on top of the basic "make" features without incurring extra overhead [167]. Unfortunately, these features at the same time are new sources of confusion and misunderstanding. Variables can e.g. be used as dependent or dependee, or to guard conditional build logic, but erroneously using `:=` (simple variable) instead of `:` (recursive variable) can drastically change the semantics. Knowledge of the run-time value of variables is a necessity to deal with this kind of problems.

Precedence of Variables Another way in which variables are hard to understand, are the precedence rules which determine which value assigned to a variable prevails. The least precedence is given to environment variables. The next ones in the hierarchy are variables specified in the makefile, then those defined

on the command line, variables in the makefile marked as overriding another variable and finally rule-specific variables. These rules are complex to remember. Even worse, the combination with GNU Make's include facility and dynamically spawned "make" processes[25] makes it very hard to find a variable's roots. Again, knowledge of the run-time value of variables is needed to resolve this.

Debugging There is no built-in GNU Make debugger, although there are some external tools for this (see Section 3.3.4). GNU Make only generates a trace file with a detailed account of build decisions taken and for each "make" subprocess an overview of run-time values of variables. This trace file is voluminous and not well-structured. It is impossible to follow the flow of the build or to keep track of the location of targets in the build.

GNU Make signals errors in the makefiles at build-time, but the error messages are rather cryptic. Especially in the case of syntax errors (2.2.2.1) these messages are not precise. No warnings are given about redundant build dependencies, loops across "make" processes[26], etc. These kinds of "semantic" errors have to be detected manually from the trace file output.

2.2.2.2 Scalability

GNU Make has not solved some of "make"'s scalability problems.

Dependency Extraction Just as "make", GNU Make still has no built-in facilities for synchronisation of source code and build system dependencies. A popular workaround is to (literally) include for each source file a dedicated makefile snippet into the actual makefiles. These snippets contain specialised build rules which are typically automatically (re)generated in the first phase of GNU Make's execution. Because included snippets are actually dependees of the surrounding makefile (see Section 2.1.3.3) and are regenerated, there will be two executions of the first GNU Make phase, which is slow (it might even double the compilation time [167]). Other tricks have to be used to make management of build dependencies feasible (see 4.5.3.2).

Portability Portability of makefiles is a real issue. Command lists can use any Bash command or shell script and the `eval`-statement may even execute any string it is passed during the build. Unless the configuration layer succeeds in abstracting away platform-dependent program names, switches, paths, line termination, etc., build scripts are hardly portable. Mecklenburg [167] notes that developers should not "hassle with writing portable makefiles", but rather "use a portable make instead". In practice, a truly portable "make" is hard to find however.

[25] Variables need to be explicitly "exported" to pass them down to the new process.
[26] GNU Make is able to detect loops in one process.

2.2.3 Problems with GBS

GBS is pervasively used in the open source community. However, many developers have complaints about it and have even migrated to other build systems (more on this in Section 2.3.3.1). We present two understandability and three scalability problems.

2.2.3.1 Understandability

Complexity It is clear from the explanation in Section 2.1.3 that GBS provides a myriad of configurability techniques and a higher-level way of specifying configuration layer and build layer. Unfortunately, this complexity is also its biggest drawback. The combination of programming languages, macro system, the sheer endless list of macros, hidden caching mechanism and slowness of the tools result in tricky build errors which are hard to resolve. This problem is actually the most important one of GBS.

Traceability to Build Templates Traceability from errors in a makefile to the build template ("Makefile.in") of which the makefile is an instance, is hard to achieve [62]. This becomes worse if multiple build variants of the source code need to co-exist, as each one conceptually uses other specifications of dependencies between build artifacts. The only solution to model this is to have one "Makefile.am" tree per variant, but this is not maintainable [62].

2.2.3.2 Scalability

Although GBS is a very comprehensive build system, it lacks a number of facilities for scaling up to bigger systems.

Portability There are three portability problems with GBS. First, it heavily depends on Unix tools like the Bash shell. Second, despite the detection of tools based on provided features (Section 2.1.3.1), GBS in general can only be ported to another platform if macros have been checked to work for the specific versions of tools and APIs in use. Third, Mecklenburg [167] argues that the requirement to generate portable GNU Make code severely restricts the features of "automake". Its most advanced language feature is the append (+=) operator. However, Buffenbarger [37] claims that the "automake" language is still too difficult to understand and only supports a limited number of languages and file name types, with no easy way of extending these.

Imprecise Dependency Checks Although GBS has been designed to not require developers and end users to have the same build tool machinery installed (cf. the

horizontal division on Figure 2.1), end users still have too much responsibility. Dolstra et al. [66] have found that GBS's dependency checks yield too imprecise results, such that developers or end users have to resolve the missing or incorrect dependencies on libraries or tools. Especially the inability to verify if some of the dependencies are implicit, i.e. unspecified, causes trouble. Automatic package management, as described in Section 2.1.4.4, is clearly missing, so users need to manually locate and install missing dependencies. Often, this involves installing other software, possibly using GBS as well, such that dependency problems can also manifest there.

Evolution In time, different versions of GBS (and its macros) have not been able to remain backward compatible. This has caused a lot of grief among build engineers. Given the advanced features of GNU Make (Section 2.1.3.3), some prominent GBS people [234] recently have been arguing in favour of dropping GBS and doing everything with GNU Make instead [27].

2.2.4 Problems with Application of Build Systems in General

The previous sections briefly have highlighted various major and minor understandability and scalability problems in "make", GNU Make and GBS. This section extends the discussion to understandability problems encountered during application and maintenance of build systems in general and to an extremely common misconception about the usage of "make", which has consequences on scalability and understandability at the same time.

2.2.4.1 Understandability

This section considers three problems related to the understandability and maintenance of build systems.

Stakeholders In Figure 2.2, we have seen that GBS distinguishes two kinds of users, regular developers and end users. In general, multiple stakeholders interact with a build system, each with their own concerns and problems. Developers assess the correctness of their code and, if the build failed, try to find out the cause (e.g. missing dependencies). When adding new sources, they need to understand how to adapt the build. Maintainers, on the other hand, require full knowledge of the inner mechanics of a system [46], want to find dead code, to check recent changes [58], etc. Deployers and open source end users prepare and configure the environment (library dependencies, system variables, etc.) in order to compile and install the software, while Quality Assurance just wants to add and run unit,

[27] http://www.airs.com/blog/archives/95/

regression and integration tests as seamlessly as possible [172]. Researchers are interested in (un)plugging experimental tools to analyse the source code, but for this they need to grasp the development architecture. In fact, everyone (except for end users) interacting with the software system from the design phase on has to deal with some facet of the build system at some time. Hence, behind the complex, parametrised build scripts a lot of knowledge is hidden, not only useful for the stakeholders, but also for comprehension [111] and re-engineering purposes. Dedicated, flexible tool support is needed to allow easy and effective access to this valuable data.

Overhead of Build Systems Dubois et al. [71] estimate that in modern scientific programming researchers spend 10 to 30% of their time on the build system, especially to make the software compile and deal with varying platforms and library versions. A lot of time is wasted on seemingly simple problems, but the large, cryptic build systems of legacy systems lack sophisticated error feedback. As anecdotical evidence, on one and the same day[28] two emails were sent independently to the mailing list of a framework we are using (LLVM[29]), each complaining about a build system problem. The framework itself is very recent, but part of it is based on the GCC compiler[30]. One person reported[31] that for some reason `make -C dir all` and `cd dir; make all` each did separate things, while someone else literally complained[32]:

> llvm-gcc doesn't build under freebsd-current at the moment. I don't know much about the build system of GCC, so figuring out what went wrong took me quite some time.

This evidence clearly shows that understanding and maintenance of build systems is an important issue and requires adequate tool support.

Build System Changes Another insight into the complexities of maintaining build systems comes from Robles [195], who has investigated the role of various non-source code artifacts in open source software. He has studied usage patterns of the KDE desktop environment's source code repository (before its migration to CMake, see Section 2.3.3.1). He finds that changes to source code and corresponding build modifications are not committed at the same time into the version control system. Instead, changes to build files are bundled into one big change set. Many such change sets occur over the investigated period of repository activity. He attributes these observations to two things: (1) the build system needs to change very

[28] Tuesday, the 18th of December, 2007.
[29] `http://llvm.org/`
[30] `http://www.gcc.org/`
[31] `http://www.nabble.com/Broken-makefile-dependencies--to14392495.html`
[32] `http://www.nabble.com/FreeBSD-current-to14401495.html`

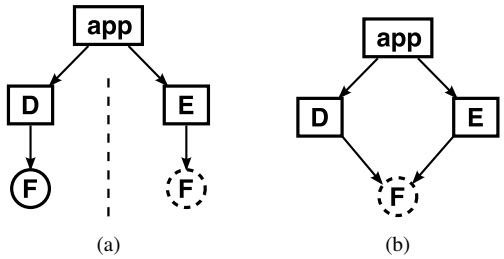


Figure 2.8: Conceptual difference between recursive (left) and non-recursive "make" (right). The former results in an inconsistent build, because "D" is not rebuilt after "E" has modified "F" during its construction.

frequently; (2) build scripts are tightly coupled, so most changes percolate through many of them at once. This is acknowledged by Keith Owens [186][33] who reports that a cleanup of one central makefile in the Linux build system between version 2.5.15 and 2.5.19 has caused 392 makefiles to be modified. Hence, grasping the impact of build changes is crucial for the maintenance of a build system.

2.2.4.2 Recursive versus Non-recursive "make"

The adoption of "make" has been plagued by a persistent misconception [171]. Although Feldman presented "make" as "most useful for medium-sized programming projects" developers soon have started to use it for ever bigger systems. To achieve this, they typically modularise their build per directory, i.e. they put makefiles in each subdirectory in their source code hierarchy and arrange that each makefile explicitly invokes a separate "make" subprocess for each subdirectory's makefile. As such, each subdirectory's build behaviour is managed locally. New code can easily be added by providing a new makefile and by making the parent directory's makefile invoke it appropriately. This scheme is called "recursive make" and is used pervasively as the basic structure of the build layer.

Figure 2.8a shows the negative implications of "recursive make" on the build system's underlying dependency DAG. A two-level directory structure with "recursive make" is assumed in this example. The key problem is that there is no communication between different "make" processes, as they do not share memory nor use a kind of common repository, except for a number of defined constants and a start target explicitly passed to the child process. As a consequence, the DAGs of sibling "make" processes like for directories D and E are disconnected, i.e. there are no interconnections. This is represented by the dashed line on Figure 2.8a. Although build scripts are cleanly modularised in separate directories, "recursive make" breaks Feldman's original DAG model by pruning (possibly) important links between subgraphs.

[33] `http://lwn.net/Articles/1500/`

This can lead to very subtle problems. In the top directory of Figure 2.8a, an application called “app” has to be built. For this, two “make” subprocesses will be needed, one for each subdirectory, as “app” relies on components “D” and “E”. Furthermore, suppose that both “D” and “E” rely on the same file, “F”, and that “D” will be built first. The first “make” subprocess will do its job and generate “D”. Then, a second subprocess will construct “E”. However, for some reason “F” is modified during the build of “E”, or in the extreme case only “F”’s time stamp is updated. In any case, “E” is constructed from the modified “F”, the subprocess terminates and the top process will link “D” and “E” into “app”. “app” obviously has a problem, as it ultimately relies on components which were generated from inconsistent versions of “F”. Because the dependency of “D” and “E” on the same file (“F”) is expressed in two separate, non-communicating “make” processes, the commonality of “F” cannot be detected by the parent process and hence not be dealt with.

This kind of problems have plagued “recursive make” for quite some time, hence there are a number of known, but unsatisfying workarounds for them. The easiest approach is to invoke “make” a couple of times, such that a more or less stable situation is obtained. As a special case, loops can be made inside the makefiles themselves to have more fine-grained control over the build subcomponent to stabilise. Of course, this drastically hampers incremental compilation efficiency and looping not always has effect, e.g. if “F” would be modified each time “E” is considered for (re)build. Instead, one could deduce a particular build order in which no inconsistency arises. In our example, this would mean to first build “E” and only then to consider “D”. In small builds, this strategy might be feasible, but this very soon gets unmanageable. A third workaround is to add extra dependencies to the various makefiles to make up for the missing interconnections. This of course requires thorough knowledge about the most crucial missing dependencies, and can also lead to maintenance nightmares. Every build re-engineering action potentially requires dependencies to be updated in various makefiles. None of these workarounds is satisfying, and they only increase the complexity of the build system.

There are other problems associated with “recursive make” as well, with less or even no possible solutions. Parallel builds are almost always broken. The missing interconnections between build subgraphs are actually crucial synchronisation beacons, i.e. they allow “make” to perform a rudimentary form of race condition detection. Fixing this is hard to do. A second inherent disadvantage is the possibly high cost of subprocess creation [167]. Third, the strategy used to recurse through the directory structure is spread across all makefiles. This is not only a matter of extra SLOC. It can lead to maintenance problems and can induce build problems because each subdirectory is responsible for resuming the build recursion in the right way. This can be beneficial, but in the case that certain special actions are

required, a traditional "recursive make" cannot enforce these. Modularisation of the build recursion logic is the answer, but this is not trivial to achieve (as we will see in 4.5.3.2).

The real solution to these problems is to drop "recursive make" altogether and to use a "non-recursive make" instead [171]. Basically, one "make" process is responsible for building everything, without delegating tasks to "make" subprocesses. As a result, this one process sees the whole build dependency graph, eliminating the problems mentioned above. An extra advantage is that the added dependencies allow better opportunities to shortcut/prune a build, i.e. for incremental compilation. Faster builds are the result of this. Figure 2.8b illustrates this on our example system. In this case, the common dependencies on "F" are known by "make" and it is possible to fix the build order problem by declaratively specifying the required dependencies. Of course, the naive approach of using only one physical makefile to describe the whole build is not feasible. Instead, people can exploit "make"'s `include` feature as discussed in Section 2.1.3.3. The build can then still be modularised in various makefile snippets, but instead of invoking them as independent subprocesses, they are textually combined at the start of the top "make" process to give the illusion of one big makefile at build-time. An added benefit is the ability to combine "recursive" and "non-recursive" build like this by conditionally choosing either the recursion or the `include` scheme.

However, there is no such thing as a free lunch. The fact that a build graph is constructed for the whole system, means that extra care is required when specifying build rules. "make" only has one namespace, so variables, macros, functions and targets should have a unique name. Similarly, the current directory in relation to which files are looked up and compilers put their generated code, is harder to find out. For "recursive make", this is almost always the makefile's enclosing directory, but for "non-recursive make" this always remains the outermost directory. All target names and commands should contain relative paths or a "make" variable can be used to keep track of the current directory. Another drawback is the overhead of generating the whole build dependency graph. Although it gives more confidence in the build, this can especially hit developers when they only need to rebuild a particular subdirectory of the system. Special care is needed to handle this. Some makefile generators like automake (see Section 2.1.3.2) explicitly provide support for non-recursive make[34].

Unless special care is taken to devise, document and enforce build system idioms like "recursive make", people are puzzled by this complex composition of the build scripts. The build parameters introduced by the configuration system aggravate things even further, as some design decisions may be tied to particular configurations only. Manual inspection of the build and configuration scripts, aided by lexical search tools like "grep", is not effective in these circumstances.

[34] `http://sourceware.org/automake/automake.html#Alternative/`

If a build system becomes too hard to understand and maintain, the risk of tailoring the source code structure to the build layout, just to make it compile and link, increases. The combination of all these problems warrants explicit tool support to manage build systems, especially if the phenomenon of co-evolution of source code and the build system presented in the next section is considered.

2.3 The Roots of Co-evolution

This section first examines existing conceptual and experimental indications of co-evolution of source code and the build system, and then postulates four "roots of co-evolution" as conceptual evidence of co-evolution. The existing indications comprise research in which source code and the build system interact in some way (Section 2.3.1), Favre's [85] taxonomy of co-evolution (Section 2.3.2) and early experimental evidence of co-evolution (Section 2.3.3). We distill the roots of co-evolution from research on programming languages and the build system. The roots correspond to specific links between source code and the build system which explain why co-evolution of source code and the build system occurs and they can serve as the starting point for techniques to deal with the co-evolution.

The key point that this section tries to make clear is that co-evolution is a fundamental characteristic of build systems, but that the understandability and scalability problems outlined in Section 2.2 make it almost impossible to firmly deal with co-evolution without the appropriate support by tools.

2.3.1 Co-evolution in Software Development

There are both indirect and direct indications of co-evolution of source code and build system. The indirect indications can be found in reverse- and re-engineering research. Explicit work on co-evolution is situated in software architecture, software design, product line research and software testing.

In the reverse-engineering community, Tu et al. [228] have proposed an additional view to Krüchten's "4+1" View model [139]. The "build time architectural view" documents the high-level architecture of build systems, i.e. the various steps involved across the whole build, even if it spans multiple invocations of "make". This technique enables them to find evidence of a kind of bootstrapping architectural build style within the build systems of GCC, Perl, etc. To some extent, Tu et al. have corrected the build system's omission from Krüchten's software development model and as such acknowledge the importance of the build.

Holt et al. [119] claim that the build process work flow is the ideal base for tool-supported comprehension of big software systems. This claim is based on the fact that developers encode various facets of program design into the build process, such as emulation of modules by header files, the order of linking (composing)

an application, etc. This knowledge has no equivalent in the source code, so the build system is the only reliable source of this data. Similar observations are made by Champaign et al. [44], in the context of the study of a source code package stability metric. They note that insight into the build system "was essential to getting an accurate view of dependencies between packages". In other words, to get a slightly higher-level view on source code organisation, the build system proves indispensable. They remark, however, that implicit dependencies generate noise in their metrics, as these correspond to unspecified relations.

In the re-engineering community, Fard et al [82] have looked at co-evolution from another angle. They describe how header file dependencies in C/C++ systems can be restructured in order to dramatically speed up builds. A reflexion model [177] is used to expose any divergences and absences w.r.t. a proposed source code architecture. All repair actions undertaken to converge concrete and conceptual architecture are used to control directory and header file restructuring. In effect, the build system is refactored by restructuring the source code. As a side-effect, the build and the software architecture become easier to understand, as they become better structured. This touches upon the observations in Section 2.2.1 and Section 2.2.2 about the integration problems of source code relations in the build system and suggests a close relation between source code architecture and the build system.

In the reverse-engineering community, there is also direct evidence of co-evolution phenomena. Many reverse-engineering techniques have been proposed to recover and document the "concrete" architecture of a system as modeled in its implementation, and to contrast it to the "conceptual" architecture originally designed for it. Most of these approaches are based on so-called fact extractors. In this field, data extracted from source code [173, 22, 176, 127, 29, 92], object files [111], etc. is stored as facts and relations between them. Using a query language, these facts can be filtered, reduced and composed (using human intervention) into a high-level architecture of a software system. Filtering and composition are repeatable in order to keep track of the recovered architecture during the system's evolution. These techniques explicitly target co-evolution of source code and architecture.

Recently, additional direct indications of the existence of co-evolution phenomena in software development have been given. Wuyts [243] has studied co-evolution of design and implementation of a software system. To keep both development phases synchronised, he proposes a logic meta-programming (LMP) approach in which a logic meta-language is integrated with a base programming language. Both languages can access each other and exchange data. The logic meta-layer is capable of encoding the program's design in various ways (design patterns, UML class diagrams, etc.) and of reacting on violations or other events in the base program. The work of Kellens [128] is closely related to this, but he

focuses on co-evolution of design regularities like programming conventions with the implementation. He uses "Intensional Views" to express the intent of design regularities, and these are evaluated on each source code change made by a developer. He or she gets immediate feedback on violations of the co-evolution. The work of van Deursen et al. [232] considers co-evolution of identified variability opportunities of a product line and the concrete variability supported by the implementation. Synchronisation between these two requires an iterative process to recover the implemented variability points, update the configuration specifications and enhance the implementation with new extension points.

Closer to the domain of build systems, Moonen et al. [172] have reported on the co-evolution of test suites and source code. They note that "when evolving a system, the tests often need to co-evolve", whereas at the same time "many software evolution operations cannot safely take place without adequate tests". This looks similar to the relation between source code and the build system. However, whereas source code is just plain text without a correctly functioning build system, a system with test failures can often be compiled and executed. Hence, developers are not automatically forced to make all tests pass before going on. Many software processes (Extreme Programming) try to artificially impose this restriction[35]. In order to make test-enabled software development manageable, Moonen et al. propose to refactor the base code driven by the desire to keep the tests intact, i.e. by not changing interfaces. However, they note that in their case studies, test code quality is not as complete and high quality as the source code. This is analogous to the many build problems which have never received the same level of attention as source code has.

The indications in this section show that co-evolution in software is not a new phenomenon. Moonen et al. [172] even hint at other co-evolution relations between source code and other software artifacts like specifications, constraints and documentation. The next section presents a taxonomy of co-evolution in general [85]. We use this taxonomy to situate the co-evolution of source code and the build system which is the topic of this dissertation.

2.3.2 A Taxonomy of Co-evolution in Software Development

Favre [85] has generalised the concept of co-evolution developed by the approaches described in the previous section. For this, he has proposed a taxonomy in the form of a four-dimensional space in which software artifacts can be positioned. The four dimensions consist of level of abstraction (instance, model, meta-model or meta-meta-model), the phases of the software development cycle (requirements, analysis, etc.), possible software representations (source code, XML, etc.) and time. Any software artifact can be assigned a unique co-ordinate in the space formed

[35] E.g. by integrating it within the development environment or promoting test-driven development.

by the first three dimensions, whereas the fourth dimension provides a mechanism to model subsequent versions of the artifact, i.e. its evolution. As an example, a source code implementation model corresponds to the source code model of a software system in the implementation phase. The build system is a complementary software representation in the implementation phase of which an instance (a running build system), model (the GBS example in Appendix A), meta-model (Figure 2.2) or even meta-meta-model (Figure 2.1) can be situated in Favre's taxonomy. This taxonomy is sufficiently general to model co-evolution.

Co-evolution can be interpreted in Favre's taxonomy as the evolution in time of changes with "vertical impact". If a change to a software artifact at one level of a given dimension (e.g. the build system representation) requires changes to an artifact on another level of this dimension (e.g. source code), this change is said to have a "vertical impact"[36]. Depending on whether the change moves from high- to low-level or the other way around, Favre talks about "downwards" or "upwards" evolution. Whichever direction the changes are made in, they have to be made to ensure consistency between different artifacts [85]. The evolution in time of software artifact changes with "vertical impact" between two levels of a given dimension, shows in fact how software artifacts corresponding to the two levels co-evolve. In our example, the evolution in time of changes with "vertical impact" between source code and build system (representation dimension), gives an overview of possible co-evolution between these two representations of software. It is important to note that Favre does not claim that co-evolution occurs between any two representations, levels of abstraction or software development phase. Rather, his taxonomy gives a reference framework to understand and situate co-evolution.

As Favre notes [85], changes with a "vertical impact", and hence co-evolution, should lead to corrective co-evolution actions. Otherwise, inconsistencies arise. If the GBS model of a build system would change, the example of Appendix A would become useless if it was not adapted to the new model. This phenomenon is similar to the relation between the build system and source code, because unbuildable source code is just plain text. Developers are forced to look after their build system, otherwise they cannot compile, link and compose their system, i.e. they would not be able to test and debug their changes or release the software product to their customers. Unfortunately, the corrective co-evolution actions are hampered by complex and hard to maintain build systems. Without tool support, the implications of source code on the build system and vice versa are not clear and hence no appropriate action can be undertaken. This nuance is not made by Favre [85], but seems obvious from other work on co-evolution [243, 232, 128, 172] and from what we have discussed about the build system in Section 2.2.

Favre's [85] taxonomy gives a general reference framework to situate co-evo-

[36] The notion of changes with "horizontal impact" does also exist, but is not relevant to this discussion.

lution in software development. In principal, co-evolution occurs when, spread over time, changes with "vertical impact" are made between two levels of a given dimension of the taxonomy. Corrective actions are required to enforce consistency between the two levels. This dissertation focuses on the source code and build system levels of the representation dimension to find conceptual and experimental evidence of co-evolution of source code and the build system. Its second goal is to propose dedicated tool support for enforcing consistency between the two representations.

The next section discusses early experimental evidence of co-evolution between source code and the build system.

2.3.3 Early Evidence for Co-evolution between Source Code and the Build System

The two cases described in this section independently give early experimental evidence of the existence of co-evolution between source code and the build system. The first case describes the migration of the open source KDE desktop environment[37] to another build system. The second case relates on the work of de Jonge [57] about constraints on source code reuse imposed by the build system.

2.3.3.1 KDE Migrates from GBS to CMake

KDE is a major open source desktop environment (started in 1996) based on the Qt rich client development framework[38]. This is a C++ framework which provides amongst others GUI widgets and an event system (signal/slot system). To overcome C++'s lack of reflection capabilities, Qt uses a special meta-object compiler ("moc") to process macros within header files before the actual compilation occurs in order to generate the meta-information required to implement the advanced Qt framework features. Developers have extended KDE's GBS build system (Section 2.1.3) in various ways to incorporate "moc" into their development environment. Many more GBS extensions have followed. The integration of the DCOP communication protocol[39] e.g. required automatic generation of many new types of files during the build process. Tools to process documentation, translations, UI configuration, etc. had to be integrated as well. To crown it all, the KDE source code currently consists of almost 6 MSLOC[40] and has to be extremely configurable to let Linux distribution providers customise it. As explained in Section 2.1.3.1, GBS is also responsible for managing this. Hence, KDE puts an enormous strain on its build system.

[37] `http://kde.org/`
[38] `http://trolltech.com/products/qt/`
[39] `http://developer.kde.org/documentation/other/dcop.html`
[40] Source Lines Of Code, i.e. the number of lines of code which do not represent comments or whitespace.

In 2005, developers unanimously have decided to move to a new build system[41]. This migration can be attributed completely to the rigid, inflexible build system KDE developers had obtained at that point in time. As noted by Troy Unrau, only a limited number of people (less than ten) had a good understanding of the build system[42], such that simple actions like moving files from one folder to another or starting a new subproject had become much too complex and heavyweight[43]. A first pilot project with SCons[44] failed because the SCons developers did not provide enough support. A second project, started in 2006, successfully migrated the KDE code base to the CMake[45] build system. CMake's biggest selling point is the cross-platform generation of the build layer, e.g. as makefiles or XCode project files, its speed and its compact, declarative configuration scripts. In the meantime, some other major open source systems like Scribus[46] have decided to move to CMake, for exactly the same reasons.

KDE's migration to another build system provides early experimental evidence of co-evolution of source code and the build system. The KDE people have decided that the flexibility of CMake (and corresponding freedom at the source code level) was worth the huge effort of reimplementing the build system from scratch, composing a new build development tool chain and educating all developers about CMake. They did not really have a choice, as there were already clear signs that source code development was suffering from build system strain. Given the large number of developers around the world (more than thousand), this could have led to a disaster. Hence, this case illustrates that a rigid, inflexible build system forms a serious threat for source code evolution.

2.3.3.2 Source Code Reuse Restricted by the Build System

de Jonge [57] has looked at source code reusability. Decades ago, this problem has almost[47] been solved at the language-level with packages and modules. In practice, many projects suffer to make components available for reuse, because each source code component should be distributed with its own piece of build system. However, most build systems do not have a clear one-to-one mapping of source code components to subsystems of the build process and even if there is one, the subsystem is typically coupled to other subsystems or requires special libraries to be installed. To boot, the source code component usually should be configurable too, but the configuration options affecting it are typically spread throughout the whole build system. Hence, the decomposition into components at the design level

[41] `http://dot.kde.org/1172083974/`
[42] `https://lwn.net/Articles/188693/`
[43] `http://dot.kde.org/1172083974/`
[44] `http://www.scons.org/`
[45] `http://www.cmake.org/`
[46] `http://www.scribus.net/`
[47] Assuming that the API is not too system-specific.

does not transfer to the build level.

To solve this, de Jonge introduces the notion of build-level components and a number of rules to which builds should adhere in order to obtain such components. Build-level components correspond to source code directories which contain build, configuration and requires interfaces. A build interface describes the build actions performed by a component's build process, whereas the configuration interface exposes all configuration choices affecting the component. The requires interface lists all the dependencies on other components. Given these, composing a system of user-defined components corresponds to configuring or invoking the right pieces of the interfaces. This is feasible because all configuration and build options are explicitly documented and enforced. To complement the build-level component interfaces and stimulate disciplined application of the components, de Jonge has devised a range of build-level development rules [57]. A re-engineering process has been defined to transform existing software into build-level components. This has been successfully applied on the well-known Graphviz[48] and Mozilla[49] applications.

The work of de Jonge gives a second experimental evidence of co-evolution of source code and the build system. Source code reuse, one of the biggest promises of software engineering [224], has been studied and remedied for years almost exclusively on the level of source code and design. However, unawareness of the crucial link with the build system results in conceptually decomposed systems which cannot be untangled at the build level.

2.3.4 Conceptual Relations between Source Code and the Build System

From the initial indications of co-evolution in software development (Section 2.3.1) and Favre's taxonomy (Section 2.3.2), we have learnt that co-evolution of source code and the build system conceptually can exist and what co-evolution exactly means. The experimental evidence from KDE and de Jonge (Section 2.3.3) has given us early proof that the phenomenon does exist in practice. However, we are not aware of more explicit investigation of co-evolution of source code and the build system, nor of conceptual explanations of the phenomenon, nor of tool support to deal with it. These three activities are the subject of this dissertation. To get a better understanding of co-evolution of source code and the build system, we first unravel its conceptual foundations. In later chapters, these will be used as the starting point for tool support and as yardsticks to detect co-evolution issues in case studies.

This section postulates and discusses four "roots of co-evolution". These rep-

[48] http://www.graphviz.org/
[49] http://www.mozilla.org/

resent four potential sources of co-evolution of source code and the build system. They have been derived from existing work on programming languages and build systems and correspond to bi-directional links between both worlds. These links are bi-directional in the sense that changes at the source code end of the link have repercussions on the other end and vice versa. Hence, each of these roots is a potential source of co-evolution of source code and the build system.

These are the four roots we discuss in this section:

RC1 Modular reasoning in programming languages and compilation units in build systems are strongly related.

RC2 Build dependencies are used to reify the architectural structure of the source code.

RC3 Architectural interfaces in the source code are not always respected by incremental build processes.

RC4 Build system configuration layers are used as a poor man's support for product line-styled variability of source code.

For ease of reference, we have assigned a name to each root. The order in which they are listed is not chosen at random. RC1 deals with basic units of composition (either in the source code or the build system), while RC2 describes how to compose these units. Composition has to satisfy certain constraints to be correct (RC3) and needs to be configured somehow (RC4). The first three roots correspond to build layer responsibilities (Section 2.1.2), while the fourth one corresponds to the configuration layer.

The next four sections discuss these four roots in detail. Afterwards, we consider a recent example of an invasive source code change, i.e. the introduction of AOSD. We argue that this triggers each of the four roots of co-evolution. Hence, our roots state that application of AOP technology requires many kinds of build system changes in order to keep source code and the build system consistent with each other. Experimental validation of this is performed in the second half of this dissertation (starting with Chapter 6).

2.3.4.1 RC1: Modular Reasoning vs. Unit of Compilation

The first root (RC1) states that modular reasoning in programming languages are strongly related to compilation units in build systems. This seems rather obvious nowadays. However, compilation units and source modules do not necessarily overlap conceptually [23]. Parnas [187] originally considered a module as an entity which hides a difficult design decision from others by only exposing a public interface (contract) to the outside without access to implementation details. As

such, different module implementations can be selected during the build or at runtime without requiring changes to other modules. Once interfaces are agreed upon, modules can be developed independently and developers can apply local reasoning to understand a module's implementation. This means that they only need to investigate the module's implementation and the imported interfaces. This definition of a module does not postulate how a module should be translated to build concepts.

We consider two ways in which modular reasoning at the source code level is related to compilation units in the build system. First, we discuss the granularity of modules and units. If a build system manipulates files (as GBS does), a module on the language level may either correspond to part of a file, a file or a collection of files, as long as it hides a specific design decision. On the other hand, given a module concept of a programming language, one could use a build system with finer-grained compilation units, similar ones or more coarse-grained units. Intuitively, a build system with units which are more coarse-grained than the module concept cannot exploit the programming language's potential modularity to its fullest, as a change of module will not be possible without changing other modules as well. Hence, although conceptually different, modular reasoning at the source code level is related to compilation units in the build system.

In practice, modules often correspond to at least one file. Historically, many people had to split their programs in different pieces because of limitations of hardware or compiler [23]. Programming languages like Fortran II [18], PL/I [193], Algol-60, Simula-67 [184], Algol-68 [151, 150], C [132], C++, Java, etc. have one-to-one mappings from their respective notion of module to compilation unit. Mesa [146] and Modula-2 [242] have more elaborate module provisions with explicit constructs to expose (export) and import interfaces. In Mesa, multiple implementation modules can contribute to the implementation of one interface, i.e. modules may span across multiple units of compilation. Nowadays, many IDEs progress in the opposite direction, i.e. they enable compilation units which are much more fine-grained than the language modules [126, 88, 199].

The first interpretation of RC1 (granularity) does point at a strong relation between modular reasoning at the source code level and build units, but there does not seem to be room for change. Once a particular programming language, build system and compiler have been chosen, both ends of the relation seem to be fixed. There is a second interpretation of RC1 which is a slight generalisation of the first one and shows that RC1 can be a real problem. This interpretation corresponds to the observations of de Jonge [57], i.e. the mismatch between source code reusability and the build system. This is essentially a generalisation of the first interpretation of RC1, as reusability in source code is achieved by a well thought-out decomposition into modules and coarser-grained components. A typical problem in the build system, as pointed out by de Jonge, is that contrary to modules, components

are not mapped transparently to build "components". Hence, the corresponding build units of the modules within a source component cannot be identified clearly and no clear build, configuration or requires interfaces [57] are specified in the build system for a particular component. Hence, there is a strong relation between higher-level modular reasoning at the source code level and build system units, and it is known to cause problems if the build system is not well-prepared.

This section has shown in two different ways how modular reasoning at the source code level is strongly related with build system units. One interpretation has compared module units at the source code level with the granularity of compilation units provided by build system and compilers. The second interpretation corresponds to de Jonge's observations [57] about the mismatch in reusability between the source code and build system. This explains the RC1 root of co-evolution. The next section discusses RC2, which looks at the composition of modules and units.

2.3.4.2 RC2: Programming-in-the-large vs. Build Dependencies

RC2 claims that build dependencies are used to reify the architectural structure of the source code. Composition of source code modules into a system is called "programming-in-the-large" [60], as opposed to "programming-in-the-small" which is concerned with manipulating data and implementing algorithms. The problem with programming-in-the-large is that, despite advances in e.g. Component-Based Software Development [224] (CBSD), dedicated language support or techniques are not used in legacy systems [84]. Instead, developers have (ab)used the build system to compose an application from source code modules. Build systems like GBS and older ones are not prepared for this task, which means that various workarounds and hacks are needed. Combined with the understandability and scalability problems of build systems (Section 2.2), this means that program composition is hidden somewhere in the build system. This clearly hampers evolution of the source code and the build system. This section elaborates on this phenomenon.

At the language level, modules are connected with each other via importing and exporting of interfaces [187]. A given composition is valid if a module is only accessed via its exported interface and if each imported interface is bound to a concrete module implementation. Languages like C, C++ or Java support this model. They allow to specify the interfaces or implementations required by modules or classes, but they are not able to specify the actual composition. If there are multiple Java classes implementing an interface and no concrete implementation is asked for in the source code, in general the build system has to select a particular implementation based on the user's configuration. The key point is that these programming languages do not provide explicit means to specify concrete

program composition[50]. They only provide programming-in-the-small, whereas module composition corresponds to programming-in-the-large.

DeRemer et al. have coined the term "programming-in-the-large" [60]. They have proposed the concept of a dedicated module interconnection language (MIL) to hierarchically compose modules into a full software system, i.e. to program with modules. The crucial difference with programming-in-the-small according to DeRemer et al. is that composition in the latter generates tree-like structures, whereas an MIL is capable of much more complex, graph-like compositions. The nodes of the graph can be associated with fragments of a module. The edges can have structural meaning like "parent-of" relations, or restrict access to nodes. Nodes without attached fragments only provide structure. Mesa [146] is the prototypical example of a language which combines programming-in-the-small and programming-in-the-large. It features a powerful module system (with primitive versioning) in which multiple instances of a module can exist at the same time and the module itself manages shared state. Mesa has a complex, built-in MIL named C/Mesa, to hierarchically compose modules into "configurations". Configurations can be nested. Multiple implementation modules together can contribute to the implementation of one common interface. INTERCOL [227] and PROTEL [42] are other examples of MILs which even provide means for checking consistency of compositions across module versions.

Despite techniques like MILs, Favre [84] observes that at the end of the 1990s developers still reason in terms of technologies of the 1970s like programming-in-the-small, files, makefiles and preprocessor constructs instead of state-of-the-art technology like (modern incarnations of) MILs, models, configuration management, etc. Program structure is modeled as directory containment and composition of libraries and executables. In other words, the "make" DAG is abused as an MIL, something it was never meant for and is not capable of, as the "recursive make" discussion in Section 2.2.4.2 shows. Only if one would restrict DeRemer et al.'s graph nodes to be complete modules instead of module fragments, a "make"-based build system could be considered as a build system-level equivalent of an MIL[51]. This leads to problems. Favre [84] gives an example of a system of which the implementation can only be understood if its makefile of one thousand lines, all preprocessor code and the C code itself are understood. Specifying program composition in the build system implicitly makes the build system a part of the source code, but the understandability and scalability problems of build systems (Section 2.2) make the separation between programming-in-the-small and programming-in-the-large a curse rather than a blessing. Keeping source code and the build system consistent is a major task.

To summarise, RC2 claims that build dependencies are used to reify the archi-

[50] Note that higher-level CBSD techniques like Enterprise Java Beans [198] and others do provide declarative means to compose systems.

[51] Some of the later build tools provide better support for programming-in-the-large [205, 23].

		type checking		
		monolithic	separate	independent
compilation	monolithic	Pascal		
	separate	Fortran	Simula 67 (bottom-up)	
		PL/I libraries	Algol 68	
	independent	Fortran	PLISS	Mesa
		PL/I	type checking linkers	CLU-library
		C/C++ [106]		

Table 2.1: Tichy's overview of inter-compilation type-checking approaches [227]. Instead of the term "separate", Tichy originally used the term "incremental". However, the latter term nowadays [242] is used for any mechanism used to shorten compilation time by only recompiling those parts which have changed. Hence, we swap the terms "incremental" and "separate" in Tichy's overview.

tectural structure of the source code. This refers to the fact that module composition has traditionally been specified in the build system instead of via dedicated source code support for programming-in-the-large [84]. Hence, a complex relation exists between the source code architecture and the build system. Note that this does not contradict de Jonge's observations [57], as those were concerned with the mapping of modules and components to the build system, whereas RC2 is concerned with dependencies between those components. The next section discusses a consequence of the usage of build systems as MILs: possibly unsafe module composition.

2.3.4.3 RC3: Interface Consistency vs. Incremental Compilation

The third root of co-evolution (RC3) is also concerned with module composition, and considers an important side-effect of RC2, i.e. architectural interfaces in the source code are not always respected by incremental build processes. As we have seen in the previous section, composition of modules requires that a module is only accessed via its exported interface and that each imported interface is bound to a concrete module implementation. There is actually an extra requirement [227, 42]: all modules of which an imported interface is bound to a module M should see the same version of M. Otherwise, strange side-effects may occur because of inconsistent state or behaviour. Unfortunately, build systems do not necessarily enforce this extra module composition requirement because they typically do not natively support versioning (see Section 2.1.4.3), and especially because of incremental compilation and environment processing. We discuss this relation between source code and the build system in this section.

As Favre [84] has claimed, a build dependency graph can be interpreted as

a coarse-grained MIL. However, a build system does not just blindly process a build DAG to construct a system, as this would cause too much redundant work. Because the DAG specifies the dependencies between compilation units and the connection between compilation units and modules (RC1 of Section 2.3.4.1), the build system knows enough to deduce which compilation units should absolutely be recompiled, which ones possibly and which ones certainly not. Incremental compilation [41] (also: selective recompilation [9]) is the collective name of techniques aimed at exploiting this knowledge to reduce compilation time. Many languages have a compiler with a built-in incremental compilation approach, but some extra-compiler incremental compilation techniques have been proposed as well [9] and implemented in language-independent build tools like "make" which operate at the file level. Incremental compilation techniques are based on assumptions and heuristics of the extent of source code changes, and hence can be characterised by the risks they take to improve build speed at the expense of type safety (i.e. consistency of interfaces). Environment processing is a complementary technique to incremental compilation which is concerned with reducing the amount of environment (i.e. imported interface declarations) to process during compilation, as these may accumulate quickly because of transitive module dependencies. Hence, a build system starts with a complete build DAG, but prunes it based on the incremental compilation and environment techniques in use. We now consider these techniques and their consequences on RC3.

Tichy [227] has been the first to categorise incremental[52] compilation techniques based on a distinction between on the one hand constraints on the incremental compilation and on the other hand the link-time type-checking (i.e. consistency of interfaces) approach which is used. Linking is the basic operation performed by the build system to combine modules together into an application or library. The constraints on incremental compilation refer to the fact whether incremental compilation is not possible at all ("monolithic"), is possible only if modules are combined following a partial order ("separate"), or is possible in any order ("independent"). The other dimension, inter-compilation type checking, ranges between no type-checking during linking ("monolithic"), type-checking of interfaces if files are processed in a partial order ("separate") and type-checking regardless of processing order ("independent"). Tichy also identified a third dimension, i.e. consistency of module versions. Tichy's INTERCOL [227] and Cashin et al.'s PROTEL [42] are MILs which are able to take the latter dimension into account.

Table 2.1 shows Tichy's categorisation. At one end of the spectrum, Pascal does not provide means for incremental compilation, whereas Mesa e.g. allows incremental compilation in any order with fully checked interfaces in the result-

[52] Tichy's original paper [227] used the term "separate compilation" for the collection of all approaches used to shorten compilation time by only recompiling those parts which have changed, and "incremental compilation" as a specific case of this. However, later both terms have been swapped in literature (see e.g. Wirth [242]), so we do the same in this dissertation.

ing system. Languages in between like Simula-67 have to be compiled and type-checked depth-first according to dependencies on other modules. C [132] implementation modules can be compiled in any order and the linker will check whether or not the declared variables and procedures used by implementation modules match the types of actual definitions within the object files, but this type checking is rather liberal (hence: monolithic). Note that some languages like Modula-2 [242, 106] do all type-checking at compile-time by first compiling all interfaces and then using the resulting type information to compile each implementation module. This categorisation clearly shows how type safety (interface consistency) and incremental compilation are related to each other.

The seminal work of incremental compilation techniques has been made by Adams[53] et al. [9]. They compare various known approaches on one medium-sized case study. In measuring the efficiency of the techniques, Adams et al. have looked at speedup of the build. The investigated strategies range from file-level techniques like "trickle-down" recompilation (cf. "make", Mesa [146] and Modula-2 [242]), over "smart" recompilation (only recompiles dependents which explicitly use changed declarations, as e.g. in the Eclipse compiler [212]) to aggressive interface inferring techniques like "smartest" recompilation. Faster techniques progressively become more dangerous for interface consistency, as they take more risks by letting some groups of modules use inconsistent versions of modules or by not using any human interfaces at all (cf. "smartest" recompilation). If linking can be done incrementally, the above techniques could be used for them too, although more specialised techniques exist for this [192, 215, 126]. Many more recompilation techniques have been invented [88, 183, 87, 43, 201, 126], but for all of them the remarks above hold.

Adams et al. further point out that selective recompilation has other harmful consequences besides breaking interface consistency. First, they might reduce the opportunities for parallel builds, although it does not preclude it. Second, inter-procedural analysis cannot be combined as is with incremental compilation approaches like "smart" recompilation, as its whole-program knowledge cannot easily be updated incrementally. Approximation of the analysis results circumvents this, but it requires extra work to obtain sufficiently precise results [52]. Third, incremental compilation suffers from the so-called "big inhale" phenomenon. Adams et al. have measured that on average for each compilation unit 1.9 times as much environment data is passed to a compiler (signatures and declarations of other modules), whereas only 20% of this is actually used. This "big inhale" can consume up to 50% of compilation time [9], especially for low-level modules which have many (transitive) dependencies [120]. As changes to interfaces are automatically detected by incremental compilation approaches, developers are not afraid of them anymore [42, 9] and hence make the "big inhale" even worse. Limiting the

[53] Rolf Adams is not related to the author of this dissertation.

propagation of low-level interface changes is a possibility to deal with this [120], but to really benefit from recompilation speedups, the "big inhale" phenomenon itself should be tackled.

Gutknecht [106, 9] has classified the existing approaches for environment processing. Environment data can either be loaded from a symbol file in source form (class α) or in a compressed format (class β). The latter is more efficient if the units change less often than they are read. A symbol file can contain data of only one module with unresolved references to other modules (class A) or flatten all dependency information transitively reached from a module (class B). Loading is faster in the latter case, but more space is required (lots of duplication). C compilers belong to the Aα category, while Ada and Modula compilers are most of the time Aβ, except for Gutknecht's own approach (Bβ) [106]. By ignoring symbol files which are not used at all ("environment pruning") or only the symbols which are not referenced ("selective imbedding") [42] up to 80% of the environment data can be filtered out [9]. Once again, finer-grained techniques obtain better results in reducing incremental compilation overhead, but take certain risks to accomplish their task.

This section has shown how incremental compilation and environment processing are build system techniques to speed up compilation, but which take risks in doing so. Hence, dependencies between source code modules may be ignored on program changes, such that the resulting composition does not respect the consistency requirement of module composition [227, 42]. Changes of incremental compilation technique or program interface modifications may lead to inconsistency problems. This explains the third root of co-evolution. The next section discusses the last root.

2.3.4.4 RC4: Program Variability vs. Build Unit Configuration

The fourth and final root of co-evolution between source code and the build system states that build system configuration layers are used as a poor man's support for product line-styled variability of source code. It is concerned with the mechanisms to statically[54] manage sources of variability in a software system:

- platform-dependent code
- selection of features
- selection of module alternatives
- consistency of versions

As a typical example, we have seen that GBS (Section 2.1.3) handles variability via a tight integration of source code and the build system. To abstract

[54] As discussed in Section 2.1.4.6, variation points may also exist long after build-time [64].

over platform-dependent code and feature selection in the source code, autoconf's build-time parameters can be used, but developers especially apply the C preprocessor. The latter is a simple but powerful tool which has lived in a symbiotic relation with the C compiler from the beginning to resolve many shortcomings of the C language and even of C++ [169]. As the C preprocessor is a text processing facility, preprocessor code ignores C's syntax rules. Because in addition normal and preprocessor code can be freely mixed, understandability of source code decreases significantly.

The C preprocessor offers three main constructs:

`#define/#undef` macro or constant (un)definition

`#include` file inclusion traditionally used to emulate a module/interface system

`#ifdef` conditional compilation which parametrises source code based on the build system configuration

Preprocessor constants are used in GBS's "config.h.in" (Section 2.1.3.1) to pass configuration decisions from the build system (via compiler flags) and also to establish hardcoded configuration choices in the source code. File inclusion on its own is not frequently used to manage variability, unless developers physically alter the included file via the file system. Conditional compilation on the other hand is the bread and butter of configuring legacy systems, not just in C environments. Based on a logical formula expressed in terms of preprocessor constants, certain code parts can be retained for compilation and others filtered out as if they were comments. Because the preprocessor is just a lexical tool, any incomplete piece of C code can be conditionally guarded. It is the most popular way to deal with platform-dependent code and user-configurable features. Ernst et al. [80] have measured that 37% of source code is controlled by conditional compilation logic. It is especially used for dealing with portability of an application (37% of use cases). On average, 8.4% of all lines of source code contains preprocessor directives. Conditional compilation takes up 48% of these, macro definitions 32% and file inclusion 15%. Hence, a considerable part of the source code is coupled directly with the build system for the purpose of configuration.

The third important source of variability, selection of module alternatives, cannot be treated at the source code level. As explained for RC2 (Section 2.3.4.2), it is the build system's responsibility to select the appropriate modules for a particular system. Based on user-specified configuration choices, certain source files can be excluded from compilation or not. Incidentally, build script constructs equivalent to the three C preprocessor operations above are typically used in the build layer behind the scenes to implement this, and also to guard build scripts against platform-dependent build logic. In GBS e.g., the conditional build logic eventually changes automake or GNU Make variables.

The fourth source of variation is versioning. As discussed for RC3 and in Section 2.1.4.3, each module can have multiple revisions between which dependencies may exist. Integration between build system and source control systems [227] deals with this, just as compilers which support versioning of compiled objects [146, 242, 42, 87, 201].

To summarise, there is a very strong link between source code and the build system on the level of configuration. Source code is typically heavily parametrised by configuration logic which can be controlled via preprocessor flags passed from the build system. As explained for RC2, module selection is completely controlled by the build system. These two observations make it clear that the evolution of source code configuration logic is closely related to evolution of the build system, and vice versa. This concludes the explanation of RC4, i.e. the last root of co-evolution.

2.3.5 Summary

This section has discussed four roots of co-evolution which represent a conceptual rationale for the existence of co-evolution between source code and the build system. RC1 relates modular reasoning in programming languages to compilation units in build systems based on the mapping of modules to build units and de Jonge's [57] observations about source code reusability and the build system. RC2 is concerned with composition of modules, which traditionally has been delegated to the build system [84]. Unfortunately, this makes program understanding and evolution more difficult, as build systems have understandability and scalability problems (Section 2.2). RC3 considers an important consequence of RC2, i.e. the problems induced by incremental compilation [227, 9] and environment processing [9, 106] on the consistency of module composition. RC4 claims that build system configuration layers are used as a poor man's support for product line-styled variability of source code. We have illustrated this by means of the GBS system described in Section 2.1.3. Source code is heavily parametrised to enable configuration control from the build system. The build system configures selection of source code modules for composition. Each of the four roots corresponds to one conceptual symptom of co-evolution between source code and the build system.

The next two chapters propose tool support to understand co-evolution of source code and the build system and apply it on a case study to validate the four roots in practice, i.e. to check whether they really are capable of explaining symptoms of co-evolution of source code and the build system. Once the roots have been validated, we can use them to predict source code or build system evolution problems. The next section discusses how the introduction of AOSD triggers changes in the source code for each of the four roots of co-evolution. Hence, our (vali-

dated) roots state that application of AOP technology requires many kinds of build system changes in order to keep source code and the build system consistent with each other. Experimental validation of these predictions is performed in the second half of this dissertation (starting with Chapter 6).

2.4 The Relation between AOP and the Roots of Co-evolution

In the previous section, we have derived four roots of co-evolution based on indications of co-evolution of source code and the build system, early experimental evidence and existing research on programming languages and build systems. These roots represent four ways in which source code changes induce changes on the build system and vice versa. Source code changes can result from small program modifications, changes to the programming language (model) or even switching from programming paradigm (meta-model). Recently, the latter kind of changes has shown to be a reality. Similar to what we have seen for the various recompilation strategies and environment processing approaches compared by Adams et al. [9], ever more complex, finer-grained composition techniques at the language-level are being put forward to enable a more flexible design of a software system. They propose new kinds of modules, other ways to structure a system, more advanced type systems and better abstractions for configurability. Whereas many ideas do not make it in practice, some of them do and present new challenges to co-evolving software participants like the build system. This dissertation focuses on one of these technologies, i.e. aspect oriented software development (AOSD).

Contrary to other technologies, AOSD has received a lot of attention from major industry players like IBM [200]. A lot of care has been taken to develop an industry-level aspect implementation for Java, AspectJ [133], which at the same time has become a yardstick for many aspect language researchers. In other words, AOSD makes a big chance to survive. Most of the software in existence today is legacy code however, so AOSD technology should be adapted to these environments to catalyse industrial adoption [142, 202, 170]. Before this can happen, the consequences and risks of introducing AOSD in a legacy system should be explored. The four roots of co-evolution established in Section 2.3 and validated in later chapters, suggest that one of these risks is the fact that existing build system technology is not able to keep pace with the source code changes [142, 202] introduced by AOP. Source code and the build system become inconsistent if co-evolution between source code and the build system cannot be managed. This is harmful for the adoption of AOSD.

Case studies which focus on the effects of co-evolution of source code and the build system on the introduction of AOSD in legacy systems are presented in

later chapters. This section discusses the conceptual impact of AOP on the source code level, for each of the four roots of co-evolution. Assuming that these roots are valid, this predicts changes in the build system. First however, Section 2.4.1 describes the ideas of AOSD at the source code level.

2.4.1 AOP

Aspect oriented programming (AOP) is targeted at modularising so-called "cross-cutting concerns" (CCCs) [134]. When implemented using traditional programming language techniques, these concerns cause two undesired phenomena to crop up in the source code: scattering and tangling. The former corresponds to implementation fragments of a concern (like e.g. caching) which occur at many places throughout the source code. Changes to the concern's implementation likely amounts to making changes at many places in the system, which is tedious, error-prone and hampers understandability. The situation is even worse, because at each location where a caching concern fragment occurs, it can be tangled (mixed) with fragments of other concerns. This means that programmers need to understand the interplay between multiple concerns before being able to modify the caching concern. AOP deals with these undesirable program properties by extracting cross-cutting concerns in a new kind of modules: aspects.

Initial aspect languages were domain-specific, geared e.g. towards synchronisation (COOL) or remote data transfer (RIDL) [161]. Very soon, a movement towards a general-purpose aspect language appeared, culminating into AspectJ [133]. To date, this is still the primary aspect language in existence, both in research and in practice. AspectJ is an aspect language for Java which introduces the concepts of advice, pointcut, join points, etc. An aspect is similar to a module as we have considered in this chapter. The main difference with e.g. C translation units or Java classes is the presence of "advice", which consists of a "pointcut"[55] and an "advice body". According to the most common school ("asymmetric AOP"), the implementation of crosscutting concerns is extracted from the "base code". The latter corresponds to the implementation of the main concerns, the so-called "dominant decomposition" which forms the backbone of the whole system. CCC implementation fragments are separated from the base code and localised into (possibly) multiple advice bodies of an aspect.

Commonly, a distinction is made between "homogeneous" and "heterogeneous" CCCs [51]. Homogeneous concerns are said to look almost identical everywhere they occur. On the other hand, heterogeneous concerns may vary widely between different occurrences. As a consequence, the implementation of homogeneous concerns may easily be localised into one advice body, whereas heterogeneous concerns are harder to implement in a robust way. In the latter case, the advan-

[55] Sometimes abbreviated to "PCD", for "pointcut designator".

tages of AOP may seem to be limited, but this actually depends on the expressivity of the aspect language, i.e. the advice and pointcut language. The better variability can be expressed in the aspect language, the easier heterogeneous advice can be extracted into advice.

Code separation is only one part of the effort required to resolve scattering and tangling. One still needs to specify at which moments during the base program execution an advice body should be invoked. Instead of embedding explicit calls to advice within the base code, an advice is invoked automatically once a condition (pointcut) is satisfied. This inversion of dependencies [182] forms the core idea behind AOP. The moments in time when advice can be triggered are called "join points", as this is where the main concern(s) and a CCC join each other. Established kinds of join points are method calls and executions, variable access and manipulation, etc. A pointcut can make use of program structure, name patterns, dynamic program state, etc. to describe the intended set of join points. The process of matching join points with a pointcut and executing advice on a pointcut match is called "weaving". Conceptually, a "weaver" monitors the program execution and checks each join point to decide whether there is a match or not. In practice, the set of interesting join points can be reduced based on analysis of the pointcuts, or weaving can be moved completely to the compiler, with only a couple of dynamic checks ("residues") left at run-time. The only restriction is that the developer should always have the perception of a run-time monitor.

Some aspect language like AspectJ also provide means for managing static crosscutting concerns, i.e. inter-type declarations[56] (ITD) [133]. Whereas advice enables means to alter program behaviour, ITD alters types or may facilitate program verification and error handling. The latter two applications solicit compiler feedback if a user-specified pointcut matches during weaving. Type alteration allows classes and interfaces to be extended with new attributes or methods and may even change the inheritance hierarchy by adding new interfaces to be implemented or a superclass. The idea is that these structural modifications support behavioural CCCs implemented as advice, but that they also allow base code developers to explicitly use the introduced attributes or methods. Griswold et al. call this "language-level obliviousness" [105], i.e. developers are aware of the woven aspects. If developers do not know anything about the possible aspects, one speaks of "designer obliviousness", unless developers may prepare the base code to expose better join points ("feature obliviousness").

From the asymmetric AOP perspective, aspects are a kind of parasite on top of the base code. They have more powerful constructs to compose concern fragments, i.e. advice and ITD, than base modules have. At the other extreme, "symmetric AOP" considers all modules to be equal in power (no base code) and to be capable of composing with each other using advanced mechanisms. For developers,

[56] The original name for this feature was "introduction".

this corresponds to "pure" obliviousness [105]. The hallmark example of symmetric AOP is Multi-Dimensional Separation of Concerns [226], but several attempts to implement this have failed thus far. Asymmetric AOP is the dominant AOP approach.

AOP has especially been studied in the context of OO systems, as a means to overcome the problems of scattering and tangling in even the most advanced OO languages. Nevertheless, CCCs are more fundamental than this. Anytime a problem is tackled by making some structural design decisions, remaining concerns have to fit into this main decomposition somehow. This problem is named the "tyranny of the dominant decomposition" [226]. Hence, CCCs not only occur in OO systems, but also in procedural or functional programs [134, 142, 202], as these also start from a main decomposition of the system. Keeping in mind that OO languages offer more powerful composition constructs than modular or procedural programming, this means that the latter have even less means to manage CCCs. Research has shown [36, 51, 33, 34, 35, 32] that CCCs represent an important evolution problem in legacy systems, especially if one takes the scale of these systems into account (millions of lines of code). Tangling and scattering of CCCs with the main concern heavily impact program understandability, while scattering increases the cost of maintenance and reduces traceability of code fragments to the modeled concern. Various researchers have considered AOP as a viable solution to deal with this problems in legacy systems [202, 179, 32].

This dissertation considers the impact on the build system of integration of AOP technology into the development process of a legacy system. The aim of this is to uncover the hidden cost of evolving the build system on source code changes, i.e. the practical consequences of co-evolution of source code and the build system. The remainder of this section shows how introduction of AOP technology indeed classifies as invasive source code changes according to the four roots of co-evolution (Section 2.3.4). Later chapters validate this conceptual evidence of co-evolution of source code and the build system with a number of representative case studies.

2.4.2 RC1: Modular Reasoning vs. Unit of Compilation

AOP introduces a significantly different kind of modular reasoning compared to previous paradigms [220, 219, 162, 135], i.e. whole-program reasoning. To better understand the virtues and nature of this kind of modular reasoning supported by AOP, Sullivan et al. [219] and Lopes et al. [162] independently have tried to formalise traditional and AOP modularity using design structure matrices (DSMs) and real options theory [19]. This section uses these techniques to highlight the shift in modular reasoning by AOP to whole-program reasoning, whereas later sections use them to highlight other characteristics of AOP. Kiczales et al. [135]

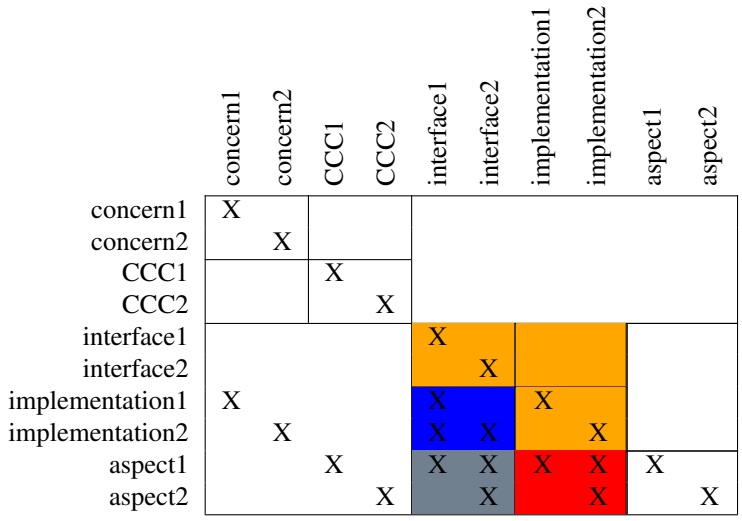


Figure 2.9: Example EDSM of an AspectJ-based system. The first four rows and columns consider concerns as environment parameters, the fifth and sixth row and column represent design rules (interfaces), and the remaining four rows and columns are design parameters. The orange four-by-four matrix in the middle is a basic DSM.

provide a different theory for whole-program reasoning. Based on RC1, this shift in modular reasoning has important consequences on compilation units in the build system.

We first explain the concepts of DSMs and real options theory. A basic DSM looks like the four-by-four orange matrix in the middle Figure 2.9. Its rows and columns consist of the same list of "design rules" and "design parameters", both of which correspond to design choices [187]. A mark in row R and column C means that design rule/decision R depends on C (or C influences R) according to the designers. In Figure 2.9, rows and columns have been partitioned and permutated to form square clusters ("proto-modules") of inter-dependent design decisions along the matrix's main diagonal. The ideal DSM is empty below the main diagonal, i.e. there is no coupling between proto-modules. This is very hard to obtain in practice, but one can reduce coupling as much as possible by using design rules. These express global rules which must be obeyed by the design parameters, e.g. that an implementation module can only be accessed via a specific interface. This shifts the coupling between implementation modules to the interface level, decoupling the implementation modules from each other. Modules which only depend on design rules, are named "hidden" because they can freely be replaced without any impact on other modules. As an example, in the orange matrix inside Figure 2.9, two design rules have been added to enforce access to the two modules (`implementation1` and `implementation2`) via

two interfaces (`interface1` and `interface2`). The blue region marks the dependencies from the two implementation (proto-)modules to the interfaces. In this case, the second module depends on the two interfaces. If we only consider the orange matrix, both implementation modules are hidden as they only depend on design rules. Hence, DSMs facilitate understanding of interaction between modules.

Baldwin et al.'s [19] "net options value" (NOV) theory allows to quantify the value of a modular design [220]. The NOV is a numerical measure for the number of options provided by a modular decomposition, i.e. the degree to which designers can freely experiment with alternative versions of modules before the costs involved with experimenting outweigh the potential benefits of the new modules. By measuring the relative NOV of two modular designs w.r.t. the same non-modular implementation, one can compare the two modular designs with each other. As design rules reduce coupling and foster hidden modules, they boost the NOV to higher values. Hence, a DSM gives a visual indication of the achievable NOV. In the orange matrix of Figure 2.9, the two design rules enable the two implementation modules to be easily swapped instead of just one of them (`implementation2`). We now apply DSMs and the NOV for understanding the influence of AOP on modular reasoning. Later on, they are used to illustrate inversion of dependencies [182].

The above formalism provides a first explanation of whole-program reasoning. Sullivan et al. [219] and Lopes et al. [162] have used the concept of "enhanced DSMs" to model AOP. These are DSMs to which "environment parameters" have been added as an extra kind of design rule [220] to model the incentives of changes to design parameters, i.e. the external constraints on the NOV of the system. Separation of concerns can be translated into environment parameters of a design, and aspects into extra design parameters (just like normal implementation modules). Figure 2.9 in full displays the EDSM of an AspectJ system which consists of two main concerns, two CCCs, two interfaces (design rules), two base modules and two aspects. We observe that the aspects depend on implementation modules and interfaces (design rules), i.e. they are not hidden. This corresponds to AspectJ's fine-grained and (conceptually) undisciplined aspect composition, in the sense that developers are allowed to write pointcuts in terms of implementation details like accidental variable accesses or internal method calls. From the perspective of NOV theory, this system has a serious flaw, which is highlighted in the colored regions in the lower two rows of Figure 2.9. As aspects depend on interfaces and implementation modules, and implementation modules on interfaces, this means that interfaces have to be designed and developed first, followed by the implementation modules and eventually the aspects. No parallel development of aspects and modules is possible (aspects are not hidden!), which decreases the potential NOV the system theoretically could achieve by extracting CCCs in separate

aspect modules. Without restrictions, aspects depend on whole-program reasoning, i.e. they require knowledge of the interfaces and modules which are part of the concrete system configuration. In other words, AOP is able to modularise CCCs in the classic sense of the word, but the fine-grained composition is not necessarily beneficial.

To overcome this, Baldwin et al.'s [19] theory suggests to add extra design rules, such that the gray and red areas of Figure 2.9 become empty, at the expense of extra rows/columns between `CCC2` and `interface1`. In the case of AOP this boils down to imposing a kind of abstract pointcut interface between base code and aspects. If aspects only use these abstract pointcuts, they are sure that base code changes will not change the semantics of the pointcut. Base code developers on the other hand have to ensure that their concrete specification of the abstract pointcuts is updated on base code changes. Griswold et al. [105] have proposed a crosscut programming interface (XPI) to decouple aspects from the base code. Other approaches achieve the same effect [11, 129]. Hence, without precautions, AOP intrinsically requires whole-program reasoning.

A second view on modularity with AOP is given by Kiczales et al. [135]. They argue that modular reasoning in the presence of crosscutting concerns still makes sense, but only if the classic notion of interface is extended to so-called "aspect-aware interfaces" (AAI). These extensions annotate method declarations with the name of the aspects, advice signatures and pointcuts which advise them. The interface of an aspect is enhanced with the enumeration of all join points matched by each advice. Note that all this information can be incrementally updated on changes to the source code. To understand a module in an AOP system, one should only look at a module's implementation, (extended) interface and interfaces of referenced modules. For CCCs, this is clearly better than the global reasoning it would take to understand them if they had not been modularised into aspects. Still, AAIs rely on global configuration information [105], i.e. they have to be computed based on the set of join points which are actually matched in a particular configuration of the system. Modular reasoning in the presence of aspects requires whole-program reasoning.

These two theories independently show how aspects require whole-program reasoning at the source code level. If the build system's notion of build unit does not change, RC1 suggests that inconsistency between source code and the build system arises. Hence, co-evolution of source code and the build system in the presence of AOP can only be managed if the aspect language restricts aspect composition somehow (by imposing design rules like XPIs [105]), or if build system technology is adapted to cope with finer-grained composition. However, there are three more ways in which AOP and build system interact. The next section discusses AOP's impact on RC2.

2.4.3 RC2: Programming-in-the-large vs. Build Dependencies

RC2 claims that build dependencies are used to reify the architectural structure of the source code. AOP introduces a different way of structuring source code, which has been referred to as "inversion of dependencies" [182]. This section gives three explanations of this kind of composition, based on EDSMs (see Section 2.4.2), comparison to OO composition [160] and connectors in engineering [182]. Inversion of dependencies by itself does not seem to be a problem to express in the build system. However, the combination with the implicit specification of advised join points in pointcuts and the ensuing fine-grained composition (see the previous section) potentially causes problems for co-evolution of source code and the build system. Design rules like XPIs may be able to relax these problems.

A first explanation of inversion of dependencies can be given using EDSMs [162]. Aspects modularise CCCs, such that tangling and scattering disappear. This can be derived from an EDSM. There is no tangling on Figure 2.9 because none of the proto-modules has a row with marks in columns which correspond to concerns implemented by other proto-modules, i.e. no module depends on concerns implemented by other proto-modules. There is no scattering either as there is no concern column with marks across multiple proto-modules' rows, i.e. no concern is depended on by more than one proto-module. As seen in the previous section, the tangling and scattering dependencies have been transformed into dependencies of aspects on interfaces and modules. In other words, explicit references between two or more components (proto-modules) have been replaced with implicit references from (external) aspects to the previously connected components, i.e. dependencies have been inverted. This corresponds to the intuitive notion of implicit invocation of advice if a given expression (pointcut) matches, instead of to explicit method calls. Hence, EDSMs clearly show how this composition works.

A second account of inversion of dependencies is given by Lohmann et al. [160], who have directly compared the composition mechanisms of OO with those of AOP by implementing a product line in C++ and in AspectC++ [214]. They have found that the AOP approach convincingly beats the OO version in terms of dynamic memory requirements and to a lesser degree in stack usage too, whereas run-time differences are dominated by hardware cost. Interfaces and late binding in the OO version can be implemented via generic advice[57] and introduction of (empty) methods, whereas RTTI or reflection can be implemented as specialised, non-generic advice for particular join points. Hence, the explicit inheritance relations and method calls of an OO system have been replaced by implicitly invoked advice, i.e. dependencies have been inverted. Note that build-time composition of aspects on the one hand obsoletes run-time structures like V-tables for registering late-bound components, but on the other hand restricts configurability to build-

[57] This is advice which is robust to join point variability, see Section 5.2.1.3.

time. This can e.g. be resolved by using run-time weaving [190, 189]. In any case, the work of Lohmann et al. [160] gives a second motivation of AOP's inversion of dependencies.

A third rationale for inversion of dependencies is given by Nordberg [182], who approaches the matter from an analogy between software development and mechanical/electronic engineering. The latter assembles mechanical/electronic parts via a standardised set of connectors, i.e. connectors are well-specified and remain stable, contrary to the parts they connect and the assemblies they compose. Nordberg [182] claims that in software development connectors are actually less abstract and less stable than parts, because there are too many different approaches and standards in use. The dependencies from the more stable parts on connectors hampers evolution as parts constantly need to be adapted to connectors. If connectors were implemented using aspects, these dependencies could effectively be inverted, i.e. the connectors would specify which parts should be connected and how, instead of the other way around. To facilitate CBSD, Nordberg [182] imposes a design rule [105] which restricts the scope of functional aspects to components and applies connector aspects between components. This technique identifies inversion of dependencies between higher-level components.

This section has discussed new composition techniques facilitated by AOP, i.e. inversion of dependencies, based on three different points of view. This introduces important changes at the source code level, which means that according to RC2 changes should be made at the build system level to keep source code and the build system consistent. At first sight, a build DAG seems capable of modeling this composition, as inverted dependencies are still dependencies. However, the fine-grained composition of aspects (Section 2.4.2), more in particular advice, with the base code precludes this. In addition, pointcuts do not explicitly expose the join points they match, as they conceptually correspond to an intensional description of join points instead of an enumeration[58]. Hence, based on RC2, co-evolution of source code and the build system is influenced by AOP's provisions for composition.

2.4.4 RC3: Interface Consistency vs. Incremental Compilation

The third root of co-evolution states that architectural interfaces in the source code are not always respected by incremental build processes. In the context of AOP, we discuss three points of view which have an impact on this root of co-evolution. Whole-program reasoning, fine-grained composition and aspect interaction conflict with the build system's requirements for incremental compilation, and hence harm co-evolution of source code and the build system.

The discussion of RC1 in Section 2.4.2 has shown that understanding of as-

[58] Note that in practice enumerations are frequently used if no intensional description can be specified.

pects requires whole-program reasoning. As shown by Kiczales et al.'s [135] AAIs, type checking of interface usage requires knowledge about all aspects by which the interface or its implementation modules are advised. The consequences of this have been made clear by the discussion of the NOV of an undisciplined AOP design (Figure 2.9). Changes in base code interfaces and modules conceptually require recompilation of affected base code modules *and* all aspects. Hence, although inversion of dependencies removes dependencies from the base code (Section 2.4.3), they are replaced by dependencies from the aspects to the base code modules. As discussed in Section 2.3.4.3, these dependencies represent new challenges for incremental compilation techniques to prune the build DAG. The new assumptions they need to make for pruning hamper the consistency of the module composition. This is similar to the problems associated with whole-program static analysis and incremental compilation [52, 9]. Hence, whole-program reasoning provides an explanation of the applicability of RC3 in AOP-based systems.

In addition to whole-program reasoning, the fine-grained nature of aspect composition precludes many traditional incremental compilation strategies. Aspects are not composed as an atomic unit with the base program, but each of its advices is composed separately. Hence, coarse-grained techniques like "trickle-down" (file-level) recompilation do not suffice anymore [96]. Instead, more fine-grained techniques which should be built into the weavers are needed to drastically reduce compilation time (similarly as for non-AOP systems [9]), or the build system should be able to model finer-grained composition of build units.

A third problem for incremental weaving is caused by aspect interaction, i.e. the different ways in which aspects may influence each other semantically. One of these interaction mechanisms which is of interest here, is structural interaction [137, 113, 114]. Pointcuts may e.g. depend on methods or annotations which are introduced by another aspect. Depending on factors like precedence of aspects, order of weaving, etc. the outcome of the weaving process may be different. However, this kind of interactions can only be identified with dedicated analyses [137, 113, 114] based on the semantics of pointcuts and ITD. These interactions are too fine-grained for build systems. Similar problems arise for behavioural interaction [74], i.e. differences in program execution caused by semantic interaction of advice.

Based on elements of whole-program reasoning, fine-grained composition and aspect interaction, this section has argued that consistent composition of aspects induces new challenges for incremental weaving (RC3). Just as for RC1 and RC2, AOP introduces changes at the source code level which require non-trivial changes in the build system. The next section presents the relation between AOP and RC4, the fourth root of co-evolution.

2.4.5 RC4: Program Variability vs. Build Unit Configuration

The fourth way in which AOP interacts with the build system is via the build system configuration. This section argues that AOP requires on the one hand stronger configuration support from the build system because the variability of an AOP system is higher, that the order of weaving is important and that there are implicit dependencies between configuration choices, and requires on the other hand a new kind of configuration support to shift variability from the base code into aspects. The first two arguments consider selection of module alternatives, whereas the third one deals with platform-dependent code and selection of features (see Section 2.3.4.4).

As shown by Sullivan et al. [219] and Lopes et al. [162], AOP systems have a larger NOV than non-AOP systems. Even without design rules like XPIs [105], aspects increase the potential of experimenting with new module implementations in compositions because they modularise CCCs. This requires better ways to deal with selection of modules, especially aspects, in the build system. The most general approach is to apply all aspects on the base modules of the whole system. Design rules like XPIs [105] or Nordberg's component-based approach [182] require finer-grained specification to select aspects and to limit their scope to well-chosen boundaries. Hence, aspects demand finer granularity and higher expressiveness of the build system's configuration layer.

Not only the scope of aspect configuration is important. Interactions and dependencies between aspects require more sophisticated control of the order of weaving by the configuration process. This becomes clear from two observations. First, Lopez-Herrejon et al. [163] have formalised aspect weaving as a process in which first ITD is performed on the base program and then one advice at a time is composed with the structurally woven base program. In their formalism, structural weaving (i.e. ITD) is commutative, whereas the operator used for applying advice does not necessarily have to. They give the example of AspectJ, where aspects which have been woven first, conceptually are re-woven into every structural program element introduced by later aspects. Second, we have seen in the previous section that structural [137, 113, 114] and semantic [74] interactions are important constraints on the order of weaving aspects. Hence, the order of weaving is an important factor when applying AOP. The configuration layer should be able to specify this order.

Implicit dependencies between aspects and possibly base code provide a third reason why more advanced aspect configuration is needed. First, aspects often depend on support code, i.e. auxiliary data structures or modules which are used inside advice. When a given aspect is selected for weaving, its support code should be selected too, otherwise not. Second, aspects may depend on or exclude other aspects [75], but there is no explicit specification of this. Hence, if the configuration layer cannot take these dependencies into account, semantically invalid woven

programs may be generated.

The three previous problems require extra features from the build system. However, AOP also provides opportunities for making configuration easier. If we consider the C preprocessor e.g. (Section 2.3.4.4), source code is heavily mixed with conditional compilation logic. Singh et al. [204] have argued that the preprocessor often is used to implement very fine-grained crosscutting concerns, i.e. the preprocessor can be interpreted as a very low-level aspect weaver. They have proposed to extract some of this conditional code into aspects instead of using the preprocessor. Aspects are e.g. not required if the whole implementation of a procedure is conditionally guarded, because the procedure can just be moved to a normal C compilation unit instead. In more fine-grained conditional compilation cases, extraction into advice makes sense. The advice's pointcut needs explicit access to the build configuration in that case. This means that aspect languages need to communicate with the build system to exploit aspect implementations of conditionally compiled code, i.e. to modularise the coupling between base code and build configuration. As a consequence, the NOV of the system increases again, which may trigger the three earlier described configuration problems.

This section has provided three reasons why AOP requires more flexible configuration support from the build system, i.e. higher NOV, order of weaving and dependencies between aspects, and one reason why AOP potentially simplifies the current configuration process if aspect language and the build system are able to communicate. This concludes the discussion of how AOP is influenced by RC4.

To summarise, in the previous four sections we have seen that AOP introduces changes to the source code for each of the four roots of co-evolution from Section 2.3.4. Hence, for each root of co-evolution the build system needs to evolve in order to maintain consistency between source code and the build system. The next section discusses how these claims are validated.

2.5 Validation of Co-evolution of Source Code and the Build System

This chapter has presented a model for build systems (Section 2.1) and has discussed understandability and scalability problems which affect them (Section 2.2). We have argued that tool support is a necessity to be able to deal with these problems. Then, we have discussed how management of build systems is further complicated by the phenomenon of co-evolution of source code and the build system (Section 2.3), for which we have found indications, a taxonomy and early experimental evidence. We have identified four roots of co-evolution (Section 2.3.4) which represent conceptual relations between source code and the build system. They relate changes in the source code to changes in the build system. To inves-

tigate these phenomena in practice, and to manage this co-evolution, tool support is indispensable as well. Finally, Section 2.4 has discussed AOP and the source code changes it introduces in the source code. Based on these changes, the roots of co-evolution predict corresponding changes to the build system.

The remainder of this dissertation consists of two parts. The first part is concerned with validation of the four roots of co-evolution of Section 2.3.4 and of tool support in legacy systems without AOSD, whereas the second part validates the predictions of the four roots, and proposed tool and aspect language support in legacy systems in which AOSD has been introduced. We briefly outline the structure of both parts.

Chapter 3 presents tool support for understanding and maintaining build systems, i.e. to deal with typical build problems in legacy systems, and for understanding and managing the co-evolution of source code and the build system. Five functional requirements are distilled from the build problems identified in Section 2.2 (goal T1) and the roots of co-evolution (goal T2). These are used to evaluate existing tools and approaches and to form the foundation of a new framework, MAKAO. MAKAO is used in Chapter 4 to analyse the co-evolution of source code and the build system in the Linux kernel build system. This analysis serves as a validation of MAKAO (goal T2) and of the four roots of co-evolution in the absence of AOSD. This concludes the first part of this dissertation.

The second part, starting with Chapter 5, discusses co-evolution in the presence of AOSD. We first discuss our own aspect language for C, Aspicere, which is able to deal with legacy systems (goal L1) and which provides means to interface with the build system (goal L2). Chapter 6 to Chapter 10 then present five case studies in which AOP is used for reverse-engineering (Chapter 6 and Chapter 7) and re-engineering purposes (Chapter 8 to Chapter 10). These cases validate the predictions of the four roots of co-evolution in the context of AOSD, the ability of MAKAO to deal with co-evolution in more general environments (T2), and the degree to which Aspicere meets the L1 and L2 goals. Chapter 11 summarises the contributions and findings of this work.

My motivation for [adding maintainer names] and [GNU Make version numbers to the build documentation] is that Makefiles are real source files and need the same maintenance and documentation as other source files.

Michael Elizabeth Chastain (Linux 2.4 build maintainer)

3

MAKAO, a Re(verse)-engineering Tool for Build Systems

THE previous chapter has shown how build systems suffer from a wide range of problems (Section 2.2). These problems not only hamper daily development, but they also complicate dealing with co-evolution of source code and the build system (Section 2.3.4). We have argued for tool support to resolve the identified build problems and to help in understanding and managing co-evolution of source code and the build system. The latter requires means to detect whether or not changes attributed to one of the four roots of co-evolution occur, and to react on them if necessary. This chapter[1] proposes such tool support.

First (Section 3.1), we define the scope of tool support by considering the build problems we want to tackle and by deducing how the roots of co-evolution can benefit from tool support. Based on this scope, Section 3.2 distills a list of concrete requirements for build system tool support, and it also considers important design trade-offs. These criteria are then used to evaluate existing tool support for build systems (Section 3.3). As no tool satisfies all criteria, Section 3.4 presents the design and implementation of MAKAO, a re(verse)-engineering framework for build systems. To evaluate the framework's capabilities to deal with build system problems, Section 3.5 demonstrates MAKAO in relevant use cases for each of the five requirements. Section 3.5.6 evaluates how MAKAO is able to deal with the build problems outlined in Section 3.1. Section 3.6 presents the conclusions

[1] This chapter is based on [3].

of this chapter. A thorough validation of MAKAO in supporting co-evolution of source code and the build system is given in the next chapter and in Chapter 6 to Chapter 10.

3.1 Scope of Tool Support

This section defines the scope of the build system tool support we focus on, based on the build problems of Section 2.2 and on the kind of tool support which is needed to identify and manage symptoms of the roots of co-evolution in practice.

3.1.1 Goal T1: Tool Support for Solving Build Problems

Of the build problems described in Section 2.2, many originate from limitations of build and configuration tools, e.g. External Dependency Extraction (2.2.1.2) or Imprecise Dependency Checks (2.2.3.2). We do not focus on this kind of problems, as they require enhanced build or configuration tool support.

Instead, we aim at automated support for understanding and maintenance of build systems, because most of the problems deal with this. The following problems require support for understanding: Implicit Dependencies (2.2.1.1), Syntax has Semantics (2.2.2.1), Advanced Language Features (2.2.2.1), Precedence of Variables (2.2.2.1), Debugging (2.2.2.1), Portability (2.2.2.2), Complexity (2.2.3.1), Traceability to Build Templates (2.2.3.1), Stakeholders (2.2.4.1), Overhead of Build Systems (2.2.4.1), and Recursive versus Non-recursive "make" (Section 2.2.4.2). Maintenance support is important for resolving Overhead of Build Systems (2.2.4.1) and Build System Changes (2.2.4.1).

Most of these problems have to do with restricted access to the right information, e.g. to find out which dependencies are implicit (Implicit Dependencies), to get feedback of build errors (Debugging) or to enable stakeholders to obtain the information they are interested in (Stakeholders). Other problems are caused by an overload of information (Precedence of Variables, Complexity, Stakeholders, Recursive versus Non-recursive "make", etc.). Lack of knowledge about build-time events and state (variables) is a third source of problems (Advanced Language Features, Precedence of Variables, etc.). Maintenance problems especially have to deal with context-dependent, invasive changes (Build System Changes) or the inability to understand the internals of the build (Overhead of Build Systems). We aim to come up with an automated solution for these problems.

Almost none of the identified problems actually deals with the configuration layer, for a number of reasons. The major channels of communication between source code and the build system for configuration purposes have been described in the context of GBS (Section 2.1.3.1). These channels have been known for years and provide a very fine-grained way of conveying configuration decisions to the

source code. Hence, they do not represent a problem, but may need support for understanding or traceability. The configuration specification on the other hand (cf. the autoconf specification of Figure 2.3 on page 19) is less defined. It deals with sets of features, constraints between them and a mapping on physical files. This corresponds to a better defined scope than the build layer has, as the latter deals with dependencies between files, compilation commands, shell scripts, time stamp-based heuristics, etc. Hence, configuration languages are conceptually easier to understand and to manipulate. Of course, if the small border (RC4) with the area of product lines (Section 2.1.4.6) would be crossed, the large scale of the source code configurability would lead to much higher demands on the configuration system. This is not the topic of this dissertation, however. Hence, we do not focus on explicit tool support for dealing with configuration specifications in the context of this dissertation.

This section has determined the kind of build problems we target for tool support. These problems require help in understanding and maintenance of build systems. We do not focus on configuration specifications. However, the communication channels between the configuration system and the source code can benefit from help in understanding and traceability.

3.1.2 Goal T2: Tool Support to Understand and Manage Co-evolution of Source Code and the Build System

This section considers the four roots of co-evolution to distill the features a tool should have to identify symptoms of co-evolution of source code and the build system, and to control the co-evolution.

The first root of co-evolution (Section 2.3.4.1) focuses on the mapping between source code components and build system components. The former requires knowledge about the architecture of the source code, i.e. the components in the source code, and the ability to manipulate it. Identification and manipulation of build system components on the other hand is based on clusters of build targets in the build dependency graph, and knowledge about the modularity of build scripts, i.e. whether build logic is stored in a monolithic file or spread over multiple ones.

Similar requirements exist for the second (Section 2.3.4.2) and third (Section 2.3.4.3) root of co-evolution. These need to know the components and dependencies in the source code architecture and have to map them on clusters of build targets and dependencies in the build dependency graph. Changes on either representation should be possible to sustain co-evolution. The third root additionally needs a means to determine which build dependencies have been sacrificed for compilation speed.

Finally, the fourth root (Section 2.3.4.4) needs fine-grained information of the parametrisation in the source code (platform-dependent code and selection of fea-

tures) and knowledge about selected source code modules in the build system. The interchange of configuration choices between source code and the build system is a third source of valuable data for RC4. Again, manipulation is needed to reinstate consistency between source code and the build system.

To summarise, information is needed about the source code architecture, source code parametrisation, the build dependency graph and communication of configuration choices between source code and the build system. At the same time, we need means to manipulate this data as well. In this dissertation, we exclusively focus on tool support for access to and manipulation of the last two sources of data. These are exactly the ones associated with the build system, because sufficient tool support for this area does not exist (cf. Section 3.1.1). There are many tools and techniques to understand and manipulate source code architecture, e.g. [173, 22, 176, 127, 29, 92, 111]. Likewise, a lot of support exists for dealing with parametrised code, e.g. regarding the C preprocessor [121, 138, 144, 145, 158, 209, 210, 110, 20, 140, 216, 236, 169, 235, 97]. An IDE for dealing with co-evolution of source code and the build system would need to combine tools for architecture, parametrisation, build DAG and configuration interchange, but in the context of this dissertation we assume that source code tools, documentation and experts are available separately to provide us with the relevant source code information and to manipulate the source code. Hence, we only focus on tool support for access to and manipulation of the build dependency graph and transfer of configuration choices to the source code.

3.1.3 Conclusion

Based on the previous two sections, we have determined three primary goals for tool support: understanding and maintenance of build systems, access to and manipulation of the build dependency graph, and access to the configuration choices which have been passed to the source code. Note that the last two goals are in fact subsumed by the first one. In this dissertation, we use the following terms interchangeably to refer to these three key goals at once: “understanding and maintenance of build systems”, “design recovery and maintenance of build systems” and “reverse- and re-engineering of build systems”. The next section derives concrete tool requirements for supporting the design recovery and maintenance of build systems.

3.2 Deriving Tool Requirements from T1 and T2

To address the three primary goals discussed in the previous section, we distill five important functional requirements which together guarantee a flexible re(verse)-engineering environment for build systems. In addition, we identify a number of

design choices to consider when trying to meet the requirements.

3.2.1 Functional Requirements

Five functional requirements are discussed in this section. They have been derived from the issues identified in the previous section.

3.2.1.1 Visualisation

Build systems typically consist of hundreds of scripts (Section 2.2.4.2), hence it is nearly impossible to get an overview of the complete system. It does not suffice to look at a single build script, as this usually works in concert with others (RC1). Besides this, the often cryptic build commands and dependencies, littered with configuration parameters, make the build logic very hard to understand (RC4). Hence, we need a visual representation of the *whole* system, but with sufficient detail to understand build subsystem composition. Visualisation also offers a build tool-neutral view of a build system.

Build systems contain a vast amount of information, for various stakeholders (2.2.4.1). Measures should be taken to make the data more digestible. One possibility is filtering of the visualisation (see the third requirement), but even simple features like color coding, layouting and zooming, should help. Another powerful way to manage the build system data is interactivity. On-demand visualisation of particular subsystems or highlighting of common information between different build targets enables the user to grasp detailed information he or she is interested in.

3.2.1.2 Querying

Visualisation can only give a qualitative idea of a build system. It should be possible to query for specific information (Section 3.1.1) about a particular build target, to access its command list, etc. As a build system has multiple stakeholders, the querying facility should be general enough to compose custom queries on the one hand, but not too complex so as not to scare off less programming skilled stakeholders on the other hand. An extensible default set of basic queries, accessible via a GUI, could enable this. A querying feature can also be used to filter information by selecting various targets based on some user-defined criteria, or to gauge certain build characteristics via metrics. This is another reason to make the querying facility general enough.

3.2.1.3 Filtering

To deal with information overload, the user should be able to filter out redundant information like unimportant files or build parts. However, we would also like to

define new views of the build, e.g. to abstract away low-level details of a build idiom, to generate a build-time view [228] or even to recover the design of the source code. Therefore, we want to enhance filtering with powerful, composable abstraction capabilities. This implies that a fixed set of default filtering actions does not suffice.

3.2.1.4 Verification

Visualisation, querying and filtering are a perfect fit for soliciting information to solve application-specific build system problems (Section 3.1.1). However, there are many recurring problems, style issues and bad practices for which common workarounds or recommendations exist. Redundant dependencies, circular dependencies or problems introduced by recursive build logic are bad smells which occur universally across build systems. Manual or visual inspection is not an effective means to detect them (2.2.2.1). Instead, an automatic verification facility should be provided which is able to model common build system patterns in a robust way (to overcome application-specific oddities) in order to flag the presence of problems. As the set of issues is open-ended, the verification functionality should be extensible. In the end, a library of verification patterns could be distilled and distributed.

3.2.1.5 Re-engineering

Visualisation, querying, filtering and verification can collect sufficient context information to decide whether build system re-engineering is needed and if so, how one should proceed. Being able to directly exploit this knowledge from within the tool improves the ability for robust and invasive re-engineering of the build system (2.2.4.1). Tools should be able to simulate the effects of re-engineerings without applying them on the actual build system. Afterwards, the modifications should be committed to the actual build system or just "rolled back". Special care is needed to ensure that the re-engineering remains applicable across all configurations.

3.2.2 Design Trade-offs

Tools which are designed according to the five identified requirements will inevitably face certain important design trade-offs. These are outlined in the next sections.

3.2.2.1 Lightweightness

A rather straightforward trade-off is the fact that, despite the advanced requirements, the tool should be lightweight and easy to integrate into the tool chain of developers, maintainers, etc. Otherwise, developers will rather try to apply ad hoc

techniques for their particular problem. Besides practical issues, the lightweight property also places constraints on the overhead induced by the tool on builds.

3.2.2.2 Static vs. Dynamic Model

A crucial decision is whether one wants to manipulate a model of the static build and configuration scripts, or of the dependencies within an actual (dynamic) build run instead. This is similar to the distinction between static and dynamic analysis of source code. In general, it is easier to obtain data from the dynamic build (e.g. via traces) than it is to analyse the static description of the build, largely because of the parametrisation of build scripts. The static data on the other hand abstracts over all information across any supported build platform, which makes it better suited to reason about and to refactor across various platforms. The complexity in reliably processing and linking build components together from the static data is, however, much higher. Common errors such as misunderstood macro expansions are particularly hard to resolve, because the actual values of variables and macros are only present during a concrete build. Unfortunately, they then correspond to just *one* run on *one* particular build platform. Hence, there is a trade-off between complete, static data which is hard to interpret, and specific, dynamic information which is restricted to one build platform.

3.2.2.3 Detecting Implicit Dependencies

The phenomenon of implicit dependencies, described in 2.2.1.1 on page 30, illustrates well why it is hard to work from the static description of build systems. Implicit dependencies hamper understanding of the build, incremental compilation and parallel builds, and may lead to inconsistent and incorrect builds. For these reasons, they should be tangible to the user, in order to really understand what is going on during a build and to be able to fix them if possible. Interpreting build script command lists to find out statically which strings represent files is not feasible because of parametrisation and flexible naming conventions. Conceptually, each file access should be monitored during the build and correlated later on with the right build rule to identify all implicit dependencies. This leads to build overhead. Hence, there is a clear trade-off between precision of the build model and effort required to obtain it.

The five requirements are used in the next section to evaluate existing tool support for build systems. Afterwards (Section 3.4), the requirements and trade-offs are used as the foundation of a dedicated framework for reverse- and re-engineering of build systems, i.e. MAKAO.

	Visualisation	Querying	Filtering	Verification	Re-engineering
Jørgensen [125]				X	
BTV [228]	X				
Dali [127]	X	X	X		
de Jonge [56]					X
Fard et al. [82]					X
Di Penta et al. [188]					X
Grexmk [13]					X
Kbuild 2.5 [156]					X
remake		X			
gmd		X			
Antelope	X		$\pm$		
AntExplorer	X		$\pm$		
OpenMake Build Monitor	X		$\pm$		
Vizant	X		$\pm$		

Table 3.1: Evaluation of existing tools to support build system understanding and maintenance w.r.t. the five requirements of Section 3.3.

3.3 Evaluation of Existing Tool Support and Techniques

This section evaluates related work based on the requirements presented in the previous section. We consider research efforts in the areas of formal methods, software reverse- and re-engineering, and specific build development tools. As we will see, none of the discussed approaches fulfills all our requirements. Querying, verification and general-purpose re-engineering remain largely unexplored. Our findings are summarised in Table 3.1.

3.3.1 Formal Methods

Jørgensen [125] addresses *verification of build systems*. Build safeness holds if all build rules meet the following criteria:

soundness Execution of a rule's commands generates the rule's target, and the

dependees have been established and are older than the new target.

fairness Execution of a rule's commands does not invalidate other rules' targets.

completeness A rule's target is only influenced by files in its list of dependees, i.e. there are no implicit dependencies.

These criteria suffice to prove soundness and completeness of executions of makefiles, and also to prove that "make"-based, incremental recompilation is guaranteed to give the same results as a full build. The latter only holds if compilation results are not modified at all, changes to makefiles do not introduce new build targets of which the name already occurs somewhere and command lists are not changed in between. Unfortunately, many of these conditions are not easy to check, as they require knowledge about side-effects. To prove soundness, fairness or completeness, one needs an explicit model of the effects of a rule's commands on target and dependees, which is not straightforward.

3.3.2 Understanding Build Systems

We present three cases of build system tools and techniques in the reverse-engineering community. As mentioned in Section 2.3.1, Tu et al. [228] have proposed the build time architectural view (BTV) as a proper addition to Krüchten's "4+1" View model [139]. It is mainly targeted at extracting and documenting (visualising) the high-level architecture of build systems. Hassan [111] claims that exploring the BTV is highly recommended when trying to analyse large software systems, which is in line with our discussion in Section 2.2.4. In practice, the BTV Toolkit uses the grok tool [175] to abstract up from low-level facts generated by an instrumented version of "make" (dynamic model of Section 3.2.2). The BTV Toolkit's current prototype only extracts build-time facts, although conceptually build time views also take source code into account. There are *no provisions for interactive querying, filtering, verification or re-engineering*. Our visualisation, querying and filtering requirements describe various ways of how to access the same data as a BTV provides. As such, tools adhering to our requirements are able to construct high-level build time views.

Champaign et al. [44] use build system data to validate a package stability metric, but implicit build dependencies potentially compromise the precision of their measurements. They have used the BTV Toolkit to obtain the build dependency data, but the tool kit's exclusively dynamic build model has precluded access to information about unevaluated targets, i.e. targets which have not been part of a build.

One of the reverse-engineering approaches mentioned in Section 2.3.1, Dali [127] (now "ARMIN"), exploits build-related facts to obtain module/file dependencies

for deriving a graph model of a system. In this model, human experts need to derive and define patterns (expressed as SQL queries) to gradually obtain a higher-level view. *For obvious reasons, none of the reverse-engineering techniques targets verification or re-engineering*. Querying is possible, but the provided build model focuses on high-level dependencies between files instead of fine-grained info about command lists or configuration parameter values.

3.3.3 Re-engineering Build Systems

This section discusses five related efforts for the re-engineering of build systems. As discussed in Section 2.3.3.2, de Jonge [57] has tried to align build system modularisation with source code architecture, in order to obtain effective reuse and recomposition of source code. For this, he (semi-)automatically splits up a code base into self-contained source trees, each with its own build and configuration process. Afterwards, these can be distributed and recomposed (on both source code and build level) into a new system. *This is a very specialised re-engineering approach.*

The technique of Fard et al. [82] is based on a "reflexion model". This model is used to derive how the source code should be restructured in order to simplify the dependencies and environment processing (Section 2.3.4.3) during the build. Hence, the build system is re-engineered by restructuring the source code. As a side-effect, the build and the software architecture become easier to understand. As this technique relies on modification of the source code, *it cannot be applied to build systems in general*.

Di Penta et al. [188] propose a framework for "renovating" software based on a dependency graph of binaries and object files. Using genetic algorithms, clustering techniques and human intervention, they are able to detect and throw away redundant object dependencies and clones from the build results. Hence, *they optimise the results of the build system by exploiting build dependencies*.

Ammons [13] considers the problem of untrusted incremental compilation because of uncertainty about the completeness and correctness of build dependencies. He proposes to semi-automatically partition a build in separate mini-builds, in practice one per loosely coupled build component. Mini-builds are scheduled incrementally and in parallel according to the available dependency information, but once they are executing they are constrained to a sandbox which only provides the mini-build with its declared dependencies. A mini-build specification is a directory with three files containing information like the names of the build, dependencies and build script ("control"), the list of exported files ("outputs") and the list of source files ("sources") respectively. Transformation into genuine makefiles ensures safe incremental compilation according to Jørgensen's theorems [125]. The Grexmk tool suite implements these ideas by preserving the original build

structure as much as possible. This *special-purpose re-engineering* is focused on accelerating the build in a safe way.

Some build systems are designed with easy modification of build rules in mind. The Linux Kbuild 2.5 (discussed in Section 4.5.3.1) has provisions to extend the build via build variables [156]. The following command invokes the kernel build with an extra build rule for building target `pkg`:

```
1 make ADD0=pkg ADD0_DEP=install \
2       ADD0_CMD=myscript.sh tmp
```

The rule itself states that `pkg` depends on target `install` and has only one command in its build (line 2). These user-provided rules have free access to any variable declared in the build scripts. This rule customisation facility is very powerful, but *has to be built explicitly into the build system*.

To conclude, only the approaches of de Jonge [56] and Ammons [13] focus on re-engineering the actual build system. Still, *no general purpose re-engineering* is possible, as the proposed techniques deal with special-purpose re-engineering. Only de Jonge, Ammons and Di Penta et al. [188] exploit knowledge mined from the build system.

3.3.4 Enhanced Build Tools and Systems

This section considers tools aimed at fulfilling one single requirement. Remake[2] is an improved GNU Make with tracing capabilities and a debugger. One can set breakpoints, step through the build and evaluate expressions. Another debugger named "gmd"[3] is implemented completely using "make" macros. These debuggers focus primarily on *querying*.

Tools like Antelope[4], AntExplorer[5] and Openmake Build Monitor[6] enable live visualisation of build runs. Makeppgraph[7] creates a build dependency graph in which colors are determined by file extensions. Vizant[8] is a similar tool for Ant files. *These programs only offer visualisation, sometimes with limited filtering control*.

Finally, there are some special-purpose tools to assist "make" users. Maketool[9] is an IDE for makefiles providing colored logs, collapsed directories, etc. Build Audit[10] transforms build traces in more structured HTML or text formats, while mkDoxy[11] is a documentation tool for "make" scripts.

[2] `http://bashdb.sourceforge.net/remake/`
[3] `http://gmd.sourceforge.net/`
[4] `http://antelope.tigris.org/`
[5] `http://www.yworks.com/en/products_antexplorer_about.htm`
[6] `http://www.openmake.com/dp/home/`
[7] `http://makepp.sourceforge.net/1.50/makeppgraph.html`
[8] `http://vizant.sourceforge.net/`
[9] `http://home.alphalink.com.au/~gnb/maketool/index.html`
[10] `http://buildaudit.sourceforge.net/`
[11] `http://sourceforge.net/projects/mkdoxy/`

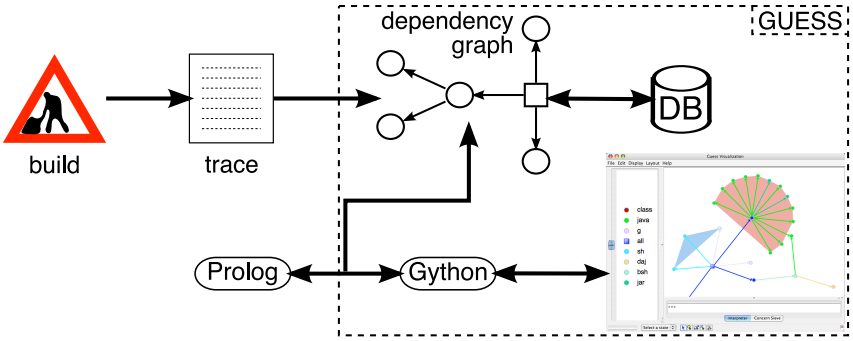


Figure 3.1: Outline of MAKAO's architecture.

3.4 Design and Implementation of MAKAO based on the Requirements

No tool or approach described in the previous section satisfies all our requirements. Many techniques focus on only one or two of them, or use build system data as a means to accomplish another task (derive architecture, slim down executables, etc.). However, we are explicitly interested in understanding and improving the build system itself. This section presents the implementation of a dedicated reverse- and re-engineering framework for build systems, named MAKAO[12] (*Makefile Architecture Kernel featuring Aspect Orientation*). Its design is based on the five requirements identified in Section 3.2 and it deals with the three identified design trade-offs. This section first describes MAKAO's architecture, and then elaborates on the various implementation and design choices.

3.4.1 Architecture of MAKAO

Figure 3.1 shows MAKAO's architecture, which is inspired by the reverse-engineering tools of Section 3.3.2. MAKAO expects as input a build dependency graph with crosslinks to the static build scripts and with the (dynamic) build-time values of configuration parameters and build variables. The graph and its attributes can be persisted into a database. Because MAKAO does not enforce a particular graph extraction approach, it is build tool-independent.

The dependency graph can be visualised, and interactively manipulated. Querying and re-engineering are possible using a built-in scripting language (Gython) through which the graph can be freely accessed. Filtering and verification facilities are provided via a Prolog engine. This engine runs in parallel with the Gython

[12] `http://users.ugent.be/~badams/makao/`

engine, managing a synchronised model of the graph.

The following sections describe in detail the important implementation and design decisions of MAKAO.

3.4.2 Build System Representation

From the beginning, "make" is based on a Directed Acyclic Graph (DAG) [89] in which nodes are build targets and edges represent dependencies between those targets. Newer build tools are still based on this model. DAGs have lots of favourable characteristics, one of them being their natural visualisation (first requirement). The presence of a configuration layer on top of a build layer does not invalidate the underlying DAG model. It only makes DAG extraction harder if one has to take unresolved configuration parameters into account, i.e. if one were to choose a static data model (Section 3.2.2.2).

Hence, for the second trade-off, we have opted for a hybrid approach in which dynamic build data is enhanced with static build script information (the actual build rules, unevaluated targets, etc.). The dynamic build data is obtained somehow from the data of a particular build run. Targets and dependencies encountered within such a build run can be linked to the build script rules and command lists by relying on names and line numbers. Starting from the dynamic model of a particular configuration makes sense, as build problems are usually first encountered in a build on a specific platform. This specific data can be generalised afterwards to other configurations.

3.4.3 Build Dependency Graph Extraction

The only way to reliably obtain a build DAG is to extract it from the build tool's internal build model (in memory) during a typical build. MAKAO initially focused on GNU Make (Section 2.1.3.3), as this is the de facto build tool used in the various case studies we have performed. However, as long as a DAG can be extracted, the remainder of MAKAO's architecture is build tool-agnostic. In fact, apart from GNU Make, we already provide support for ClearMake[13].

Retrieving a build's dependency graph from GNU Make can either be done using a modified "make" (as "BTV Toolkit" does; see Section 3.3.2) or by parsing the trace output produced by the build tool. Because of loose coupling and low overhead (first trade-off), we currently use the latter option. We have implemented parser scripts to extract the right data from the trace and to generate a graph from it.

To detect implicit dependencies (third trade-off) we leverage the Bash shell's "xtrace" option. During a build, this prints every single executed command to

[13] `http://www-306.ibm.com/software/awdtools/clearcase/index.html`

```
1 nodedef> name,localname VARCHAR(255),makefile VARCHAR(255),\
2          concern VARCHAR(50),error INT,...
3 t1,subdir/file.o,/path/to/subdir/Makefile,o,0,...
4 t2,subdir/file.c,/path/to/subdir/,c,0,...
5
6 edgedef> node1,node2,directed,tstamp INT,implicit INT,...
7 t1,t2,true,2613,0,...
```

Figure 3.2: Sample .gdf file representation of a build dependency graph.

Name	Type	Meaning
name	string	unique ID
localname	string	target name as occurring in the Makefile
dir	string	full name of the target's directory
base	string	local target name relative to "dir" attribute
makefile	string	name of Makefile (or directory if target is a file)
line	integer	line number in Makefile (or "-1" if target is a file)
concern	string	name of target's concern (.o, .c, ...)
error	integer	target's error status (or "0" if no error)
phony	boolean	is target phony (see Section 4.5.3.2)?
inuse	boolean	has the target been used in the current build trace?
style	natural	symbol used to visualise the target
color	string	color of target

Table 3.2: Attributes of nodes of a build DAG.

the trace file, with all arguments expanded. We then apply a heuristic to identify which strings represent file names. Currently, we look for all file name-like strings containing a dot (e.g. "A.class") which are not listed as target or dependee of the currently processed rule. This ignores implicit dependencies on files like Linux binaries which typically have no extension. False positives can also occur. An alternative we are currently investigating is to run "make" through "strace" (like Maak does [62]). This explicitly prints out any files which are opened and closed during a build, but this data is harder to correlate with the build rules.

Name	Type	Meaning
node1	string	ID of source node
node2	string	ID of destination node
directed	boolean	is the edge directed?
ismeta	boolean	is this a meta-edge?
tstamp	natural	discrete time stamp of edge
implicit	boolean	is edge implicit?
color	string	color of edge

Table 3.3: Attributes of edges of a build DAG.

```
1 main_target(t1).
2 target(t1, 'file.o').
3 in_makefile(t1, m0).
4 target_concern(t1, 'o').
5 path_to_target(t1, ['', 'path', '', 'to', 'subdir']).
6 makefile(m0, ['', 'path', '', 'to', 'subdir'], 'subdir').
7 error_flag(t1,0).
8 %analogous for t2
9 dependency(t1,t2,t1:t2).
10 dependency_time(t1:t2,2613).
11 dependency_implicit(t1:t2,0).
```

Figure 3.3: Prolog representation of Figure 3.2.

3.4.4 Implementation on Top of GUESS and SWI Prolog

We have built MAKAO on top of GUESS [10][14], a graph exploration framework with a built-in Python-based scripting language ("Gython"). Graphs can be loaded from a file or from a database. Figure 3.2 displays a snippet of a very simple .gdf file which models a graph of two nodes and one edge. First (lines 1–4), the two nodes are defined by assigning values to the (typed) attributes declared on lines 1–2. Node `t1` e.g. denotes a target in the Makefile located in `/path/to/subdir`. The target's build concern is `o`, i.e. it is a C-like object file. No error has occurred during execution of the corresponding build rule (`error` attribute has as value `0`). Edges modeling build dependencies can be specified analogously (lines 6–7). Implicit dependencies have `1` as value for the `implicit` field. In total, MAKAO defines twelve attributes for nodes (see Table 3.2) and seven for edges (see Table 3.3). Besides these, there are some built-in attributes like `visible` (is node or edge visible or not?), `inEdges` (incoming edges of a node), etc.

[14] `http://graphexploration.cond.org/`

Once loaded, the graph can be visualised and interactively manipulated, e.g. by zooming in on particular subgraphs. All nodes, edges and hulls are Gython objects (in the OO sense) with their own behaviour and the user-definable attributes mentioned previously. These are accessible within the Gython engine for querying, navigation and re-engineering of the graph. We give examples later on. One can also write Gython scripts to enhance GUESS with new views or panels, or to customise it for a particular domain, as is done by MAKAO.

While Gython could also be used for filtering and verification, symbiosis with a declarative rule-based approach offers more advantages for these [174]. We therefore integrated the SWI Prolog engine [223][15] into MAKAO, in which an equivalent virtual representation of the graph model is kept in sync with GUESS's internal model. Figure 3.3 displays the Prolog equivalent of Figure 3.2's graph. On line 1, target `t1` is declared as the start node of the (simple) build graph. The Prolog facts on lines 2–8 specify the various attributes of each target. The same is done for the dependencies on lines 9–11.

Conceptually, MAKAO should forward every change on GUESS's internal graph model to the Prolog engine and vice versa. However, the current prototype only allows off-line filtering of the DAG within the Prolog engine, followed by starting up an instance of GUESS with the resulting graph.

3.4.5 Re-engineering of the Build System using Aspects

There are various ways to model build system re-engineering, but we have chosen an aspect-oriented [134] approach. For limited re-engineerings, the best re-engineering techniques is to manually modify the build scripts, guided by MAKAO's four reverse-engineering features. Once changes need to mingle with e.g. existing command lists or need to be distributed across various places in a context-dependent way (2.2.4.1), tool support is required. In AOSD terminology, the first phenomenon is named "tangling", the second one "scattering". Because AOSD explicitly addresses these kinds of problems, we think it is a good fit for advanced build re-engineering.

Applied to build systems, a pointcut selects all join points where advice should be woven during the build. These are moments during a build, like command invocations or dependency checks, where one would like to enhance (advise) the existing behaviour with new commands, extra dependencies, new rules, etc. Weaving in MAKAO can be done in two modes:

virtual change GUESS's internal graph model

physical change the actual build scripts

[15] `http://www.swi-prolog.org/`

This duality enables the user to non-destructively experiment with changes before physically altering the build scripts. To enable platform-independent build changes, the re-engineering specification should be expressed in terms of build and configuration parameters instead of platform-dependent values.

3.4.6 Summary

To conclude, MAKAO satisfies the requirements of Section 3.2 in the following way:

visualisation build dependency graph

querying GUESS's embedded Gython language

filtering Prolog predicates

re-engineering aspect oriented techniques

verification Prolog predicates

The trade-offs have been taken into account as well:

1. Build tool-specific graph extraction (external to MAKAO) limits build overhead.
2. A hybrid build model is used which links data from a concrete build to the static build scripts.
3. Implicit dependencies are detected via a heuristic.

The next section applies MAKAO on relevant use cases for each of the five requirements. This is used in Section 3.5.6 to evaluate how MAKAO deals with the targeted build problems of Section 3.1.1 (goal T1). Validation of MAKAO for understanding and managing co-evolution of source code and the build system happens in the next chapter and in Chapter 6 to Chapter 10.

3.5 MAKAO at Work: Achieving Goal T1

This section evaluates the ability of MAKAO, and hence the five requirements of Section 3.2, to tackle the build problems we have decided to target in Section 3.1.1, i.e. goal T1. Validation of MAKAO in the context of co-evolution of source code and the build system, i.e. goal T2, follows in later chapters. To evaluate MAKAO, and indirectly the requirements of Section 3.2, for goal T1 we have applied the framework on examples for the build problems of Section 3.1.1. Most of these examples are situated in the industrial Kava system, Quake 3 Arena or the Linux

2.6.16.18 kernel. We first describe these systems. Then, we consider the requirements one by one. Afterwards, Section 3.5.6 evaluates how MAKAO deals with all build problems associated with goal T1.

Kava (Royal Pharmacists Association of Antwerp) is a non-profit organization of over a thousand Belgian pharmacists. Some ten years ago, they have developed a suite of C applications which were built using "make". Due to successive health care regulation changes, this service has been re-engineered, and migrated to an ANSI C platform. The build system comprises 272 makefiles, but no configuration scripts. Using sloccount [241], we have measured 4683 non-comment, non-whitespace lines (SLOC) in these makefiles.

Quake 3 Arena is a high-profile, commercial 3D video game [191][16] written in C. It has been released as open-source in August, 2005. Revision 1041 (February, 2007) contains 6 build scripts ($\pm$2000 SLOC), but no configuration scripts. Quake 3 represents a middle ground between the closed source character of Kava and the open source Linux kernel.

The Linux kernel (discussed in depth in Chapter 4) has been analysed extensively by other researchers [29, 99, 44]. It has [154] a custom build layer named "Kbuild"[17] based on GNU Make, and the "Kconfig"[18] configuration layer for flexible configuration of the kernel modules. There are about 859 build scripts ($\pm$13147 SLOC) and 345 configuration files ($\pm$51489 SLOC) involved in the 2.6.16.18 kernel.

We now report on the results of use cases performed in the context of the above three systems. We do this in the order of the requirements set out in Section 3.2.

3.5.1 Visualisation

We have visualised the Kava, Linux and Quake 3 build dependency graphs to qualitatively derive interesting information (Stakeholders problem) about the build systems and source code architecture.

3.5.1.1 Kava

Figure 3.4a shows the dependency graph (4040 nodes and 6207 edges) of Kava's build system, as visualised by MAKAO. The overall layout of this graph has been taken care of by one of the force-directed layout algorithms built into the GUESS framework. Figure 3.4b shows a more detailed view of the marked subgraph of Figure 3.4a. Every target has a color based on its concern, i.e. the type of file indicated by its extension, while edges have the same color as their destination node. By default, MAKAO defines only a couple of build concerns, like .c source

[16] http://www.idsoftware.com/games/quake/quake3-arena/
[17] http://kbuild.sourceforge.net/
[18] http://www.xs4all.nl/~zippel/lc/

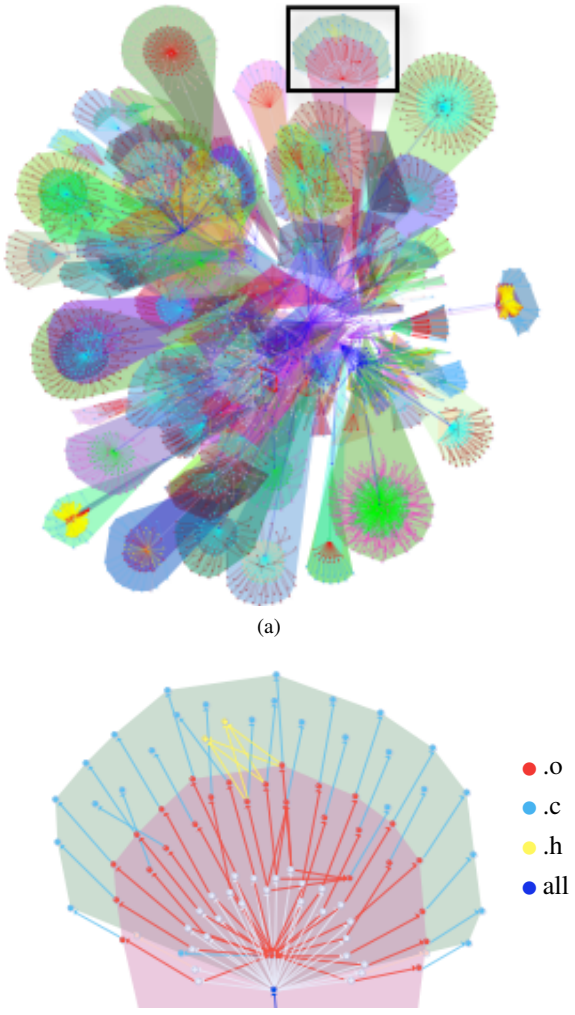


Figure 3.4: (a) Kava's build dependency graph in MAKAO. (b) Detailed view on the marked subgraph.

files and .o object files. One can add other concerns to MAKAO's list and assign a color to them, or even a different symbol if too many colors have been used already. By doing this we can differentiate the different types of targets.

The filled polygons on Figure 3.4a and Figure 3.4b are convex hulls, which enclose all nodes having the same value for a specific characteristic. We have chosen as characteristic the name of the makefile which specifies the target. The transparent colors of these hulls are chosen at random by GUESS and convey no semantic meaning. Hulls just aid visual recognition, but can be turned off if deemed unuseful.

Our particular choice for the hulls' semantics allows to identify recursive "make" systems. On Figure 3.4a, we see that most hulls have only one incoming edge and possibly multiple outgoing ones. This means that a hull can be considered in this case as a "make" subprocess spawned by another process (hull) to build a particular target. The hulls are disjoint and diverging and they do not originate from the build dependency graph's start node. Taking into account that typically a directory contains at most one makefile (hull), we get strong indications that the Kava build is recursive. Browsing through the build scripts quickly acknowledges this.

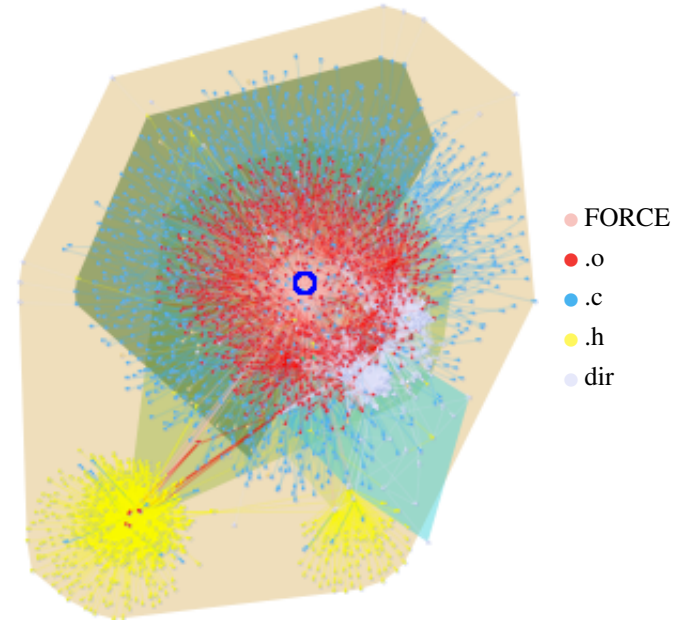
3.5.1.2 Linux 2.6.16.18

Figure 3.5a shows the dependency graph (3015 nodes and 8308 edges) of the Linux 2.6.16.18 kernel, more in particular the "bzImage"-target. This DAG looks very different from the one of Kava. Apparently, there is a strange light purple core which depends on object files. There are also two clusters of header files. These correspond to two places where custom and system header files are checked. There are not many hulls, and the ones which are there are rather concentric. The DAG is also very dense, so this does not look like a recursive build at first sight, although it is [154] (see Chapter 4). We discuss in Section 3.5.3 how we can untangle this DAG to get a better understanding.

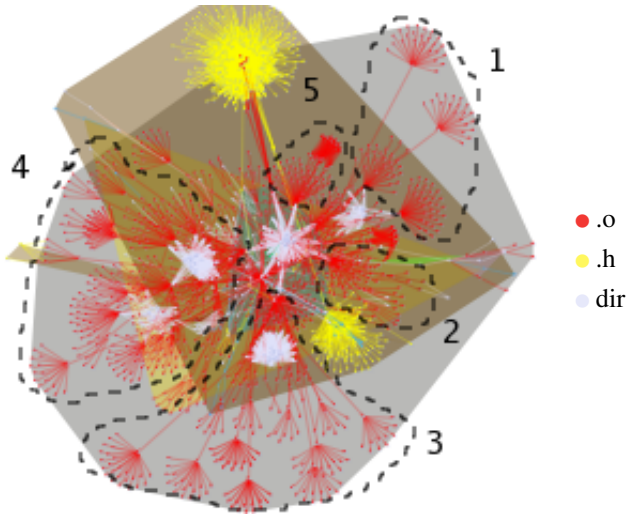
3.5.1.3 Quake 3 Arena

More can be learnt from Quake 3's build graph in Figure 3.6a (626 nodes and 908 edges). There are five clusters of compiled .c files, of which three yield dynamic libraries (black .dylib nodes), while the other two (the lower clusters) represent executables. Figure 3.6b zooms in on the top left cluster (dynamic library).

A strange pattern can be identified in this cluster, as it actually consists of two subclusters. Gauging from the names of the two libraries, these two library versions form a kind of mini-product line which consists of the original Quake 3 game ("baseq3") and an expansion pack ("missionpack"). The build dependency graph immediately makes it clear how this product line is implemented: conditional compilation. Indeed, we see that most .c files have two incoming edges, one from each



(a)



(b)

Figure 3.5: Linux kernel 2.6.16.18 ("bzImage"), (a) before and (b) after filtering.

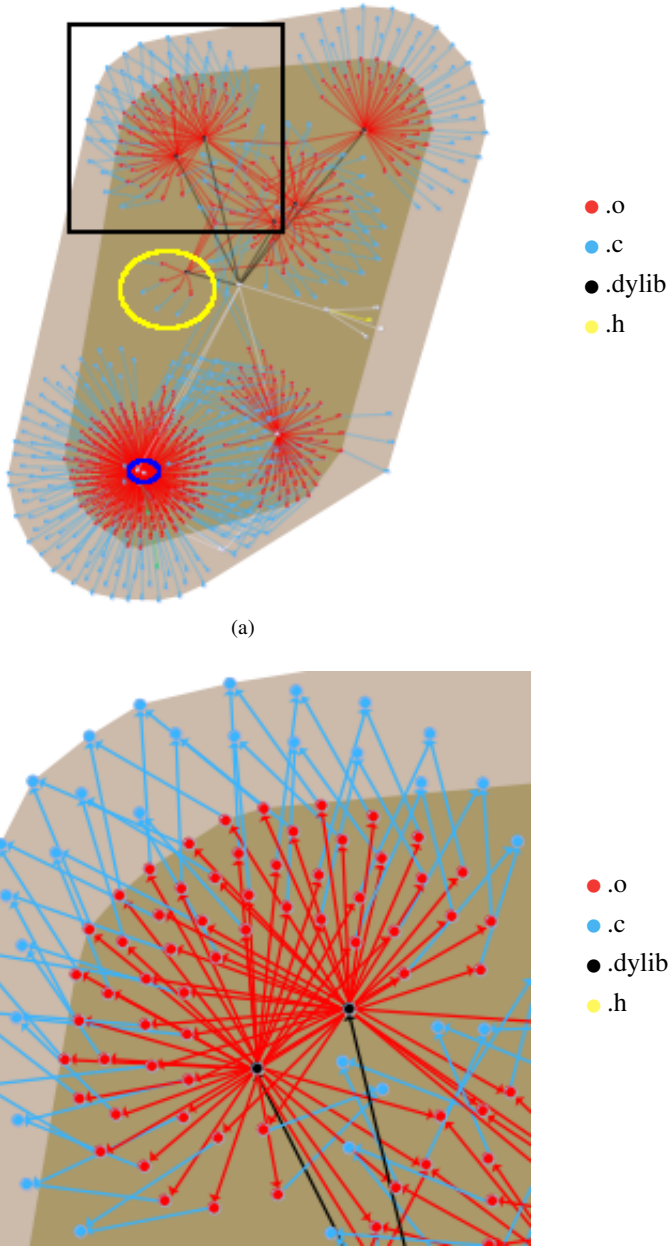


Figure 3.6: Build DAG of Quake 3, (a) in full and (b) zooming in on the subgraph marked in black. The yellow cluster does not use conditional compilation, while the blue ellipse marks the two variants of the main executable.

product variant. Hence, the same file is used for multiple variants. Inside the source files, struct members and logic for new weapons are indeed conditionally guarded.

One other dynamic library exhibits the same symptoms, whereas the third one does not (the small cluster marked in yellow). It contains only new files compared to the base library. We have checked this with the build scripts, and it seems that the "missionpack" version of that dynamic library requires the base version to be present instead of being autonomous.

Two other things can be observed visually. The lower left cluster of Figure 3.6a is also made up of two subclusters. As file names suggest, the variability ranges between single processor and SMP versions of the game executable (marked by a blue circle). Both versions are composed of the same object files, they just use other flags at link-time. The second oddity is that the Subversion administration file ".svn/entries" is a dependee of three object files. Every time the working directory is updated or reverted, those object files have to be remade, even if the corresponding .c files did not change. It turns out that the versioning info is used to display the current source code version when the corresponding executable starts up.

To summarise, even without querying support, one gets a useful qualitative view of a build system, which can even be used to obtain concrete ideas about the high-level source code architecture.

3.5.2 Querying

We now show four examples of Gython queries to access more specific information from the build dependency graphs and the actual build run.

3.5.2.1 Error Detection

Build errors are often hard to remedy (Debugging problem). Many builds try to survive across failed compilation attempts by continuing the build process to construct other build targets. As a consequence, the exact error message of intermediate errors is hidden somewhere within the vast build output, which comprises compiler invocations, changes of the current directory, etc. Once the right output line is finally located, it becomes even harder to make sense of the exact circumstances (missing file, wrong library version, etc.) under which the build has failed.

In MAKAO, we can write a simple Gython query to isolate the build context of all build errors from unimportant build information. The query boils down to keeping only the paths within the graph which have led to an error:

```
1 (error==0).visible=0
```

Figure 3.7: Error paths in Kava's build system.

The expression between the parentheses gives us the so-called "node set" of all targets in which the "make" `error` status (Figure 3.2) is zero. We then just need to hide all these targets by setting their `visible` node attribute to zero. Visually, this leaves us with a minimal subgraph which contains only the paths starting from the main build target to all failing targets. This makes it much easier to locate the exact circumstances of build errors.

We have applied this query on the Kava build system, and have found a number of erroneous build paths (Figure 3.7). From this, the Kava developers have been able to verify that the failing targets actually correspond to dead code. Those build failures were not noticed before because of this. The above query has enabled Kava to clean up this abandoned code.

3.5.2.2 Tool Mining

When one is confronted with an unfamiliar build system, a first clue about its internals is given by the file types ("concerns") which appear. If we observe dependencies from .o files to .c files, we can assume that this corresponds to compiled C source code. However, the transformation from source code to object file may vary because of different compilers, symbol strippers, obfuscators, etc. Parametrised build commands make this information hard to deduce from the build scripts. Hence, finding out the particular compilers in use is a useful query (Stakeholders and Overhead of Build Systems).

For each build target, MAKAO stores both the static and the actual command list in a dictionary (`commands` and `actual_commands` respectively). In the latter, all configuration parameters and build variables have been replaced by their actual value. By randomly browsing the `commands`- or `actual_commands`-dictionary for some of the object files' command lists, one can get an initial idea about the compilation tools in use. To find out whether these are the only tools in

use, we can issue the following query:

```
1 Ts=(concern=="c").inEdges.node1.findNodes()
2 base=[T for T in Ts
3         if not using_tool(T,["gcc"])]
```

The first expression finds all source nodes (`node1`) of edges pointing to .c targets, i.e. all targets depending on targets of the `c` concern, and binds this set to the variable `Ts`. Lines 2–3 form a list comprehension which yields all targets `T` which are not using any of the compilers we have already found. The `using_tool`-function internally performs regular expression matching on the actual command list of the target passed as its first argument.

The reason why we did not just store all .o files in `Ts`, is that other kinds of source files may also be compiled into object files. In the Kava build system, .ec files behave like this. These other types of source files can be detected like this:

```
1 set((concern=="o").outEdges.node2.findNodes().concern)
```

This expression finds the set of concerns of those targets which represent dependees (`node2`) of .o files.

3.5.2.3 Name Clash Detection

A "make" process has only one global namespace shared by all targets, environment variables and macros. This can lead to unexpected results in non-recursive builds (Section 2.2.4.2) where dozens of files are combined into one main makefile. MAKAO can detect these problems like this:

```
1 clusters=[c for c in groupBy(localname)
2             if len(c)>1]
3 clusters.makefile
```

This snippet uses a list comprehension to first cluster the targets based on their unqualified name (node attribute `localname`), and then only keep the clusters with more than one target. On line 3 we print out the `makefile`-attribute of all targets within the returned clusters. If there is a cluster with different values for the `makefile` attribute, there is a real name clash (Debugging).

3.5.2.4 Where do Compiled Objects End up?

Another typical build problem is finding out where build artifacts are stored, e.g. when trying to locate intermediately generated files to debug them (Overhead of Build Systems). This is quite easy, e.g. for the file named "built.o":

```
1 build_dir=by_localname("built.o")[0].dir
```

This query stores into variable `build_dir` the name of the directory (`dir` node attribute) in which target "built.o" resides. The `by_localname` function

```
1 force_target(Force, 'FORCE') :-
2   clean_target_a(Force, 'FORCE').
3
4 force_dependency(Target, Force, Key) :-
5   force_target(Force, _),
6   clean_rdependency_a(Force, Target, Key).
7 % - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
8 forceless_cached:-
9   forall(clean_target(Target,Name),
10         assert(clean_target_a(Target,Name))),
11  forall(clean_dependency(Src,Dst,Key),
12          (assert(clean_dependency_a(Src,Dst,Key)),
13           assert(clean_rdependency_a(Dst,Src,Key)))).
14
15 forceless_target(Target, Name):-
16   clean_target_a(Target, Name),
17   \+ force_target(Target,_).
18
19 forceless_dependency(Target, Dependency, Key):-
20   clean_dependency(Target, Dependency, Key),
21   \+ force_dependency(Target,Dependency,Key).
```

Figure 3.8: FORCE idiom filtering step (extracted from Section B.4 on page 326).

finds all nodes of which the `localname` attribute matches the regular expression passed as its argument (we assume that we have verified in Section 3.5.2.3 that there is only one match).

In addition, we might check whether or not multiple makefiles use this output directory:

```
1 len(set((dir==build_dir).makefile))
```

This expression returns the number (`len`) of unique (`set`) makefiles of which targets are produced inside `build_dir`. Deducing such information from the build trace or scripts is much harder to do.

3.5.3 Filtering

In Section 3.5.1, we have seen that the Linux DAG of Figure 3.5a is very dense. In order to remedy this, one could try to manually collapse nodes one by one, or maybe to script these actions [180]. Unfortunately, this approach is too coarse-grained and limited. Just as for source code, developers agree upon project-specific idioms to streamline the build process. As a simple example, the Linux build dependency graph contains some dependencies from .c files to files of which the name ends in "_shipped". This is a pattern used for shipping binary blobs or generated files with the kernel source code. It enables developers to override the default

shipped file by overwriting the .c file. Without fine-grained filtering support, one cannot write a crisp pattern to capture the essence of the build idiom. Hence, one risks to filter out too many or too little nodes. In this sense, exploiting knowledge of build idioms leads to more effective filtering (Stakeholders). At the same time, filtering the build dependency graph also gradually raises the abstraction level. If no crucial build dependencies are removed, one ends up with a high-level view of the source code architecture as recorded within the build system.

To this end, MAKAO allows to apply a user-customisable sequence of logic rules to detect patterns in the graph and to remove/add edges/nodes based on the results. The predicates can use the logic build dependency graph representation as shown in Figure 3.3 and any cached facts. Besides helper predicates, each filtering step consists of:

1. an initialisation predicate, e.g. to cache previous results
2. a predicate which selects the targets to be retained
3. a predicate for the dependencies one wants to keep

We have applied MAKAO's filtering to the Linux 2.6.16.18 build graph of Figure 3.5a, which forms a part of the experiment in Chapter 4 to investigate the evolution of the Linux kernel build system. Section 4.5.3.2 extensively elaborates on the build idioms we have uncovered, but we do not explain all of the actual rules we have used to filter these idioms (listed in Appendix B). In this section, we focus on one of the rules (Figure 3.8) to show how the filtering rule formalism works. The other rules can easily be understood based on this explanation. We have chosen to discuss the rule for the "FORCE" idiom because of its conceptual simplicity, but clear visual effect.

We have already mentioned the strange light purple core of Figure 3.5a in Section 3.5.1.2. When zooming in, we observe a node named "FORCE" on which all object file targets depend (the central node marked in blue). It is because of this node that the Linux build DAG turns out to be compact and tied together. The filtering step we have used to filter out this node is shown in Figure 3.8.

The rules above the dashed line indicate helper predicates[19]. The first one identifies the central "FORCE" target based on its name. The second predicate captures all edges pointing to this node. In other build systems, the first rule should be adapted to match naming conventions there. In the Linux 2.4.0 graph e.g., there are multiple "FORCE"-like targets named "dummy".

Below the dashed line, the three predicates of this filtering step are specified. The initialisation rule on lines 8–13 makes sure that all resulting facts of the previous filtering step (the "clean" filtering rule of Section B.3 on page 325) are

[19] A single underscore is a "don't care" value, while variable names start with a capital letter.

cached. For every edge between nodes a and b, it also asserts a fact representing the inverse edge from b to a. This is for efficiency purposes, as querying for `rdependency(a,_,_)` is typically much faster than for `dependency(_,a,_)`. This is especially important in the early filtering passes, where lots of nodes and edges are still present. The target selection predicate (lines 15–17) keeps all the targets passed from the previous filtering step (line 16), except for the "FORCE" target (line 17). Similarly, the dependency selection predicate (lines 19–21) filters out all edges pointing to the "FORCE" target (line 21).

The filtered build DAG is shown on Figure 4.13 on page 131. The DAG has opened up and a conventional, hierarchical visualisation appears. Subsequent filtering rules can be applied on idioms which now become visible (see 4.5.3.2). The "FORCE" idiom is only one of the seven filtering steps we have applied to unravel the internals of the Linux 2.6.x build system in Chapter 4:

- elimination of meta-edges (see `ismeta` in Section 3.4.4)
- cleanup of auxiliary files
- FORCE idiom
- shipped targets (see earlier)
- source-level abstraction
- composite objects (4.5.3.2)
- circular dependency chain (4.5.3.2)

All of the filtering steps follow the same outline as Figure 3.8. The semantics of the major idioms are described in detail in Chapter 4. The last two idioms (and the shipped targets) are highly specific to the Linux build system; the others can be reused in other systems. After applying the rules, the build dependency graph opens up completely and results in the much more structured Figure 3.5b. Here, we can identify various subsystems: (1) network support, (2) the kernel image, (3) file system code, (4) drivers and (5) architecture-dependent facilities. This graph can be the starting point for further filtering, e.g. by clustering tightly coupled nodes.

3.5.4 Verification

Automated verification of common build bad practices and anomalies like those in Figure 3.9 is important (Debugging). Manually detecting missing dependencies (Figure 3.9a) or loops across "make" processes[20] (Figure 3.9b) is possible, but is tedious and error-prone to repeat across multiple build systems. Applying Gython

[20] GNU Make has support for detection of loops within one process.

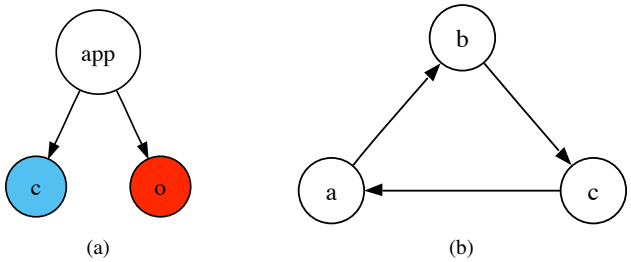


Figure 3.9: Typical build dependency graph anomalies for which verification should be used.

scripts is more interesting in this regard. Unfortunately, most patterns are awkward to express imperatively, e.g. if Figure 3.9b would extend to arbitrary-length loops. The declarative Prolog representation of the build graph is preferable for describing and checking common mistakes or style abuses. As a simple example, the following verification query finds all targets visited more than once, which could give indications of suboptimal builds:

```
1 redundant(Target,List):-
2  target(Target,_),
3  findall(R,rdependency(Target,_,R),List),
4  nth0(1,List,_).%at least 2 elements
```

The following rule detects the pattern of Figure 3.9a:

```
1 mixed(SrcNode,ObjNode):-
2  target(SrcNode,SrcName),
3  concat_atom([Root,'c'],'.',SrcName),
4  concat_atom([Root,'o'],'.',ObjName),
5  rdependency(SrcNode,Node,_),
6  dependency(Node,ObjNode,_),
7  target(ObjNode,ObjName).
```

The rule considers all targets (line 2) of which the name ends with `'c'` (line 3), i.e. a blue node on Figure 3.9a. The idea is to locate a matching red node, if it exists. For this, the object file name is constructed (line 4) by replacing the `'c'` by an `'o'`. The red node with the right name (line 7) has to be a sibling of the blue node (lines 5–6). If this situation occurs in the currently verified build DAG, this is immediately reported.

Other verification rules may focus on loop detection across “make” processes (Figure 3.9b), redundant dependencies (also transitively), detection of non-object nodes pointing to source files, etc.

3.5.5 Re-engineering

In this section, we elaborate on AOP-based re-engineering of a build (Build System Changes). Information recovered through one of the four reverse-engineering steps (visualisation, querying, filtering or verification) is a prerequisite to assess a build problem, to decide whether to re-engineer manually or via MAKAO, and to collect the necessary context information.

MAKAO's aspect-based re-engineering consists of the following steps:

1. locate join points and context in the build DAG
2. compose advice
3. weave the advice

To illustrate these steps, we integrate a source code preprocessing tool into Kava's build system. The difficulty is that this tool has to be applied to each C-like source file before the file is processed by a compiler. We assume that alternative solutions like compiler wrappers or regular expression transformations are not able to help us, either because they require thorough manual inspection or because consistent re-engineering of the build cannot be guaranteed [248]. Access to build context information in MAKAO's aspects helps to avoid these pitfalls.

3.5.5.1 Selecting the Join Points and the Join Point Context

The join point model of a build system is rather straightforward. It comprises behavioural crosscutting on commands in a rule's command list, and static crosscutting on a rule's dependency list or a build system's collection of rules. In the current MAKAO prototype only the behavioural part has been implemented.

In our example, the join points we are interested in are compiler invocations on source code files. In terms of the build DAG, this means we first have to find the targets `T` which manipulate .c files (`concern` node attribute), and then have to look up the relevant commands `C` in `T`'s `commands` dictionary. Join points and context are defined by the following Gython expressions:

```
1 Ts=(concern=="c").inEdges.node1.findNodes()
2 base=[(C,tool,T) for T in Ts
3                  for C in commands[T.name]
4                  for tool in ["CC","gcc"]
5                  if C.find(tool)!=-1 ]
```

On line 1, all targets depending on .c files are selected. We could have looked directly for .o nodes instead, but then we would also have captured those object files built from e.g. embedded SQL files[21]. Then (lines 2–5), we collect for each

[21] These require slightly different advice.

selected target `T` as context the name of the compiler it uses (`tool`) and the specific compiler invocation command (`C`). For this, we only need to look inside the command lists of the element targets of `Ts` and locate the sole command which invokes a compiler. We can actually reuse the tool mining query of Section 3.5.2 to locate the right compilers.

3.5.5.2 Composing Advice

For each combination of join point and context within `base`, we then compose a `before`- and `after`-advice, i.e. commands which should be invoked before or after every `C` respectively. The `before`-advice looks like this:

```
6 before_advice=
7  ["\n".join([C.replace(t,t+" -E -o ${<}-p"),
8              "mv ${<} ${<}-orig",
9              "mytool.sh ${<}-p ${<}"])
10        for (C,t,T) in base ]
```

The advice contains the shell commands which should be executed right before the compiler is invoked. The third command within the advice (line 9) contains the invocation of the preprocessor tool we want to integrate. This tool takes as input a source file which has been preprocessed by the C preprocessor (line 7). To not destruct the original source file, the command on line 8 backs up this file. The `join` on line 7 glues the three commands together, while the list comprehension (lines 7–10) generates the three commands for every element of `base`. The `after`-advice (not shown) is much simpler, as it only needs to rename the backup file to the original source file name after invoking the compiler.

3.5.5.3 Virtual and Physical Weaving

Finally, the MAKAO weaver should weave the advices before or after each command captured in `base`. To perform virtual weaving, the next Gython commands or used (for the `before`-advice):

```
10 cc_weaver=weaver("aspicere-cc",1)
11 cc_weaver.weave_before(
12               [T for (C,to,T) in base],
13               [C for (C,to,T) in base],
14               before_advice)
```

Virtual weaving acts as a simulation of the re-engineering. After execution of lines 11–14, the `commands`-dictionary contains the extra commands added by the advice. The user can inspect these results, manually or automatically (using verification), to check the re-engineering's impact, undo changes, and refine the advice.

	Visualisation	Querying	Filtering	Verification	Re-engineering
Implicit Dependencies (2.2.1.1)	X	X			
Syntax has Semantics (2.2.2.1)	X	X			
Advanced Language Features (2.2.2.1)		X			
Precedence of Variables (2.2.2.1)		X			
Debugging (2.2.2.1)	X	X		X	
Portability (2.2.2.2)		X			
Complexity (2.2.3.1)	±	±	±	±	
Traceability to Build Templates (2.2.3.1)		±			
Stakeholders (2.2.4.1)	X	X	X	X	
Overhead of Build Systems (2.2.4.1)	X	X	X		X
Recursive vs. Non-recursive "make" (2.2.4.2)	X	X	X		
Build System Changes (2.2.4.1)	X	X	X		X

Table 3.4: Evaluation of how MAKAO tackles goal T1.

If the re-engineering behaves as expected, physical weaving can be performed via a Perl script which has been generated as a side-effect of virtual weaving. This script contains the necessary logic to modify the actual build scripts, where needed, and to undo the changes if they are not satisfactory. If the advice is not written in terms of build or configuration parameters, one cannot distribute the woven build scripts, nor the Perl script, as they are not portable to other build platforms. Nevertheless, platform-specific customisation of a build system is still a powerful feature.

3.5.6 Evaluation

Table 3.4 gives an overview of how each of the five requirements implemented by MAKAO contributes to a solution for all identified build problems, i.e. the degree to which MAKAO fulfills goal T1. We can make a couple of observations from this table. First, querying is an indispensable activity for any build problem. It enables detailed access to build dependency graph data, command lists, build-time values of variables, defined preprocessor constants, etc. This knowledge is a prerequisite to understand which variable assignment has the highest precedence (Precedence of Variables), which build parameter requires platform-dependent in-

put (Portability), obtain context information for changes (Build System Changes), etc.

Second, MAKAO's re-engineering support is not needed in the majority of problems. As noted in Section 3.4.5, and observed by Robles [195] and Owens [186], AOP-based re-engineering of a build system especially makes sense for invasive changes scattered across the build layer. For less invasive changes, it makes more sense to re-engineer the build scripts in place, as problems often result from syntactical details (Syntax has Semantics) or the location of build logic (Precedence of Variables). Similar remarks can be made about verification, as only two problem categories require support for this.

Third, as soon as problems occur at a larger scale (the four general problems at the bottom), a combination of visualisation, querying and filtering is needed. As observed for the Linux kernel build graph in Figure 3.5a, a build dependency graph can visually be very complex. Querying is not fully suited for reducing the level of detail in a graph, especially because it requires imperative means to express the filtering intent. Instead, filtering support enables declarative specification of patterns or idioms of which abstraction should be made. An example of this has been discussed in Section 3.5.3, but the real value of filtering will become apparent in Chapter 4.

Fourth, there are a number of $\pm$-symbols. These signal that MAKAO is not entirely suited to deal with the GBS problems, because these are especially related to macro usage (autoconf) and generation of the build layer (automake). MAKAO only provides access to the instantiated build layer in which the autoconf macros are not retained and there is no reference back to the original build script templates. This is closely related to our decision to not focus on the configuration layer. MAKAO could be extended to support GBS's configuration layer by complementing the build dependency graph with information extracted from the configuration specifications. This would not require changes to MAKAO, only to the parser scripts.

To summarise, we have shown that the five requirements on which MAKAO is based are capable of tackling the build problems considered in Section 3.1.1. The ability of the requirements to deal with co-evolution of source code and the build system has not been validated yet. This is done in the next chapter in the context of the Linux kernel build system, and in later chapters for systems in which AOSD has been introduced in the source code.

3.6 Conclusion

In this chapter, we have distilled five requirements which together guarantee a solid reverse- and re-engineering environment for many typical build problems (goal T1) and for dealing with co-evolution of source code and the build system

(goal T2). Goal T2 has not been validated yet in this chapter. Together, the requirements enable visualisation of the build (sub)system's structure on various levels of abstraction, access to specific build data and automatic verification of common bad smells. For invasive build changes, aspects are promising. They have explicit support for build context, they clearly express the applicability conditions of the re-engineering task and they are able to modularise changes.

Because existing tool support does not satisfy the five requirements, we have designed and implemented a dedicated tool, i.e. MAKAO. It manipulates the build dependency graph of a concrete build, enhanced with various dynamic and static build data. Integration in a powerful graph manipulation framework, GUESS, enables interactive visualisation, scripted querying and re-engineering of the build system, and declarative filtering and verification of the build model. We have demonstrated how MAKAO works on a number of build problems which are representative for goal T1, in systems ranging from closed-source (Kava), closed/open-source (Quake 3 Arena) to fully open-source (Linux 2.6.16.18). Afterwards, we have evaluated how MAKAO, and hence the five requirements, contributes to a solution for all the build problems associated with goal T1.

In the next chapter, we use MAKAO to analyse the evolution of the Linux kernel build system, which requires detailed investigation of the build system at crucial points during the evolution. This analysis serves as validation of the conceptual roots of co-evolution postulated in Section 2.3.4 and of the ability of MAKAO to assist in understanding build systems in general, and symptoms of co-evolution of source code and the build system in particular. The latter validates the support for understanding co-evolution of source code and the build system (goal T2 from Section 3.1) in non-AOSD systems. Later on, in Chapter 6 to Chapter 10, MAKAO is validated on its ability to understand and manage co-evolution of source code and the build system (goal T2) in the presence of AOP technology.

"Correctness trumps efficiency."

I just hate hearing stuff like "if we leave this dependency out, the Makefiles will run faster, and the user will just have to remember to run 'make dep' after they change the configuration". That is so Microsoft!

Correctness comes first. Then maintainability. Then speed.

Michael Elizabeth Chastain (Linux 2.4 build maintainer)

4

Experimental Evidence for Co-evolution of Source Code and the Build System

THIS chapter[1] uses MAKAO to analyse the evolution of the Linux kernel build system from the kernel's inception up until present day to find experimental evidence of the co-evolution phenomenon between source code and the build system. This evidence serves as validation of the four roots of co-evolution postulated in Chapter 2 and as validation of MAKAO's support for identifying and understanding symptoms of co-evolution (goal T2). Our observations are based on measurements of the number of lines of code, number of files, characteristics of the build dependency graphs and manual browsing through the build scripts. Finding the right balance between obtaining a fast, correct build system and migrating in a stepwise fashion turns out to be the general theme throughout the evolution of the Linux build system. Hence, the kernel maintainers implicitly acknowledge the important role of the build system.

This chapter first (Section 4.1) introduces the rationale behind the analysis of the evolution of the Linux kernel build system, before explaining the setup and the scope of the case study (Section 4.2). The analysis is divided in three parts. First (Section 4.3), we investigate the evolution of the physical relation between source code and the build system by means of the number of lines of code and the number of files. Second (Section 4.4), the evolution of the internal complexity of the kernel

[1] This chapter is based on [5].

build system is examined via a number of simple, but effective metrics. Third, Section 4.5 gives a detailed, and at times very technical, account of maintenance activities and their consequences on the build system. Section 4.6 summarises the experimental evidence of the co-evolution phenomenon between source code and the build system, which validates the four roots of co-evolution and MAKAO. Similarly, Section 4.7 validates how MAKAO achieves Goal T2, i.e. tool support for understanding and managing co-evolution of source code and the build system (see Section 3.1.2).

4.1 Rationale behind the Linux Kernel Case Study

This section introduces the case study, its goals and why the Linux kernel build system has been chosen for our analysis. Linux is a widely used and acclaimed open source operating system, used both by enthusiasts as well as in industrial settings. It started out in 1991, when Linus Torvalds sent an email to the Minix newsgroup, stating that he had developed a free operating system he wanted to share with everyone. It consisted of a monolithic kernel, in which device drivers were hardcoded (e.g. Finnish keyboard), with user space programs ported over from Minix. In a little over fifteen years, Linux has grown from this one-man hobby project into what is probably the largest open-source project on the planet, praised for its portability across computer configurations. The Linux kernel features a custom build system based on GNU Make, Bash scripts and various small tools.

Linux has been the target of many studies. The one which is of most relevance to this dissertation has been performed by Godfrey et al. [99]. They have studied the kernel source code from the perspective of software evolution, both in its entirety as well as separate subsystems in isolation. Contrary to Lehman's laws [148], the kernel exhibits a superlinear growth in size. This means that strangely enough the growing complexity does not temper the kernel's evolution. Independent evolution of kernel drivers and code duplication among code for different architectures explains some of their findings, but Godfrey et al. consider the open source development model as the biggest reason for this strange phenomenon, although other open source systems do not exhibit this evolutionary behaviour [39]. Ramil et al. [91] later have observed that the results of published open source experiments are less uniform than in the proprietary case and that evolution in open source is harder to predict in general. Godfrey et al. explicitly have excluded the build system from their investigation. Another relevant study has been made by Bowman et al. [29]. They deduce the system's concrete architecture from the Linux implementation and a conceptual idea of the architecture. To detect and populate the subsystems of the latter conceptual view, they have manually inspected the source code instead of mining the build system for architectural information.

We have studied the existence of co-evolution of source code and the build system in the Linux kernel build system by analysing the evolution of its build system. More in particular, we have investigated most of the early kernel releases and all subsequent major releases, i.e. we have opted for a discrete-time perspective [39]. It is important to note that we explicitly have not examined the kernel source code, nor the changes or log messages in the source code repository, for a number of reasons. First, given the scale of the Linux kernel and the experiment, analysis of the effects of fine-grained source code changes on the build system and vice versa is unmanageable. Focusing only on changes in two consecutive minor releases is too limited, whereas source code changes between two major ones are too numerous to process and interpret. Picking out one subsystem is more feasible, but to avoid that the selected component is not representative (both from the perspective of the source code and build system), multiple ones should be examined. However, the evolution of the global system is not simply the sum of all its components [99, 39]. Second, a coarser-grained perspective of the source code can be deduced from physical characteristics like the directory structure [39] or the number of files. These are easy to measure and convey important information, but do not require analysis of the source code. Third, the availability of extensive email discussions between maintainers of the Linux build system, online documents which describe design discussions of the build system and the documentation of the build system provide a thorough account of the rationale of changes to the build system. Based on these three arguments, we claim that the analysis of the evolution of the Linux kernel build system suffices to detect the existence of co-evolution of source code and the build system. Our observations acknowledge this.

These are our research hypotheses:

1. Co-evolution of source code and the build system is an important factor in the evolution of the Linux kernel build system.

2. The symptoms of this co-evolution can be explained in terms of the four roots of co-evolution of Chapter 2.

3. MAKAO is able to support developers in detecting and understanding the symptoms of co-evolution.

The next section discusses how we intend to validate these hypotheses, but first we highlight our choice for the Linux kernel build system as subject of our analysis. Like any open source system [39], almost the whole development history can be freely accessed online. This encompasses all releases, the source code, documentation, mailing list discussions, dedicated web sites, etc. Second, because of its popularity in various application domains, the Linux kernel lives under constant pressure of changing requirements, feature requests, bug fixes, optimisations, etc. As such, we would suspect that co-evolution induces the same intensity of changes

onto the build system. There is an extra catch however, in the sense that any build error potentially reaches thousands of developers. Due to the diversity in development environments, changes to the build system need to be robust to platform dependencies. This makes the Linux kernel a good choice for validating our hypotheses.

There are also a couple of reasons why we should not have chosen the Linux kernel. First and foremost, the aberrant source code evolution discovered by Godfrey et al. [99] does not align with Lehman's findings on proprietary systems [148] and is also not representative for all open source systems [91]. However, in our case we are not interested in the absolute evolution of e.g. number of lines of code or number of files. We focus on the relation between the evolution of source code and the build system, whatever their respective pattern looks like. Second, the fact that we have not investigated a closed source system can also be interpreted as a drawback. This type of system is developed by companies with a limited number of analysts, designers, developers, etc. As their number does not scale with the number of feature requests or bug fixes, the evolutionary behaviour of such systems could indeed be more akin to Lehman's laws. It is unknown whether the build system would behave similar in those circumstances. Again, as we are interested in the relative evolution behaviour of source code and build system, analysis of open source software does not invalidate our research hypotheses.

Ramil et al. [91] have identified a number of threats of validity attributed to studies of open source systems. We briefly discuss how we deal with these threats:

incomplete or erroneous data We combine data from multiple sources, i.e. build scripts, documentation, online email discussions and MAKAO.

biased samples To avoid this, we have examined most of the early releases, as changes to the build system and the source code are easier to make, and all subsequent major releases. To ensure that we have not missed important evolution steps in between the major releases, we have taken random samples of intermediate releases.

errors in data extraction The tools we have used for counting the number of lines of code and the number of files were existing programs which have been tested extensively in practice. MAKAO has been applied on the three build systems described in Section 3.5 before we have examined the evolution of the Linux kernel build system. Hence, we are confident that it extracts data in a correct way.

representativeness of the investigated system for the case This is discussed above.

granularity of the investigated components Idem.

Date	Version	Date	Version
September 17,1991	0.01	March 13, 1994	1.0
December 8, 1991	0.11	March 7, 1995	1.2.0
May 25, 1992	0.96a	July 3, 1996	2.0.1
July 5, 1992	0.96c	January 26, 1999	2.2.0
July 18, 1993	0.99.11	January 4, 2001	2.4.0
September 19, 1993	0.99.13	December 18, 2003	2.6.0
February 3, 1994	0.99.15	June 11, 2007	2.6.21.5

Table 4.1: Chronological overview of the Linux versions we have investigated.

undocumented development periods There is not much information about the early releases, but there are more samples of this period and the scale of the system is easier to manage.

unknown variables The biggest unknown variable is the source code, for the reasons outlined above.

lack of characterisation of the investigated system The Linux system is an interesting system to study, as previous research suggests [99, 29, 91]. As discussed above, current research has not yet found proof whether or not the Linux kernel is representative for open source systems in general.

This section has presented the research hypotheses we intend to validate and has motivated our choice for the Linux kernel build system to do this. In the next section, we explain the setup and scope of the case study. Afterwards, we present the analysis results and report on the validation of our research hypotheses.

4.2 Setup of the Linux Kernel Case Study

To study the evolution of the Linux kernel build system, we have looked at most of the pre-1.0 releases of Linux, as well as the major stable post-1.0 releases (up to the 2.6 series). Table 4.1 gives an overview of the processed versions, which span fifteen years of real-world development time. The distribution of systems seems to be skewed towards the earliest versions (1991–1995), but as Linux was still a young system back then, it was much easier to make drastic changes than later on. Indeed, samples in the later kernel series revealed no drastic build changes within a series, except for the 2.6 line. This is due to the changed development model, i.e. there is no unstable kernel branch anymore in parallel with the stable one. The next three sections describe how we have processed the selected Linux kernel releases.

4.2.1 Measuring SLOC and Number of Files

To each of the kernels of Table 4.1 we have applied David A. Wheeler's SLOC-Count tool[2] to calculate the physical, uncommented SLOC (Source Lines Of Code) of source code (.c, .cpp, etc.), build scripts ("Makefile", "Kbuild", etc.), configuration data ("config.in", "Kconfig", etc.) and support build files (.sh, .awk, etc.). We have also counted the number of files in these categories. Both metrics are simple and general, but are representative for the size and modularity of both source code and the build system. We have compared our findings with those of Godfrey et al. [99].

4.2.2 Calculating Metrics for the Internal Build Complexity

To get more specific information about the nature of the build system itself, we have compiled[3] each of the kernels based on an initial configuration which we have reused and enhanced with new kernel features as needed throughout the measurements. The first kernels did not have configuration support, as this was only added in version 0.99.11. From this version on, we have used a run-off-the-mill configuration for compilation, i.e. neither a full-fledged nor a sparse one. This enabled us to measure builds as encountered by the average developer or end user. There were two possibilities to carry on:

- reuse the same configuration throughout all subsequent kernels
- gradually adapt the configuration to adhere to the changing requirements in kernel functionality

This dilemma actually applies to any product line. At first sight, the first option is preferable. However, this defies a big deal of the forces behind system evolution, i.e. changing requirements and feature requests. As a concrete example, networking and sound support were only added in the 0.99.x series. Both are indispensable in current systems and hence are always compiled in nowadays. This suggests that the second option is the way to go. Of course, careful consideration is required to limit the amount of noise. Only mainstream configuration choices should be made. We are lucky in this regard, as the kernel's configuration system allows to import configurations from older systems and suggests default configuration actions for any new features. Only a minority of configuration options are suggested to be part of the kernel, the majority is added as modules[4]. Hence, by focusing our detailed analysis of Section 4.5 on the kernel and the default configuration suggestions, we

[2] `http://www.dwheeler.com/sloccount/`

[3] For the sake of completeness, all experiments have been made on an Intel Pentium 4 (3.4 GHz) with 2GB of RAM.

[4] These are optional components like hardware drivers which can be easily (un)loaded by a kernel, even dynamically, without requiring recompilation of the kernel.

are able to analyse representative configurations of the Linux kernel. The focus on the kernel itself is not a big restriction, as modules are by themselves independent from each other and mostly correspond to drivers and other code produced by external stakeholders.

Each kernel compilation yields multiple build traces, one for each build phase, i.e. the compilation of the kernel itself, the compilation of modules and usually also the extraction of source code dependencies. As we want to find out how the global build system evolves, the traces we look at correspond to full builds. We have used MAKAO to obtain the corresponding build dependency graphs, and to calculate complexity metrics on the number of targets and dependencies. We measure these simple graph characteristics because they are easy to grasp, but at the same time convey a lot of information about the internal complexity of a build system. Graph nodes represent build targets, hence if the number of edges remains stable, the appearance of more targets denotes that more files (physical or virtual) have been checked during the build, i.e. the build takes longer. If the number of targets remains constant, but the number of edges (build dependencies) increases, the build has become more complex to obtain the same build products. This identifies redundant build logic, introduction of new tools, integration of new build idioms, etc. Hence, the metrics give important indications on the evolution of the internal complexity of a particular configuration of the build system.

To validate our findings, we have investigated the available (online) documentation, literature [167], email archives of the Linux build system and comments in the makefiles. This has provided us with independent sources of evolution data for the Linux kernel build system, which we have used to compare our metrics with. As we will see, there is a high correlation between the metrics and the actual kernel build evolution.

4.2.3 Detailed Study of Crucial Evolution Steps

As mentioned in the previous section, we have found indications of maintenance activities in particular kernel versions based on measurements of the number of lines of code, files, targets and dependencies. To verify whether these predictions are correct, each dependency graph has been loaded into MAKAO (Chapter 3) to investigate them in detail by means of visualisation, querying and filtering. This detailed data is correlated with information from the actual build scripts (makefiles), mailing list discussions and other online development information. Especially the transition from the 2.4 to the 2.6 kernel series turns out to be fundamental, as big changes in the build dependency graph can be observed. For this reason, we spend a considerable part of this chapter on unraveling the 2.6.0 kernel build system to detect idioms and patterns in it and to deduce their relation to co-evolution of source code and the build system.

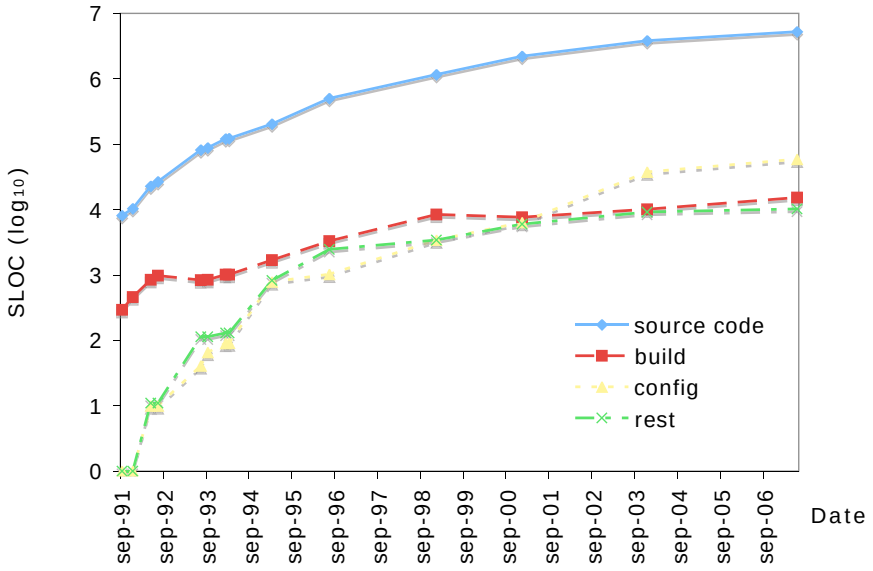


Figure 4.1: Evolution of the number of non-comment, non-whitespace lines of source code (SLOC), build and configuration scripts in the Linux kernel build system.

Note that this detailed study of build dependency graphs does not consider the configuration layer of the Linux kernel build system, as a dependency graph in MAKAO corresponds to a build of one particular configuration. However, during our examination of the kernel build history, we have found some evidence of co-evolution of source code and the build system in this area. We briefly present this in Section 4.5.1.

The next three sections report on the analysis results we have derived using the approach discussed in this section.

4.3 Observation 1: the Build System Evolves with the Source Code

This section presents and discusses the measurements obtained with SLOCCount and a count of the number of files. Figure 4.1 shows the growth over time (in logarithmic scale) of the number of non-comment and non-blank lines of source code and build system files. The number of files is presented in Figure 4.2. Both consider the following four groups of files[5]: (1) the source code files, (2) the actual

[5]Note that measurement of SLOC and also number of files occurs statically, which means that it is independent of the chosen build configuration.

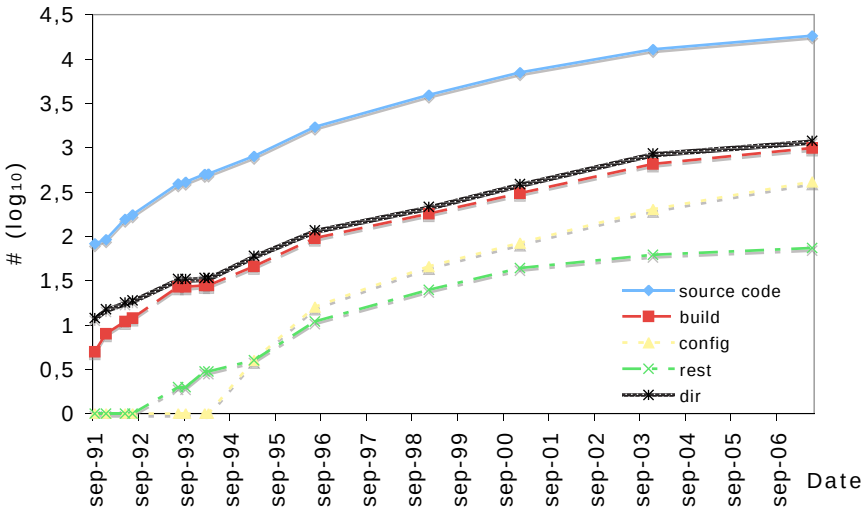


Figure 4.2: Evolution of the number of source code files, build and configuration scripts in the Linux kernel build system.

makefiles (`build`), (3) the configuration files which drive the selection process of what should get built (`config`), and (4) other tools and scripts which assist the actual build process[6] (`rest`). Figure 4.2 also shows a fifth group of measurements, i.e. the number of directories[7] within the source code distribution. This has been added as a means to better assess the modularity of the build and configuration scripts.

A first thing to note is the sheer order of magnitude exhibited by the build system (Figure 4.1 and Figure 4.2). Our measurements confirm the claim made in [99] about the source code's super-linear evolution in SLOC and in file count, but suggest similar findings for the build system (on a lower scale). The build layer has grown from 293 SLOC in 5 build scripts to 15351 SLOC in 990 files (2.6.21.5). As for the configuration layer and build support, this is even more impressive as it has evolved from nothing to 58801 SLOC in 415 files and 10215 SLOC in 74 files respectively. By way of reference, the source code has exploded from 8102 SLOC in 83 files to 5274927 SLOC in 18337 files. These figures suggest a very high complexity, not only in the build system and source code themselves, but also on the scale of induced changes.

Figure 4.1 and Figure 4.2 support RC1 (modular reasoning vs. build units) and

[6] These are build-time scripts and programs to extract symbol tables, install kernel components, etc. As many of them have to be compiled at the beginning of the build, the Linux build system conforms to a simple version of the "Code Robot" architectural style [228].

[7] For this, we excluded the "Documentation" directory introduced in the 2.x series, as it merely contains documentation about the kernel and its build system.

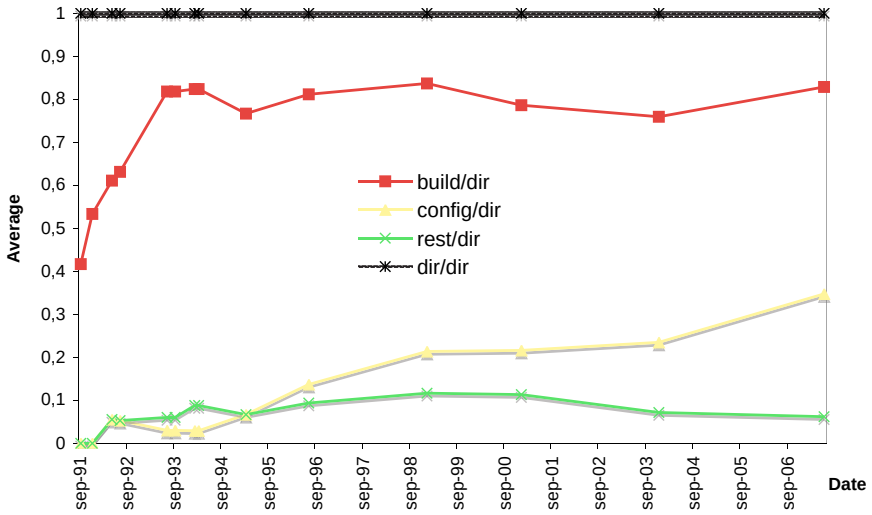


Figure 4.3: Evolution of the average number of build and configuration scripts per directory.

RC2 (programming-in-the-large vs. build dependencies). A first indication is that the build system grows superlinear, similar to the source code. Over the course of fifteen years, the SLOC of the makefiles has grown by a factor of about 52, although it has slowed down in the more recent versions. The file count on the other hand keeps on increasing superlinearly. The average number of build system files per directory seems more or less constant on Figure 4.2. If we explicitly plot this ratio (Figure 4.3), we can indeed see that from March 1994 on (kernel 1.0), each directory contains on average near 0.8 makefiles. The auxiliary tools and scripts on the other hand, are all located under one directory ("scripts") and a couple of subdirectories, hence the low average. As the directory structure of the source code corresponds to the high-level structure [40] of the Linux kernel, the makefiles seem to co-evolve with the kernel architecture. This is a second indication of co-evolution of the build system with the source code. Third, the build system is modularised such that each new component, i.e. subdirectory, gets its own makefile. This is acknowledged by the fact that the Linux build system is known to use a "recursive make" (Section 2.2.4.2). However, not each subsystem has its own configuration specification, as the ratio of specifications per directory is lower than for the makefiles, although within the 2.6 kernel the average number has jumped from 0.2 to over 0.3 because of changes in the configuration specification language (more on this later). These three observations give evidence of RC1 and RC2, i.e. source code components have corresponding build subsystems and the

source code architecture and build system evolve together.

What is striking from Figure 4.1 is the amount of work which has been put into the configuration and support system. These have grown from nothing in the first version[8] to almost 60K and 10K lines respectively, the former even getting ahead of the core build files. The first version of the configuration layer in the 0.99.1x series consists of a simple Bash script which takes a "config.in" configuration specification and generates a ".config" file for usage during the build and an "autoconf.h" for usage inside the source code (cf. GBS in Section 2.1.3). This script has been enhanced to support configurations of older kernel releases and the `modules` phase in the 1.2.0 kernel. In the meantime, nearly all the techniques for passing configuration choices to the source code which we have seen in the context of autoconf (Section 2.1.3.1) are used in the Linux kernel build system. In the 2.0.0 kernel, the central "config.in" has been distributed across all source directories and various graphical and textual configuration front ends have been added (which partially explains the increase in SLOC for `rest`). The introduction of the configuration front ends shows how the build system has had to react to the extreme configurability provided by the source code. The Linux kernel actually is a product line [55] which can be tweaked to the particular hardware it is going to control, and hence requires a sophisticated configuration system. The evolution of the configuration layer clearly supports RC4.

To summarise, the simple metrics of this section have shown that source code and the build system structurally follow a similar growth pattern (RC2), the build scripts (and configuration scripts to a lesser extent) are spread across the source code structure (RC1) and that the build system's configuration layer has seen a steep evolution to cope with the source code configurability (RC4). These are important symptoms of co-evolution. The next section studies metrics of the build dependency graphs of the investigated kernel releases.

4.4 Observation 2: Build System Complexity Fluctuates

In order to assess the evolution of the complexity of the build system, we have calculated a number of metrics of the build dependency graphs of the examined kernels. This means that we do not have data on the configuration specification. The meaning of the metrics especially becomes clear when combined with analysis of the build dependency graphs, hence this section only shows that maintenance has been performed to reduce build complexity, whereas the next section goes into more detail about the semantics of the maintenance.

[8] Technically, the measurements for `config` and `rest` around '91–'92 have zero as value, which should be mapped to minus infinity on Figure 4.1. For practical reasons, this has been approximated by using a logarithmic value of zero on the graph, corresponding in fact to 1 SLOC.

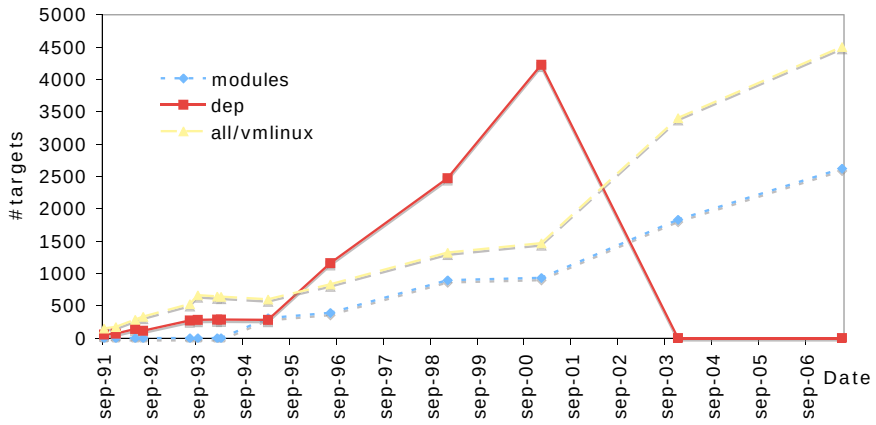


Figure 4.4: Evolution of the number of build targets during the compilation of the Linux kernel.

Figure 4.4 presents, for all analysed kernel releases, the growth over time of the number of build targets which appear in each build phase. As mentioned before, the Linux build process is divided into a number of phases, of which we will consider the most important ones. The kernel image is either built by a phase named `all` or `vmlinux` (from version 2.6.0 in 2003 on), while modules are built by the `modules` phase. Extraction of source code dependencies occurs during the `dep`-phase. During this phase, the `#include`-dependencies found within .c files are scanned and corresponding makefile snippets are generated for use in the `all` or `modules` phases. As MAKAO works with dynamic traces of a build (Section 3.4.2), Figure 4.4 shows the number of targets checked or built by the build process during a concrete run of each of the three phases[9].

Overall, Figure 4.4 reflects the point made in the previous section: the build system grows, not only in lines of code, but also in the number of tasks it attempts to complete. From the 2.6 kernel in 2003 on, the `dep`-phase disappears and is subsumed by the other two phases. Note that, contrary to the measurements in the previous section, the examined build traces correspond to one particular build configuration, i.e. they only take into account the contribution of selected source code and build files, not the whole source tree. Second, as we have indicated in Section 4.2.2, we have started from a given configuration and gradually enhanced it with new features. Hence, it seems only natural that the number of build targets increases. However, this noise does not invalidate our measurements, as the measurements of the number of dependencies will show.

Figure 4.5 and Figure 4.6 relate the growth over time of the number of explicit

[9]Every kernel has been built from scratch, i.e. we have not measured incremental builds.

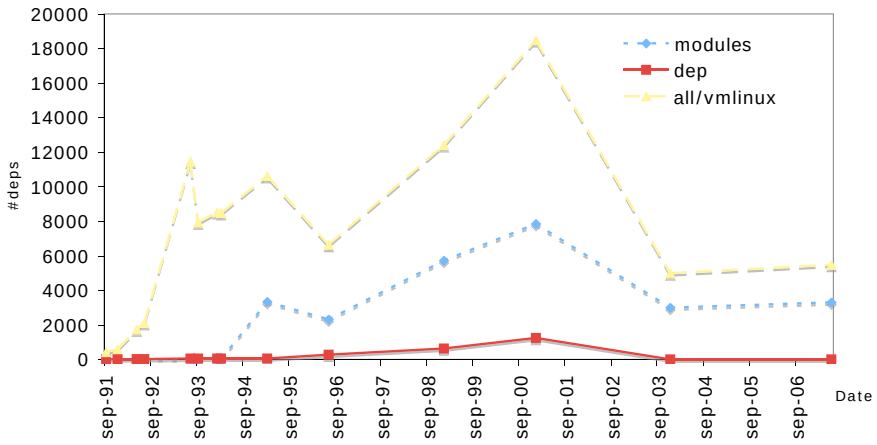


Figure 4.5: Evolution of the number of explicit dependencies during the compilation of the Linux kernel.

and implicit[10] build dependencies respectively, for each of the three build phases. The turbulent course of Figure 4.5 culminates in a huge growth up to September 2000 (kernel 2.4.0), followed by a serious dip. We also notice that eventually the number of dependencies rises again, albeit at a slower pace. Typically, the number of dependencies grows when more build targets appear in a build, redundant checks are made, artificial dependencies have been added, etc. However, there are also periods when the number of dependencies decreases. If we combine this with the observation that the number of targets only grows (partly because the compiled configuration is extended), this means that the build system complexity fluctuates a lot. Because every new target adds at least one extra edge to the build dependency graph, the reduction in complexity can only be the product of human intervention, i.e. maintenance operations in the build system. Deeper investigation is needed to verify the true nature of these changes. This is done in the next section.

Figure 4.6 shows that there is also a steady growth in the number of implicit dependencies. This means that the number of relationships the build system knows nothing about is on the rise. This is not only problematic when trying to understand the build system, it also constitutes a potential source of build errors, and (at best) may lead to suboptimal builds. Luckily, most of the implicit dependencies originate from temporary files created during the `dep` phase, i.e. files which contain the dependency makefile snippets extracted from the source code's `#include`-relations. Extraction of dependencies only seems to do a depth-first iteration of the directory structure. This can be deduced from the resemblance between the

[10] As seen in 2.2.1.1, implicit dependencies are relationships which are not explicitly declared as makefile rule dependencies, but rather are buried inside a rule's command list.

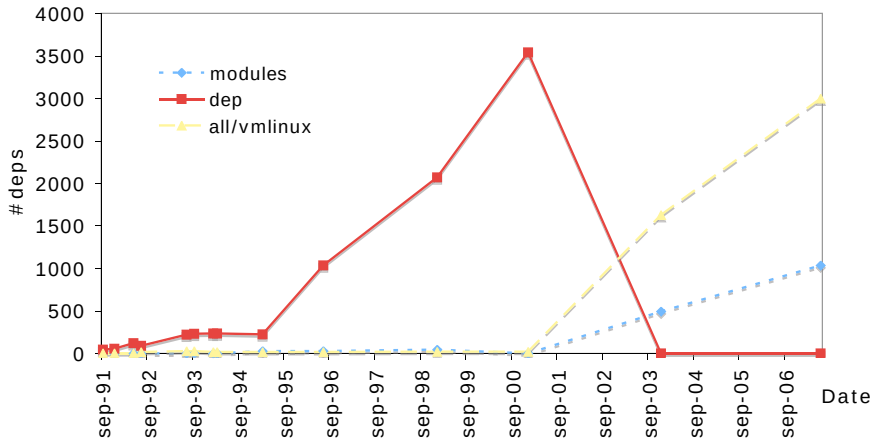


Figure 4.6: Evolution of the number of implicit dependencies during the compilation of the Linux kernel.

corresponding charts in Figure 4.4 and Figure 4.6, which means that the number of implicit targets (edges) closely follows the number of targets (nodes), i.e. the graph of the `dep` phase resembles a spanning tree (all targets are reached only once). This has been acknowledged by investigating the build dependency graphs. The fact that the `modules` and `vmlinux` phases only start to exhibit implicit dependencies after the `dep` phase has been merged with them suggests that these dependencies are harmless as well. This merger is also largely responsible for the steep growth of the number of targets of the `vmlinux` phase in Figure 4.4, but not for the `modules` phase. This is probably due to the addition of extra drivers to the build configuration.

This section has presented the measured complexity metrics for the build systems of each of the investigated kernels. The number of build targets keeps on growing. However, this is not only due to the expanded build configuration as the fluctuating course of the number of explicit dependencies shows. There are periods where less dependencies (edges) appear for a higher number of targets (nodes), i.e. complexity decreases. The opposite can also be observed. Hence, it is clear that maintenance has been performed to reduce the build complexity. The next section links these numbers to actual maintenance activities, which also give us new indications of co-evolution of source code and the build system.

4.5 Observation 3: Co-evolution as Driver of Build Evolution

Having observed that the build system evolves and that maintenance is done to reduce the growing complexity, we now examine the specific maintenance actions which have been performed by the Linux build developers. These activities clearly illustrate the four roots of co-evolution. First, we briefly consider changes in the configuration layer. Then, maintenance activities up until the 2.4.X kernel series are presented. Finally, we take a very detailed look at the changes between the 2.4.0 and 2.6.0 kernel build system. Although the latter section is very technical, it illustrates many different facets of co-evolution of source code and the build system, and shows how MAKAO is capable of supporting developers to understand a build system.

4.5.1 Configuration Layer under Pressure

We have not directly analysed the configuration scripts of the Linux kernel build system. Nevertheless, the measurements of Section 4.3, casual browsing of the scripts and various email discussions have made it clear that there has been a very apparent evolution in the configuration layer. Initially, there was no possibility at all to configure the kernel, but since the 2.6 series the configuration system has surpassed even the build scripts in SLOC.

There are a number of interesting facets of this evolution. The graphical front ends which we have discussed before take as input configuration specifications which contain the various configuration options. These options are hierarchically ordered in groups and have constraints between them. Each option contains a name, explanation and default configuration decision (to select something as part of the kernel, as a module or to exclude it from the build). These constraints have to be updated manually when the source code evolves, as each module's (implicit) expectations and provisions determine the feasible build configurations. Similarly, the applied configuration mechanisms (the same as those provided by autoconf in Section 2.1.3.1) require updates to the build system to cope with changes to the source code. This clearly represents evidence for RC4.

The specification languages for the configuration and build scripts correspond to high-level domain-specific languages which have been built on top of low-level technology like Bash scripts and GNU Make. These high-level languages hide low-level details for developers. In personal communication, Linux 2.6 build maintainer Sam Ravnborg has coined the phrase "simple syntax for simple things" for this. Whereas the build system engine ("how") may significantly be optimised, maintained and tweaked, the Linux build users are shielded from this via domain-specific build and configuration languages ("what"). Although these languages

```
1 obj-y :=
2 obj-m :=
3 obj- :=
4 obj-$(CONFIG_SOUND) += soundcore.o
```

Figure 4.7: Format of list-style build scripts in the 2.4.0 kernel build scripts.

have seen a number of different incarnations, they mostly remain stable. This is an important countermeasure for protecting developers against the frequent maintenance activities which are performed. As most of these activities have to do with co-evolution of source and the build system, the domain-specific languages are actually a countermeasure to deal with the co-evolution.

Although we have not investigated the configuration layer in detail, This section has identified important configuration characteristics which are related to RC4. The next section briefly reports on specific maintenance actions on the build system up to the 2.4.X kernel series.

4.5.2 Evolution until the Linux 2.4 Series

Since its inception, Linux sports a "recursive make"-based build system. However, its particular face has gone through many revisions for a variety of reasons. Especially the evolution to the 2.6.0 kernel has caused important changes (see the next section), but there have been many changes before.

The decrease of fourteen percent in SLOC during the pre-1.0 era (1992-1993) on Figure 4.1 indicates that effort has been spent to mitigate build complexity. This effort corresponds to a new scheme by which the recursive build traverses through the source code directories, replacing the naive implementation of "recursive make" in use until then. Each makefile defines a variable which contains the subdirectories to traverse over and a specific build rule ("linuxsubdirs") which specifies the traversal. These rules depend on a phony target[11] named "dummy" to make sure that each build always executes a rule's command list, i.e. the subdirectories are always traversed. However, this is only the beginning of a long list of changes to the "recursive make" infrastructure. Their goal is to assist developers in adding new drivers or other components by making it as easy as possible to integrate their makefiles into the build (RC1).

Figure 4.5 hints at many of these changes. Between 1.2.0 and 2.0.0 (1995-1996), common build logic has been extracted into a shared build script called "Rules.make", resulting in a 40% reduction of explicit dependencies (while the number of targets has increased). Between 2.2.0 and 2.4.0 (1999-2000), the de-

[11] A phony target is a target which has an associated time stamp which lies in the future. Hence, if a dependee is declared as phony, the dependent always is always out-of-date and needs to be rebuilt by executing the rule's command list. More on this in 4.5.3.2.

crease of nine percent in SLOC can be attributed to a massive rewrite of the build scripts and "Rules.make" in a more concise "list-style" manner [155]. This is a higher-level way to specify build dependencies, as shown on Figure 4.7. Depending on the fact whether sound support (CONFIG_SOUND) is chosen as a built-in feature ("y"), module ("m") or left out (""), either the "obj-y", "obj-m" or "obj-" variable is assigned the object file soundcore.o. Developers do not need to specify any build rules anymore. The central "Rules.make" rule base of the 2.4.0 kernel has been completely rewritten in terms of the "obj-y", "obj-m" and "obj-" variables and just needs to iterate through the list of names they contain. Hence, the build logic boils down to list processing. The list-style makefiles actually form a kind of high-level specification language which hides build rules for the developers. This forms additional evidence of the validity of RC1, i.e. to facilitate dedicated build components for each source code component.

A second area of important build evolution steps, indirectly related to the three "recursive make" changes, is the `dep` phase. In the earliest kernel releases, automatic detection of source code (header file) dependencies within the build system was not that sophisticated. Releases 1.2.0, 2.0.0 and 2.2.0 each tried to make dependency management more correct without sacrificing (build) speed. New dependency extraction scripts, recursion schemes, decomposition of the generated configuration header files, etc. tried to achieve this. Eventually, as people still encountered many nuisances, the separate `dep` phase has completely disappeared in the 2.6 kernel and has been integrated into each build phase (see 4.5.3.2). The trade-off between accurate source code dependency information versus build speed is very apparent in this area. This was also coined as "correctness trumps efficiency" by Michael Elizabeth Chastain, the Linux build maintainer [155]. Less dependencies results in less things to check during the build, but ignores certain file changes and hence may lead to inconsistent builds. These findings clearly relate to RC2 and RC3, i.e. the build DAG reflects the dependencies in the source code and compromises are made to speed up the build by leaving out certain dependencies.

This section has presented maintenance actions undertaken before the 2.6.x kernel series. These actions have provided evidence of RC1, RC2 and RC3. The next sections considers the 2.6.0 kernel build system, which is a complete rewrite of the Linux kernel build system.

4.5.3 Towards the Linux 2.6 Kernel

This section elaborates on the huge evolution step the Linux build has gone through between Linux 2.4.0 and 2.6.0 (2001-2003). This step immediately caught our attention because the separate dependency extraction step has vanished and is subsumed in the other phases, the number of explicit dependencies has dropped dras-

tically to one third of the 2.4.0 level, but the number of targets in the `vmlinux` build has increased with a factor of 2.3. Hence, we have compared the build dependency graphs of the 2.4.0 and 2.6.0 kernel builds, and have investigated the available developer documentation and email discussions. The resulting findings and their relation to co-evolution of source code and the build system are discussed in this section. Note that this section is rather long, detailed and at times very technical. However, it clearly illustrates many symptoms of co-evolution of source code and the build system.

The overhaul of the 2.6 kernel build system dates back to 2000, when two people independently proposed substitutes for both the configuration and the build layer. CML2[12] tried to replace the informally defined configuration layer by a domain-specific language implemented on top of a custom rule-engine written in Python, and also tried to address the tight dependency on conditional compilation and building. Kbuild 2.5[13] was a rewrite of the Linux build layer as a "non-recursive make" (see Section 2.2.4.2). Both approaches represented a big improvement over the then current build system. Nevertheless, the introduction of non-conventional (for kernel development) technologies like Python, and the absence of incremental migration strategies — in contrast to the list-style makefiles of Section 4.5.2 — have precluded both independent initiatives from being accepted. This is a very important point, as it implicitly acknowledges the existence of co-evolution of source code and the build system. Linus Torvalds ignored the technological improvements to the build system to safeguard its correctness. For one, not all features of the old build system were reimplemented (module versioning e.g.) and if they were, some were not tested thoroughly. Second, developers had to learn a new way of working and adapting all components. Third, although the new system could run in parallel with the old one, there was no incremental way of migrating from one to the other. Hence, the risk of bringing daily development to a halt, even temporarily, caused the two proposals to be blocked. This contrasts with the KDE case of Section 2.3.3.1, where developers unanimously decided to migrate to another build system.

Instead, the existing 2.4 build system has been incrementally upgraded to the 2.6 kernel by squeezing every possible bit of performance out of it. Roman Zippel has designed Kconfig (initially called "LinuxKernelConf")[14], i.e. a standard configuration language specification to which all front ends had to adhere. The existing "config.in" files have been rewritten and renamed to "Kconfig", which explains the slight increase in SLOC on Figure 4.1. The build scripts have been refactored by Kai Germaschewski. He managed to incorporate a number of Kbuild 2.5's proposed features on top of the existing build infrastructure, while avoiding

[12] `http://lwn.net/2001/features/KernelSummit/`
[13] `http://sourceforge.net/projects/kbuild/`
[14] `http://www.xs4all.nl/âĹijzippel/lc/`

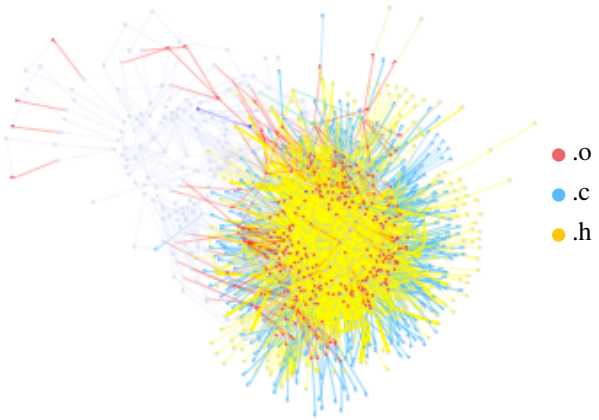


Figure 4.8: Build phase `all` of the Linux 2.4.0 build process (with header file targets).

a non-recursive "make". In the 2.6 build layer, "Rules.make" has been made much more sophisticated (and renamed) and all "make" subprocesses are now invoked with the top directory as working directory. Nevertheless, build speed is lower than in the (blocked) Kbuild 2.5 system. Hence, the original build system has been largely rewritten (in small increments) instead of moving to a completely new infrastructure.

The following two sections look in depth at the changes introduced by Keith Owens' Kbuild 2.5 and at the eventual 2.6 build system written by Kai Germaschewski.

4.5.3.1 Kbuild 2.5 Eliminates Recursive Make

This section discusses the design of the proposed Kbuild 2.5 build layer, as this exposes the fundamental problems of the 2.4 kernel build system and the technically advanced solutions for them. This knowledge helps in understanding the changes which eventually were made to obtain the 2.6 kernel build system.

Keith Owens' design of Kbuild 2.5 is based on a number of observations made from the 2.4 kernel and feedback solicited from other build developers [156]:

- Build execution flow is very hard to follow because of the hundreds of build scripts.
- The "recursive make" symptoms mentioned in Section 2.2.4.2 surface, including the scattered, unenforced specification of recursion logic.
- Build scripts in subdirectories of a "recursive make" system do not function in isolation, because they need various environment variables to be set by higher makefiles in the directory structure.

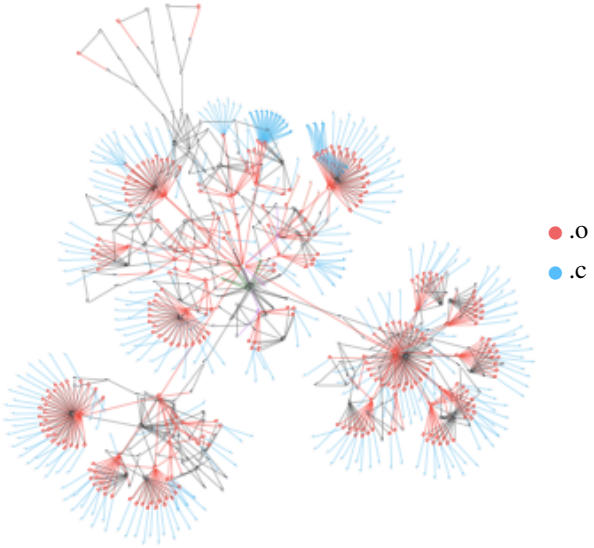


Figure 4.9: Build phase `all` of the Linux 2.4.0 build process (without header file targets).

- There are lots of important implicit dependencies[15], e.g. between a Linux module and the kernel modules on which it relies.

At the same time, dependency extraction from the source code was said to be "fundamentally broken" in the 2.4 kernel[16]. First, its existence as a separate build phase means that everyone who makes changes to the build system needs to remember to run the "dep" phase before recompiling their system. Second, header files generated during the build are not taken into account at all. Third, the script used to mine the .c files for #include-statements only knows about two header file locations, i.e. the current directory and a global one. Hence, custom header file directories used by some subsystems are simply ignored. Fourth, no difference is made between source files selected in the build configuration and unimportant ones, such that some build errors are actually false positives caused by files which do not need to be compiled. Finally, the many small files generated by the "dep" phase clutter with the actual source code, and `#include`-relations between .c files (not a recommended practice) are not dealt with correctly.

Finally, Owens pointed out that "make" previously was used for two things:

- defining how objects and libraries are to be combined from other targets, i.e. the build dependency graph

[15] These implicit dependencies cannot be derived from the rule command lists, which means that MAKAO cannot detect them.

[16] `http://osdir.com/ml/kbuild.devel/2002-04/msg00066.html`

- traversing the dependency tree to only do the needed things (based on time stamps)

From his perspective, "make" was only suited, both in speed and accuracy, for the latter. This acknowledges Favre's claims [84] and is related to our second and third root of co-evolution.

To solve all these issues, Keith Owens has proposed many major changes. First, he has simply dropped the "dep" phase. The required dependencies are automatically rebuilt as needed during the normal build phases. Second, an even bigger change is the introduction of "non-recursive make"[17]. Except for a top-level makefile and another one with common build code, all directories get a "Makefile.in" file. These contain high-level declarations of what the developer wants to build and which dependencies or side-effects come into play. The current directory problem of "non-recursive make" is dealt with by considering target locations relative to the Makefile.in's directory and by providing various macros for directory locations. This manipulation requires that `#include`-statements in the source code refer to unqualified file names, i.e. without any path info. The latter should be set up using compiler flags. Third, the kernel and module build phases have been unified. Fourth, contrary to the makefiles in the 2.4 kernel build which depend on environment variables set by other makefiles, developers can now build separate subdirectories. We now explain the most important changes.

Owens did not use an `include`-based "non-recursive make" implementation because the resulting makefile after expansion of all included build scripts would become too big and hence cause overhead. Instead, a complicated five-phase process forms the heart of Kbuild 2.5. Almost every phase uses special preprocessor commands named pp_makefile$i$ with i getting as value 1, 2, 4 or 5:

1. pp_makefile1 processes the required source code hierarchies and records file names, locations and time stamps in a database. This phase has to be rerun during each build to detect changes to files.

2. pp_makefile2 glues the "Makefile.in" files together and preprocesses the result.

3. The preprocessed global makefile is now given to "make", not for actual compilation but to let "make" expand all variables, macro expansions and conditional build logic, and to store the resulting generated rules to disk.

4. pp_makefile4 takes the output of the previous two phases and generates a specialised global makefile. The time stamps stored in the database are

[17] There have been other attempts before to turn the Linux build into a non-recursive "make", e.g. by Martijn van Oosterhout in the 2.2 kernel series (`http://svana.org/kleptog/make/index.html`).

taken into account to only include the necessary build rules. This mechanism detects both newer and older files, i.e. it can deal with source code repositories, clock skew and agile reuse of shadow trees.

5. Finally, the actual build occurs, but with a twist. The generated global makefile uses another preprocessing tool, pp_makefile5, as a wrapper around all compilers and other tools, with the aim of finding out whether the commands succeed or not. In the case of build success, the database entries of the built target and its dependencies are updated with a time stamp, the (parametrised) compilation command and the configuration settings. In case of failure, all dependency information is marked as a failure. This bookkeeping helps for incremental compilation, i.e. in step 4 of the next iteration.

Step 4 took 1 or 2% of the total kernel build time [156]. The total build was 9% faster than the 2.4 build, but could be sped up by building in parallel (only limited by hardware). The generation of the global makefile actually took longer than the build time for small source code changes, but there was an option to bypass this generation and reuse the previously constructed makefile (although this was not completely safe). In any case, speed problems have been dealt with by adding the correct dependencies instead of removing valuable information from the source code architecture. This is clearly related to RC2 and RC3. At the same time, developers can still incrementally add separate build scripts for new components, without the need to change or understand existing makefiles. This facilitates RC1.

Finally, Kbuild 2.5 supports multiple output directories, one per build configuration. The inverse is also possible, i.e. multiple source code directories. This is very practical to incorporate external code bases. This mode is sophisticated, as developers can e.g. specify that some files shadow others or even that the contents of a file should be pre- or appended to another one. This means that instead of sets of patches, the changed source code or just the modifications could be kept in a separate directory hierarchy. This facilitates management of source code changes. As such, this shadow tree mode is much more advanced than the existing configuration infrastructure (RC4).

Regarding the rejection of Kbuild 2.5 by Linus Torvalds, Owens has argued[18] that a step-by-step migration to his approach did not make sense, given the number and nature of changes (syntax, shadow trees, non-recursive build, etc.). As contemporary maintenance actions caused similar scattered changes to makefiles [186], he claimed that a big bang was reasonable. Instead, the Kbuild 2.5 has been blocked from integration into the official kernel source tree, and Kbuild 2.5 has been abandoned. Still, the discussion of Kbuild 2.5 enables us to better understand the rationale behind Kbuild 2.6, which is presented in the next section. As a side note, recently (October 2007) a new proposal for a non-recursive "make" has been

[18] http://lwn.net/Articles/1500/

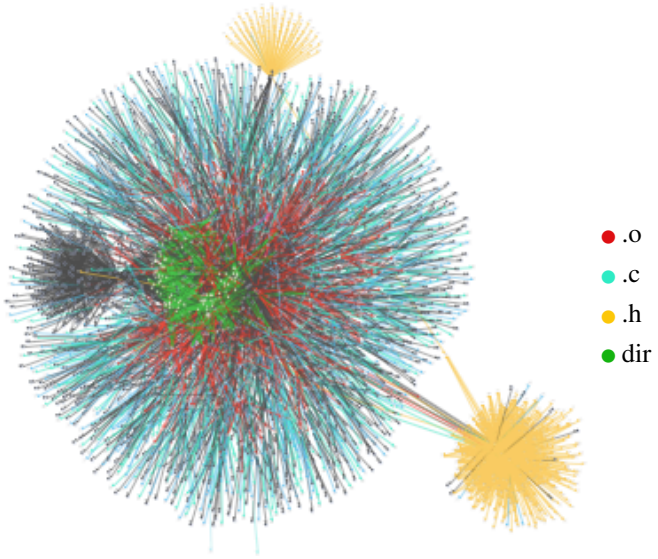


Figure 4.10: Build phase `vmlinux` of the Linux 2.6.0 build process.

launched on the Linux Kbuild mailing list[19]. This time, the promise has been made to reuse the existing makefiles as is.

4.5.3.2 Kbuild 2.6 Converges to Kbuild 2.5 via Build Idioms

This section elaborates on Kbuild 2.6. First, we qualitatively compare the build dependency graph of the 2.6.0 kernel to the one of the 2.4.0 kernel. Then, we focus on the new mechanism for extraction of dependencies. Afterwards, we use MAKAO to unravel the Kbuild 2.6 dependency graph. In doing this, we have discovered various build idioms, each of which forms part of the Kbuild 2.6 implementation to solve the problems addressed by Kbuild 2.5. The three most apparent idioms are discussed, i.e. the FORCE idiom, composite objects, and the circular dependency chain. Especially the last idiom is complex to understand.

Figure 4.8 shows the dependency graph of the 2.4.0 kernel image build (`all`). There are many (16615) dependencies on header files (yellow edges to yellow nodes). On Figure 4.9, these edges have been elided such that we can clearly observe the various object (red nodes) and source files (blue nodes) and the dependencies between them. By querying the static information attached to the nodes and edges, we find that each cluster of nodes corresponds to a particular directory. Kernel subcomponents are delimited by directory boundaries, as mentioned earlier.

[19] `http://www.mail-archive.com/linux-kbuild@vger.kernel.org/msg00026.html`

The corresponding build DAG of the 2.6.0 release is shown on Figure 4.10. Apparently, the majority of header file dependencies have vanished, and the remaining ones (843) are localised in two clusters. In the middle, we can also see a very dense build, almost one huge cluster of nodes. Second, there are many new implicit dependencies (near the leaf nodes) which represent relations between object files and temporary files generated during the build. These hint at changed build commands. Paradoxically, the dependency graph looks much more complex after the maintenance activities than before (Figure 4.9). However, it works much faster and fulfills a lot more functionality than the 2.4.x build.

In the next subsections, we will consider the header dependency disappearance, and dissect the build dependency graph using MAKAO's logic rule engine to drill down into the concrete changes compared to the 2.4 build. We do not consider the logic rules here. They are all listed in Appendix B, and the necessary concepts to understand them have been introduced earlier in Section 3.5.3.

Evolution of Algorithms for Extraction of Source Code Dependencies Just as Kbuild 2.5 does, the 2.6 build system has dropped the separate "dep" phase. As a consequence, dependency generation now occurs as part of the normal build instead of as a separate phase (see section 4.5.2). To make this more efficient, a technique invented by Tom Tromey for "automake" (Section 2.1.3.2) has been applied [167]. Basically, if a source file has been modified, this file has to be recompiled anyhow, hence the file's dependees do not have to be checked to decide whether or not to rebuild. However, the changes to the source file can introduce new build dependencies, or remove existing ones. To generate the file's new dependencies for use in future builds, it suffices to produce them as a side-effect during the actual compilation. Hence, during an initial, full ("clean") build, none of the source code files requires checking of its dependencies, which explains the disappearing of more than 15000 edges between the 2.4.0 and 2.6.0 build. Subsequent builds will discover the previously generated dependencies and incorporate them. This means that dependencies are only taken into account starting from the build iteration following their generation, and until they are out-of-date. This is actually a special case of RC3.

In the 2.6 kernel, the generated dependency files are enhanced with the specific compilation command used to build the particular target. This enables to detect when the build mode is changed from a debug to a release build, such that all compiled files with debugging info are recompiled.

The FORCE-idiom Enhances Make's Time Stamp Checks If we zoom in on Figure 4.10, we obtain Figure 4.11. Apparently, there is one node ("FORCE") which is a dependee of all object files in the system. A "FORCE" node is an idiom frequently used to emulate "phony" targets in older "make" implementations. One

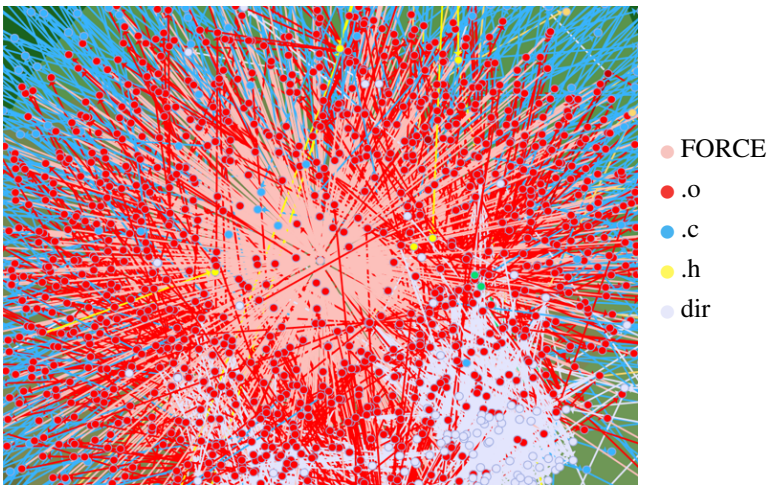


Figure 4.11: Zooming in on the `vmlinux` *build phase of the Linux 2.6.0 build process.*

could interpret a phony target as having a time stamp far ahead in the future. If it is used as a rule dependee, the rule's target always has to be remade as it is older than any phony target. Hence, phony targets can force execution of a rule's command list. Newer implementations like GNU Make have explicit support for phony targets by means of the declarative ".PHONY", whereas the traditional "FORCE" idiom lists the phony target as the target of a rule without any dependencies or command list[20].

Even when building incrementally, the command list of a build rule with a phony dependee will be executed, hence there is no visual difference with a full build graph[21]. At first sight, this seems counter-intuitive. One of "make"'s strengths is to only rebuild what is required, i.e. only execute the command list if needed. Here, the kernel build developers explicitly bypass this by using phony targets as dependees.

To understand what is happening, the relevant build logic is shown on Figure 4.12. Lines 22–24 represent the build rule for compiling .c files into object files. The "FORCE" target is indeed a prerequisite of each object file. The "FORCE" target is declared as phony in two ways here (lines 26 and 28) for backward compatibility. Explicit usage of the ".PHONY" declarative is said to be more efficient than the emulation [218]. The heart of the Linux kernel's FORCE idiom is the call to the GNU Make function `if_changed_rule` on line 23. The definition of this function (lines 1–4) contains a complicated if-test with the condition spread over lines 1–3 and the conditional action on line 4. In the case of

[20] If that rule has a command list, this will be executed anywhere the phony target is listed as a dependency. This is the equivalent of a function call during dependency processing [167].

[21] Except for the addition of header file dependencies starting from the second (re)build.

```
1 if_changed_rule = $(if $(strip $? \
2   $(filter-out $(cmd_$(1)),$(cmd_$(@F)))\
3   $(filter-out $(cmd_$(@F)),$(cmd_$(1)))),\
4   @$(rule_$(1)))
5 ...
6 cmd_cc_o_c = $(CC) $(c_flags) -c -o $@ $<
7 ...
8 define rule_cc_o_c
9   $(if $($(quiet)cmd_checksrc), \
10    echo '  $($(quiet)cmd_checksrc)';) \
11  $(cmd_checksrc) \
12  $(if $($(quiet)cmd_cc_o_c), \
13    echo '  $($(quiet)cmd_cc_o_c)';) \
14  $(cmd_cc_o_c); \
15  $(cmd_modversions) \
16  scripts/fixdep $(depfile) $@ '$(cmd_cc_o_c)' \
17                > $(@D)/.$(@F).tmp; \
18  rm -f $(depfile); \
19  mv -f $(@D)/.$(@F).tmp $(@D)/.$(@F).cmd
20 endef
21 ...
22 %.o: %.c FORCE
23   $(call if_changed_rule,cc_o_c)
24   $(touch-module)
25 ...
26 .PHONY: FORCE
27
28 FORCE:
```

Figure 4.12: Build logic for the FORCE idiom in the Linux 2.6.0 kernel build system.

the call on line 23, the conditional action corresponds to the expansion of the `rule_cc_o_c` macro[22]. This macro does a lot of things, but the most important actions are the expansion of the variable `cmd_cc_o_c` on line 14 and the invocation of the `fixdep`-script. The definition of `cmd_cc_o_c` on line 6 makes it even more clear that the `rule_cc_o_c` macro does the actual compilation, followed by postprocessing of the generated header file dependency files. This means that the GNU Make function `if_changed_rule` determines whether or not compilation is necessary. The Linux kernel apparently uses its own checks to determine whether recompilation is necessary or not, bypassing "make"'s standard time stamp-based heuristic. The control flow of the build is automatically forced into the command list on lines 23–24, in which custom build logic uses time stamp information to decide whether the target really has to be (re)made (i.e. command

[22]The invocation's sole argument is appended to `rule_` to obtain the macro's name.

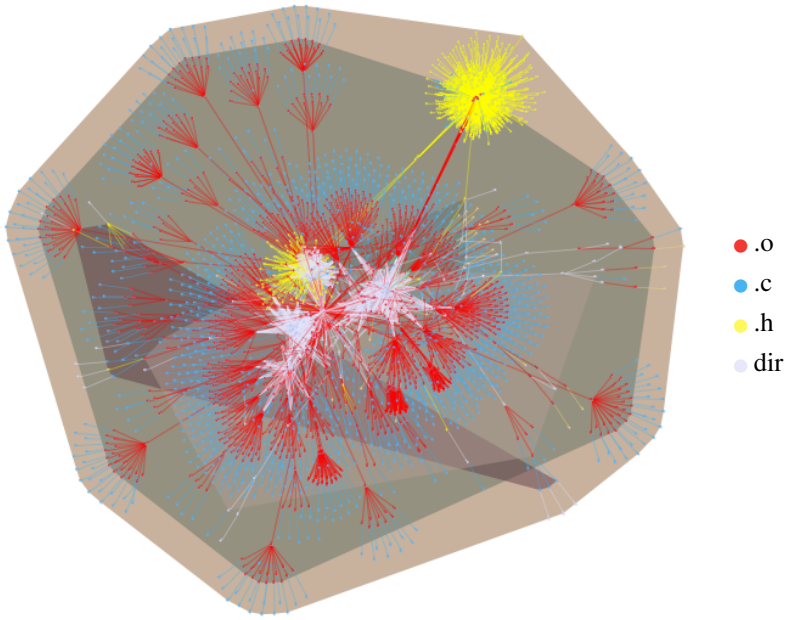


Figure 4.13: Build phase `vmlinux` of the Linux 2.6.0 build process after abstracting away the FORCE-idiom and re-layouting the graph.

list continued) or not.

What exactly does Linux' custom build decision logic do? The basic idea is that changes in time stamp *and* compilation command lead to recompilation. The `if` check on line 1 checks to see whether a list of three values (`$?` and `$(filter-out ...)` on line 2, and `$(filter-out ...)` on line 3) is non-empty, in which case the condition evaluates to true and compilation occurs. `$?` contains all modified source code files, based on "make"'s standard time stamp information. The other two "make" expressions together calculate the difference between the currently used compilation command and flags (`cmd_$(1)`, i.e. `cmd_cc_o_c`) and the previous build iteration's one (`cmd_$(@F)`[23]). The latter command is stored together with the header file dependencies in a separate file per object file. As mentioned before, this is useful information when switching e.g. between production and debug builds. More advanced schemes using e.g. MD5 check sums could be implemented by modifying the sole `if_changed_rule` function.

To summarise, the 2.6 build does not solely rely anymore on "make"'s time stamp-based heuristic to decide when recompilation is required. A custom "make" function is forced to execute via a phony target and enhances time stamp checks

[23] `$(@F)` contains the unqualified file name of the current build target, i.e. the object file.

by taking compilation commands and flags into account. As a consequence, the build dependency graph always reflects the same set of dependencies, whether incremental compilation occurs or not. This is another special case of RC3. The dependency graph is not pruned by omitting dependencies, but instead the static dependencies are left intact and dynamic checks are used to prevent redundant build actions.

An important note to make: a precursor of this "FORCE" idiom already was in use in the 2.4 kernel. There, all object files in a subdirectory depend on a "dummy" target, which is phony. As each directory has its own "dummy" target, this idiom is visually not as spectacular as the 2.6 "FORCE"-idiom. Indeed, the fact that all object files reference one unique, virtual "FORCE" node in the 2.6 build, means that each time this target is encountered within the build, the current directory remains the same. This is easy to understand, as the absolute path of a target is the only way to define identity of build targets in a build graph. In a "recursive make" like the 2.4 kernel has, the current directory each time corresponds to the current subdirectory. This results in various "dummy" targets, one per directory. This learns us that a lot of directory fiddling is taking place in the 2.6 build to obtain a similar situation as in a "non-recursive make"-like build system. Kai Germaschewski indeed has noted that recursion has changed from "make -C subdir" to "make -f subdir/Makefile", which means that the current directory is kept fixed throughout the build (no "-C").

To be able to delve deeper into the Linux 2.6.0 build dependency graph, we need to abstract away the "FORCE" idiom. For this, we can use MAKAO's Prolog component, as we have already shown in Section 3.5.3. The filtering rule we have applied removes `FORCE` and its incoming edges. After relayouting, the resulting graph looks like Figure 4.13. Two more idioms become visible, which we discuss in the following sections.

Composite Objects Structure the Kernel Build System Figure 4.13 reveals new idioms. The constructed Linux kernel image consists of a number of modules which are actually big object files linked together from various smaller ones. This makes these "composite objects" [153] easy to recognise. A concrete example of composite objects is shown in Figure 4.14, which displays the component of the Linux kernel ("net" directory) which is responsible for networking support. The red node at the bottom is a composite object which is linked from an array of eight object files ("simple objects").

The upper four nodes of Figure 4.15 give a more schematic overview of how composite objects participate in the construction of the Linux kernel at the highest levels of the build DAG. Figure 4.15 corresponds to the build subgraph responsible for building a (simplified) version of the "net" subdirectory (the networking subsystem of Figure 4.14). We assume that this directory contains one .c file

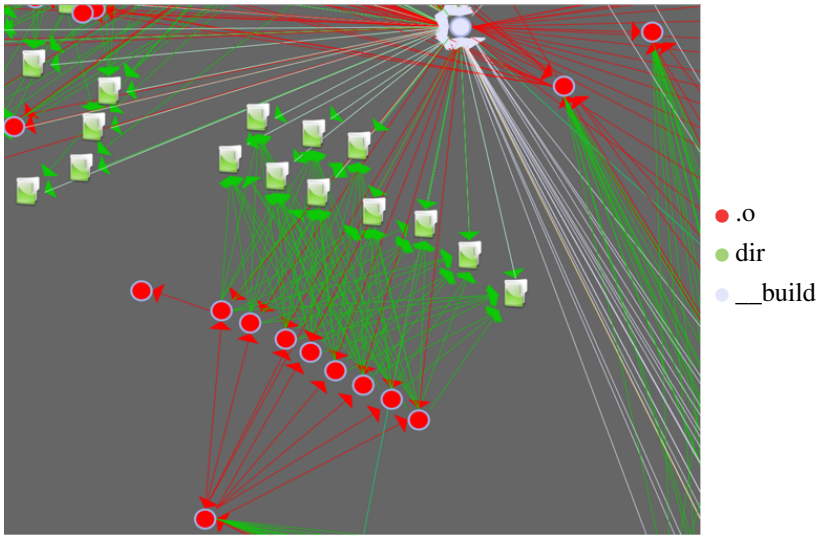


Figure 4.14: Figure 4.13 after zooming in on the networking subsystem. Slightly different colors are used in this graph. Directory nodes e.g. are now light green.

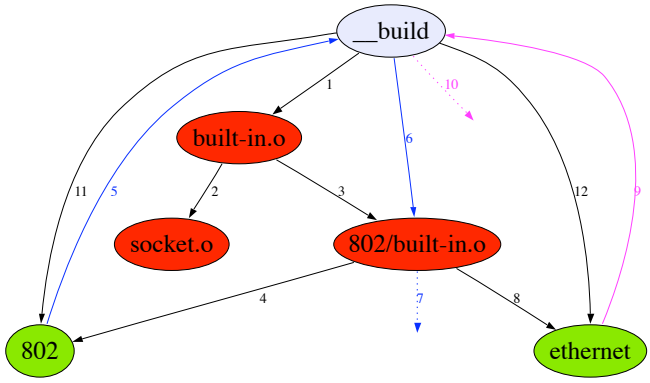


Figure 4.15: Circular dependency chain in the Linux 2.6.0 build system.

("socket.c") and two subdirectories ("802" and "ethernet"), each of which contains its own source code. We further assume that the contents of "802" have to be built into the kernel, while "ethernet" should go into a module. By way of convention, all composite objects in the Linux kernel are called "built-in.o". During the `vmlinux` build (i.e. construction of the kernel image), each of the selected subdirectories (only "802" in this case) generates such a "built-in.o" file, which links the compiled .c files of a directory. The same goes for the "net" directory itself, which eventually contains "socket.o" and "built-in.o". The latter will be

```
1  __build: $(if $(KBUILD_BUILTIN),$(builtin-target) \
2     $(lib-target) $(extra-y)) $(if $(KBUILD_MODULES), \
3     $(obj-m)) $(subdir-ym) $(always)
4    @:
5  ...
6  $(sort $(subdir-obj-y)): $(subdir-ym) ;
7  ...
8  %.o: %.c FORCE
9    $(call if_changed_rule,cc_o_c)
10 ...
11 $(builtin-target): $(obj-y) FORCE
12   $(call if_changed,link_o_target)
13 ...
14 .PHONY: $(subdir-ym)
15 $(subdir-ym):
16   $(Q)$(MAKE) $(build)=$@
```

Figure 4.16: Build logic for the circular dependency chain in the Linux 2.6.0 kernel build system.

linked with the "built-in.o" files of other top-level directories into the actual kernel image ("vmlinux"). The compilation of modules leads to similar results for those directories which represent module code (like "ethernet" in our case). Hence, the Linux kernel build system is structured via a hierarchical, layered composition.

Keith Owens has explained on the Linux build mailing list that composite objects have been in use since the very first Linux version (but not necessarily using the "built-in.o" convention). They are actually one of the workarounds in use in the 2.4 kernel to overcome the "recursive make" ordering problems [156] which we have discussed in Section 2.2.4.2. Composite objects introduce a kind of build layering which reflects the directory structure, as each directory offers a known build interface ("built-in.o") to higher-level directories. Hence, this idiom provides evidence of RC2. However, composite objects have important disadvantages. They cause a lot of linking and extra file activity. Owens attributes a higher probability of out-of-date object files to this idiom. This is why he dismissed this idiom in his Kbuild 2.5, except for modules. In the 2.6 kernel, however, composite objects are still widely used.

The next section discusses an idiom which is complementary to composite objects, the "circular dependency chain".

The Circular Dependency Chain and Generic Build Recursion Logic In combination with composite objects, a phenomenon we named "circular dependency chain" can be observed from Figure 4.14. It seems that all object files in the "net" directory depend on all subdirectories of "net", and that the simple objects (i.e. the

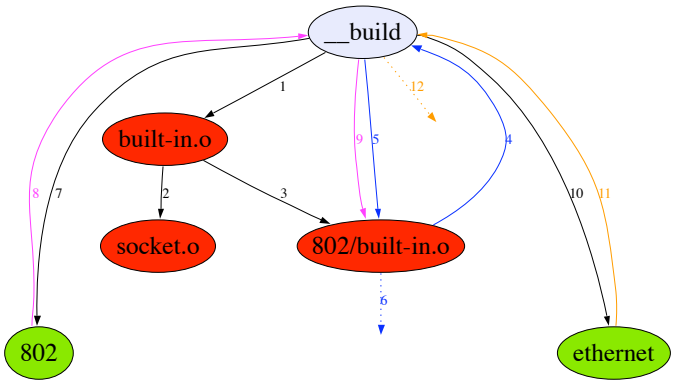


Figure 4.17: An alternative for the circular dependency chain we would expected instead of Figure 4.15.

```
1 $(sort $(subdir-obj-y)):
2   $(Q)$(MAKE) $(build)=$(@D)
```

Figure 4.18: Build logic for the circular dependency chain of Figure 4.17.

array of eight object files) are referenced by the composite object (red node at the bottom) and the "__build" node (white node on top). Especially the high number of dependencies on directories (green edges) seems odd. This section takes a detailed look at this idiom and ultimately explains it in terms of RC1, RC2 and RC3. Note that this section contains a lot of technical details.

Figure 4.15 shows the basic pattern of the circular dependency chain for the simplified "net" example introduced in the context of composite objects. The build logic responsible for this pattern is shown in Figure 4.16. The node colors in the scheme of Figure 4.15 correspond to those of Figure 4.14, but the edge colors identify the active "make" subprocess (more on this later). The central (phony) `__build` target depends on all top-level composite objects (arrow 1; `$(builtin-target)` on line 1 of Figure 4.16), which in turn depend (`$(obj-y)` on line 11 of Figure 4.16) on all object files which have been selected by the developer during configuration (recall the list-style build script syntax of Figure 4.7). These object files correspond to simple objects like "socket.o" (arrow 2), but also to composite objects of subdirectories, like "802/built-in.o" (arrow 3). Dependencies on simple objects are handled by the rule on lines 8–9 of Figure 4.16, which we have explained earlier on in Figure 4.12 in the context of the "FORCE" idiom. Hence, simple objects are recompiled if deemed necessary. Composite objects require more complex build logic.

The subdirectory composite objects are processed by the rule on lines 15–16

```
1 $(sort $(subdir-obj-y)): $(@D)
```

Figure 4.19: Pseudo-GNU Make build logic (which cannot be achieved) for the ideal circular dependency chain.

of Figure 4.16, which corresponds to arrows 4 and 8 on Figure 4.15. The rule states that every subdirectory composite object (each element of the list `$(subdir-ym)`) has no explicit dependencies, but should be built by spawning a new "make" subprocess (`$(Q)$(MAKE)`) to build the particular subdirectory. Edges corresponding to build dependencies in a new "make" subprocess are colored differently on Figure 4.15. Hence, building a subdirectory graphically boils down to a differently colored dependency on "__build" like arrows 4 and 8, followed by a change of the current directory to the particular subdirectory passed as an argument to the subprocess, e.g. "802". The change of directory enables "__build" to depend on a specific composite object of a subdirectory (arrow 6). This composite object is built in the same way as we are describing now for "net/built-in.o", so we do not elaborate on arrow 7.

The explanation up until now has unraveled a large part of the spine of the Linux kernel build logic, i.e. the interplay between simple and composite objects. However, there are some inexplicable arrows on Figure 4.15. The presence of arrow 4 of Figure 4.15 is expected, as "802/built-in.o" has to be built anyway. However, the appearance of arrow 8 is unexpected, as we have selected "ethernet" to be a module whereas the dependency subgraph on Figure 4.15 corresponds to the "vmlinux" build phase, i.e. the build of the kernel image. Indeed, line 6 of Figure 4.16 explicitly mentions `$(subdir-ym)` instead of just `$(subdir-y)`, hence subdirectory composite objects depend on subdirectories of "net" which have been selected as built-in code or as module. Just as for "802", a subprocess is spawned and "ethernet/built-in.o" is built (arrow 10). Arrow 8 has consequences on the remainder of the build graph.

At this point (arrow 8), all dependencies of "net/built-in.o" have been built. Now, some sanity checks are performed. In particular (`$(subdir-ym)` on line 3 of Figure 4.16), "__build" requires that all "net"'s subdirectories are built. This seems like overkill, as both arrows 11 and 12 result in "pruning" of the build graph. This is an optimisation of GNU Make which makes sure that any target considered for (re)build is only built once within a given "make" subprocess. The underlying assumption is that the particular target's dependees will not have changed in the meantime[24]. In that case, every time the target is re-visited, the current build path stops and the build backtracks. On Figure 4.15, pruning occurs any time two edges of the same color enter a given node. Hence, arrows 11 and 12 result in pruning.

[24]This is actually not that conservative, as it is easy to modify a target's dependees in between two compilation attempts of it.

The cleanup performed by "__build" is overkill if there is at least one subdirectory composite object specified as a dependee of the "net" directory's composite object. Indeed, if "802" would not have been selected as a dependee of "net/built-in.o" to be built into the kernel, but as a module instead, neither "802" nor "ethernet" would have been built when the sanity checks would start. The latter would notice this, and build the two subdirectories' composite objects. To summarise, the build logic shown on Figure 4.15 contains inexplicable dependencies like arrow 8 between composite objects and subdirectories, and seemingly redundant dependencies like arrows 11 and 12 between "__build" and the same subdirectories. Together, these edges explain the build graph of Figure 4.14. The remainder of this section discusses the rationale behind these strange dependencies.

The explanation for the sanity check dependencies is straightforward. The following note in the main makefile of the Linux kernel build system gives us a hint: "We are using a recursive build, so we need to do a little thinking to get the ordering right.". Indeed, similar to the composite objects idiom, arrows 11 and 12 of the circular dependency chain correspond to a clever iteration strategy the Linux developers have added to their "recursive make" to avoid problems with the evaluation order of targets [171]. This is one of the alternatives for "recursive make" we have discussed in Section 2.2.4.2. This workaround tries to fix dependencies which are not guaranteed to exist because of the deficiencies of "recursive make", but which should have been expressed in the makefiles to closely model the kernel architecture (RC2) and to make a valid system composition (RC3). This still does not explain arrow 8, however.

The remaining question is: why does every subdirectory composite object depend on every subdirectory of the enclosing directory? We have tried to rewrite the build logic without these dependencies, but this turned out to be impossible. Figure 4.17 shows the build graph which gets closest to the expected situation. This graph corresponds to the code of Figure 4.18, which replaces line 6 of Figure 4.16. The difference with Figure 4.15 is the absence of dependencies of "802/built-in.o" on "802" and "ethernet". Instead (arrow 4), "802/built-in.o" immediately invokes a "make" subprocess to build itself within the "802" subdirectory (arrows 5 and 6). The cleanup performed by "__build" does make sense now. First (arrows 7, 8 and 9), the build process learns that "802/built-in.o" already exists. The second cleanup check does a useful job (arrows 10, 11 and 12), as it detects that "ethernet/built-in.o" does not yet exist and builds it[25]. Hence, we observe that it is possible for a subdirectory composite object to not depend on all subdirectories, but only at the expense of an extra "make" subprocess. This separate process is needed because "__build" explicitly depends on subdirectories (`$(subdir-ym)` on line 3 of Figure 4.16) instead of on the composite object inside these directories to keep

[25] It is not clear to us why this functionality has to be built during a kernel build, but based on private email communication there seem to be hidden dependencies on module code.

track of the current directory. Only after dispatch to the new "make" process the mapping of the subdirectory to the particular "built-in.o" is made. We can now investigate the ideal solution, which reduces the number of "make" processes, but is not feasible in practice.

Figure 4.19 shows the pseudo-GNU Make code needed to add "802" as a dependee of "802/built-in.o" instead of spawning a subprocess (arrow 4 on Figure 4.17) in the command list as Figure 4.18 does. `$(@D)` is a special GNU Make variable which always refers to the directory part of the name of the current rule's build target. Unfortunately, this variable can only be used within a rule's body, not in the list of dependees. This is unfortunate because the latter is required to maintain the generic build rules used by Kbuild 2.6. More in particular, the 2.6 build rules try to obtain genericity by using rules with multiple targets at once (with "a" and "c" as directories):

```
1 a/b c/d: a c
```

This is equivalent to:

```
1 a/b: a c
2 c/d: a c
```

The ideal rule of Figure 4.19 tries to reduce the dependencies on other directories in a generic way:

```
1 a/b c/d: ``if a/b then a else if c/d then c''
```

This would be equivalent to:

```
1 a/b: a
2 c/d: c
```

However, GNU Make is not expressive enough to allow this. As our description of Figure 4.15 and Figure 4.17 has revealed, the circular dependency chain idiom is the actual engine of the 2.6 kernel build. It is a centralised, generic piece of build logic which expresses how the build should find its way through the build dependency graph. It combines generic, parametrised build rules (Figure 4.16) with current directory manipulation and GNU Make-specific features like pruning. The aim of this build logic is to resolve the problem of "recursive make" where each subdirectory has to reimplement the recursion logic. This is tedious and error-prone, as it requires every developer to understand and modify the existing build system when new components are added (RC1). The 2.6 kernel tries to do better, but in doing this, it meets the limitations of GNU Make. GNU Make does not allow to express the build dependencies in a generic way other than using multi-target rules, but then it is not possible to only mention the directory of the currently built rule dependent as a dependent. Hence, a workaround is needed. As the next most logical solution requires an extra "make" subprocess (Figure 4.17),

a compromise is used which does the job, but cannot be understood easily (Figure 4.15). In other words, the evolution of the build logic digresses from the actual source code's evolution path (RC2) to facilitate integration of new source components (RC1).

Figure 3.5b on page 91 shows the resulting build dependency graph after filtering out the above idioms using the logic filtering rules of Appendix B. The result resembles Figure 4.9's structure, as the basic directory structure has remained stable between 2.4.0 and 2.6.0, and each directory more or less corresponds to a composite object (RC2). The big difference between 2.4.0 and 2.6.0 are the idioms which have been used to produce a higher-quality build at the expense of extra complexity during the build.

4.5.3.3 Summary

The detailed discussion of changes introduced in Kbuild 2.6 compared to Kbuild 2.4, and the design of Kbuild 2.5 have given us valuable evidence of the existence of co-evolution of source code and the build system, and of the validity of all four roots of co-evolution (summarised in the next section). At the same time, MAKAO has enabled us to identify and understand the complex build idioms used in Kbuild 2.6, i.e. to identify and understand the workarounds introduced to deal with co-evolution of source code and the build system in the Linux kernel build system. The next section gives an overview of all evidence gathered in the Linux kernel build system case study.

4.6 Validation #1: Roots of Co-evolution Experimentally Confirmed

This section distills the various elements of evidence gathered throughout the Linux kernel build system to prove the validity of the four roots of co-evolution (Section 2.3.4). This evidence is grouped by root of co-evolution. Afterwards, the next section (Section 4.7) discusses the validity of MAKAO.

4.6.1 RC1: Modular Reasoning vs. Unit of Compilation

The measurements of SLOC and file count in Section 4.3 have indicated that build logic and to some degree configuration logic are distributed across the source code architecture, i.e. that separate source code components have a dedicated piece of build and configuration logic. This corresponds to the notion of "recursive make" as presented in Section 2.2.4.2. However, during the evolution of the build system in Section 4.5, we have seen that the tension between "recursive make" and the ease of plugging in build logic of new components has been a continual source

of build maintenance. We have described one of the earliest recursion schemes in Section 4.5.2, which requires explicit knowledge from existing makefiles and the right traversal strategy. To accommodate this, a common body of build logic has been extracted ("Rules.make") and the build specification has been lifted to a list-style approach in which developers only need to manipulate build variables.

As Keith Owens has noted [156], the build scripts in the 2.4.x build system still were not independent from each other, as they required environment variables to be initialised by their environment. This situation was slightly better than the context considered by de Jonge [57], but still not good enough. Keith Owens has proposed a radical change to a "non-recursive make", in which each source code component has its own build script which is almost independent of other scripts. However, the fear of breaking the Linux build without an incremental migration path to the new build system has blocked Kbuild 2.5. Instead, the 2.4.x build system has been incrementally rewritten, especially to deal with the problems caused by "recursive make". The circular dependency chain is a compromise to establish a generic build logic kernel with limited build technology (GNU Make) at the expense of artificial build dependencies.

The continuous struggle to improve the build system for integration of new source code components is clear evidence of evolution of the build system as an answer to source code modularity. This validates RC1.

4.6.2 RC2: Programming-in-the-large vs. Build Dependencies

The measurements of SLOC and file count in Section 4.3 have also shown that build and configuration logic physically co-evolve with the source code, i.e. they follow a similar growth pattern but at different order of magnitude. The fact that build and configuration logic are distributed across the source code directories is a second indication that the build system co-evolves with the source code, as directory structure in legacy systems typically reflects the source code architecture [39].

The desire to keep the build dependency graph synchronised with the source code architecture is a second source of tension in the Linux kernel build system. The constant improvements on extraction techniques of source code dependencies (Section 4.5.2), Keith Owens' "non-recursive make" and the idioms in Kbuild 2.6 clearly demonstrate this. The composite objects idiom (4.5.3.2) e.g. shadows the source code architecture during the build to avoid the imprecise dependencies of a "recursive make", similar to the sanity checks of the circular dependency chain (4.5.3.2). However, the latter idiom disturbed the synchronicity by adding artificial dependencies.

These constant maintenance efforts show that RC2 is an important force behind build system maintenance and that it is governed by the desire to stay connected to the source code architecture.

4.6.3 RC3: Interface Consistency vs. Incremental Compilation

There was no numerical data on RC3. Instead, online documentation and published email discussions have shown that a strong tendency exists to leave out important architectural dependencies in the build system just to speed up the build process (Section 4.5.2). The build maintainer of the Linux 2.4.x kernel build system has aptly summarised this as "correctness trumps efficiency". Hence, a long stream of new dependency extraction techniques has been proposed — the `dep` phase has even been sacrificed later on — in parallel with the techniques to work around the problems introduced by "recursive make".

The most advanced solution to date has been the Kbuild 2.5 system (Section 4.5.3.1). Because it implements a "non-recursive make", there are no valuable dependencies between source code components missing. The improved dependency extraction technique makes sure that the dependencies which are part of the build dependency graph are as complete as possible. Kbuild 2.6 has tried to get as close as possible to Kbuild 2.5 via the FORCE idiom (4.5.3.2). This idiom does not prune static dependencies in the dependency graph, but instead always traverses the whole dependency graph and checks for each target if re-compilation is required or not. The sanity checks of the circular dependency chain (4.5.3.2) on the other hand add redundant dependencies to make the compilation more correct.

A slightly different take on RC3 is the dependency extraction technique of Tom Tromey (4.5.3.2). This speeds up a full build, i.e. the very first compilation attempt, by not checking any header file dependencies at all. In subsequent incremental builds, header file dependencies are exploited, but not for compilation units which have changed. To date, this approach is still used, as the risks it takes are rather limited.

To conclude, throughout the evolution of the Linux build system, RC3 has been a constant force to assure that the actual source code dependencies are not randomly ignored during incremental compilation, even though compilation speed has always been very precious to kernel developers. Hence, RC3 is a source of co-evolution of source code and the build system.

4.6.4 RC4: Program Variability vs. Build Unit Configuration

The configuration system has only been studied directly by the measurements of Section 4.3. There, we have seen that the SLOC and file count of source code has exploded, and that configuration logic has seen a similar evolution. Starting from nothing, it has overtaken the SLOC of build scripts near the 2.4.x kernel. Together with the observations that especially driver code attributes to the growth of source code [99], this gives an indication of the high demands on the configuration system to be able to manage the source code's variability.

Further evidence, especially in the kernel documentation [152, 154], shows

that the configuration layer has its own graphical front ends to facilitate configuration of the kernel. The huge amount of configuration options is structured hierarchically with constraints between conflicting or co-operating choices. The introduction of high-level domain-specific languages to textually specify configuration options has further fostered the integration of new configurability in the source code.

Finally, Kbuild 2.5 has proposed a novel way of dealing with different versions of the source code, e.g. with different patches applied. Its shadow tree feature allows to maintain separate directory trees and even provides pre- and append actions to non-destructively compose new versions of the source code.

To summarise, the source code has put a tremendous pressure on the configuration layer to manage the sheer unlimited potential of configurability in the kernel. Evolution of the build system has been driven by the desire to keep configuration manageable and to only produce valid configurations.

4.7 Validation #2: MAKAO Achieves Goal T2

This section validates the ability of MAKAO to support developers in identifying and understanding symptoms and workarounds corresponding to co-evolution of source code and the build system.

MAKAO has been used in the Linux kernel build system case study to measure the internal complexity of the build dependency graph (Section 4.4), and to understand the semantics of the Linux build. Calculation of the complexity has only required the number of targets and dependencies (explicit/implicit) in the graph, whereas for the build semantics, we have exploited MAKAO's visualisation, querying and filtering functionality. Visualisation has given us clear visual clues of changes to the structure of the build system, like Tom Tromey's dependency generation (4.5.3.2) or the compact 2.6.0 build DAG (Section 4.5.3). Querying has enabled us to access detailed information about build commands or e.g. time stamps, i.e. to clearly understand the behavioural parts of the build DAG and values of build and configuration parameters. Filtering has provided us with the flexibility to leave out artificial dependencies introduced by build idioms in the Linux 2.6.0 kernel build DAG in order to investigate the actual structure of the build and to identify remaining build idioms (Section 4.5.3.2). Hence, MAKAO has enabled us to understand the evolution of the build system and as such has helped us to identify workarounds and problems caused by co-evolution of source code and the build system.

The features which have been used correspond to those which have been suggested by the roots of co-evolution in Section 3.1.2 for understanding the co-evolution of source code and the build system. RC1 requires knowledge about build components and about the modularity of build scripts. The former corre-

spond to clusters of targets in the build dependency graph, like the composite object idiom (4.5.3.2), whereas MAKAO's convex hulls give indications about the modularity of build scripts. The visualisation of dependencies facilitates RC2 and RC3. RC3's extra requirement for a means to detect whether dependencies have disappeared has only partially been fulfilled. Clear differences like for Tom Tromey's dependency extraction technique (4.5.3.2) can be observed visually. However, there is no prescribed way to automatically detect finer-grained differences. Filtering can be used to detect certain patterns of (missing) dependencies, but there is no "diff"-functionality between two graphs. Finally, RC4 requires knowledge of selected source code modules in the build system and communication of configuration choices between source code and the build system. The latter has not been validated because we were not interested in this data. The collection of selected modules on the other hand corresponds to the existing build targets in the build DAG.

As discussed in Section 3.1.1, MAKAO is not able to study configuration specifications. However, the available documentation and online developer archives have provided us with enough data to distill useful observations about the configuration layer.

4.8 Conclusion

This chapter has examined the Linux kernel build system as it has evolved over time. This analysis has provided evidence of the validity of the four roots of co-evolution and has also validated MAKAO's support for understanding and identifying symptoms of co-evolution of source code and the build system. This means that the four roots of co-evolution have been acknowledged as crucial sources of the co-evolution. This experiment does not prove that these roots are the only ones which form an explanation of co-evolution of source code and the build system, but we do not claim this either. Further experiments should point this out, but conceptually and experimentally the roots have been proven to cover a wide range of co-evolution dimensions. As a consequence, if a source code change can be associated to one of the roots, that root predicts that build system changes are necessary to keep the source code and the build system consistent. This means that the introduction of AOSD in a legacy system, which has been shown to have an impact on each of the roots of co-evolution (Section 2.4), requires important changes to the build system in order to keep the source code and the build system co-evolving.

MAKAO has proven itself as a valuable framework to understand a build system and to support identification and understanding of symptoms of co-evolution of source code and the build system. In Chapter 6 to Chapter 10, MAKAO is applied on five legacy C systems to understand *and* manage co-evolution of source

code and the build system when AOSD technology is being introduced. Before we can report on these case studies, the next chapter introduces Aspicere, i.e. the aspect language for C which has been used throughout the five cases.

ORIGIN late Middle English (denoting the action or a way of looking at something): from Latin aspectus, from aspicere 'look at', from ad- 'to,at' + specere 'to look'.

Origin of word "aspect" (Oxford American Dictionaries)

5

Aspicere, AOP for Legacy C Systems

IN the previous chapters, we have discussed the co-evolution of source code and the build system in legacy systems. We have proposed a conceptual explanation of this co-evolution (four roots of co-evolution), and requirements and design of a re(verse)-engineering framework for build systems, i.e. MAKAO. The roots of co-evolution and the tool support for understanding and managing co-evolution of source code and the build system have been validated on the Linux kernel build system. In this chapter[1], the focus of the dissertation shifts to legacy systems where AOP technology is introduced, either for reverse- or re-engineering purposes. The tool support proposed in Chapter 3 should still be valid in this environment, because we have not made assumptions on the source code. However, we do need an aspect language which is able to deal with legacy systems and which facilitates management of co-evolution of source code and the build system. This chapter distills requirements for such aspect languages and presents the design of an aspect language for C, Aspicere, which satisfies these requirements.

In the remainder of this dissertation, we especially focus on legacy C systems. C is unique in the sense that it is one of the only languages for which many legacy systems are readily available, e.g. as open source, and for which tool support does not need to deal with dozens of different dialects. This chapter first distills requirements for an aspect language for legacy systems (Section 5.1), and then evaluates how existing aspect languages for Cobol and C deal with these requirements (Section 5.2). Afterwards, the design and implementation of our own aspect

[1] Parts of this chapter are based on [248, 2, 8].

language for C, Aspicere[2] is discussed in Section 5.3. It is based on the principles of LMP [237, 30, 202]. We consider the rationale behind it, the language itself and the two weavers (Section 5.4) which have been developed for it. To put the design of Aspicere in perspective, we compare it with an industrial aspect language for C which has only recently been proposed. Aspicere is used as the primary aspect language in the case studies presented in the five subsequent chapters. Section 5.5 briefly explains the intent of these case studies and the outline of the remainder of this dissertation. Finally, Section 5.6 presents the conclusions of this chapter.

5.1 Requirements for an Aspect Language for Legacy systems

This section distills requirements for aspect languages for legacy systems (goal L1) which are able to facilitate dealing with co-evolution of source code and the build system (goal L2). The requirements for each of these two goals are discussed in the following two subsections.

5.1.1 Goal L1: Language Features to Deal with Legacy Systems

De Schutter [202] has made an explicit account of the rationale behind and the design of an aspect language for typical legacy (Cobol) systems. Other researchers have discussed specific facets of aspect language design in legacy environments [50, 34, 75, 179]. From this work, we have distilled four requirements for aspect languages for legacy systems:

base integration the aspect language constructs should blend with the base programming language

expressive pointcuts the pointcut language has to be expressive to deal with weaker base language provisions for typing, structuring, etc.

generic advice advice should be robust to small variability in types and context across the advised join points

join point context the language should offer access to sophisticated join point context

The first requirement is about integration of a new technology, both psychologically and practically. As Cobol programmers are fluent in writing Cobol code and mostly weary of new technologies, adoption of aspects can be accelerated if the aspect language does not try to copy or re-implement existing features [75].

[2] Aspicere: verb, Latin, "to look at". Root of our modern word *aspect*.

Instead of a separate `aspect` construct like in AspectC++ or AspectJ, it is much more natural to adopt ordinary Cobol files as aspect. One just needs to add new constructs for pointcuts and advice, but they should be as close as possible to existing language constructs to lower the learning curve. This also makes integration into existing IDEs easier, because these only need to support the new advice and pointcut concepts.

The second and third requirement can be illustrated best by an example. Bruntink et al. [33, 34] have used AspectC [50], the first aspect language for C (described later), to implement an aspect which checks whether or not pointer arguments passed to a procedure correspond to a `null` pointer. As C does not have a kind of "super-type" similar to Java's `Object`[3] and it does not have templates like e.g. C++ provides, there is no type-safe way to refer to a generic type. As AspectC does not have explicit provisions for dealing with this, Bruntink et al. [33, 34] have been forced to duplicate their argument checking advice for each occurring argument type and to use plain enumerations of procedure names as pointcut. This situation impeded maintenance, as the long enumeration-based pointcuts had to be adapted on every non-trivial source code change, and changes to the advice logic had to be percolated to all duplicates of the advice. To resolve these problems, Bruntink et al. [33, 34] have developed a domain-specific language (DSL) for parameter checking, which is translated by a preprocessor to AspectC advice. Although they show that their solution greatly improves the source code quality, it still remains an ad hoc solution. Aspect languages for legacy systems should provide support for writing robust pointcuts and to specify generic advice, i.e. advice which is robust to small variability in types and context across all join points it advises.

As an answer to the need for writing robust pointcuts, De Schutter [202] has proposed to make pointcut languages more expressive such that pointcuts can rely on sophisticated patterns expressed in terms of program structure and meta data instead of being enumeration-based. He has based the pointcut language of Cobble [142, 202], his aspect language for Cobol, on the pointcut language requirements proposed by Gybels et al.[4] [108]:

set of predicates to select join points with the right properties and to do general-purpose actions like list handling, mathematical operations, string manipulation, etc.;

Prolog-like unification to make pattern matching and variable bindings available in an elegant way;

[3] C does have `void` pointers, which can point to anything, but using them precludes compile-time type checking.

[4] A join point is a run-time concept. However, for each join point a corresponding location in the source code can be found (join point shadow) which contains the actual code which is executed by the join point. E.g. an actual procedure call forms the shadow of a `call` join point. Shadows are used by compile-time weavers to statically weave aspects.

access to join point shadows for navigation through the (static) structure of the base program;

parametrisable pointcut definitions to facilitate reuse;

recursion to render the pointcut language computationally complete.

Support for all these features makes the pointcut language Turing-complete, which may be overkill feature-wise and may induce performance overhead [109]. However, the erosion of program structure in legacy environments needs more declarative and descriptive pointcuts instead of unevolvable enumerations of procedure names. The support for reusable pointcut definitions enables experts to implement a library of domain-specific pointcuts [30] which can be used by base code developers. Hence, only a limited group of experts needs to face the full complexity of the pointcut language. This reconciles the expressive pointcut language with the base integration requirement, and prevents manageability, security and reliability problems.

The fourth requirement, i.e. sophisticated access to join point context, refers to the base elements in terms of which pointcuts are expressed. To be able to specify robust patterns of join point, Gybels et al. [108] and De Schutter [202] have proposed access to program structure as a prerequisite. De Schutter has elaborated on this by stressing the importance of weave-time meta data in pointcuts, i.e. logic facts which represent design information or results of reverse-engineering analysis. They allow to write more robust pointcuts which is synchronised with design changes or more precise analysis results. In general, any kind of information could be offered as context to pointcuts [185].

To summarise, aspect languages for legacy systems should blend naturally with the base programming language, should support generic advice, should offer means for composing expressive pointcuts and should offer access to sophisticated join point context. The next section discusses one additional aspect language requirement for dealing with co-evolution of source code and the build system.

5.1.2 Goal L2: Integration of the Build System with the Aspect Language

This section extracts one important aspect language requirement from the four roots of co-evolution to be able to deal with co-evolution of source code and the build system, i.e. to make this co-evolution as easy as possible. The requirement follows directly from an observation made for RC4 in the context of AOP (Section 2.4.5 on page 67): the build system should be integrated with the aspect language. This integration lets build system information flow to the aspects, both to pointcut and advice, to make them robust to build system and source code changes. Recently, Nagy et al. [179] have suggested a similar requirement.

In the context of co-evolution of source code and the build system, the integration of the build system with the aspect language facilitates exchange of build structure and configuration information. The former enables access to the build components and their dependencies. As the build system is typically used to model high-level source code architecture [84], this means that integration of the build system with the aspect language enables aspects to exploit this high-level information for specifying more expressive pointcuts which are automatically synchronised with the build system's current understanding of the source code architecture. This facilitates dealing with co-evolution problems related to RC1 and RC2. It is less obvious how RC3 can benefit from this, although knowledge of pruned build dependencies during incremental weaving could contribute to a more advanced incremental weaving implementation by limiting the scope of aspects to the actually used build dependencies. The advantages to RC4 have been explained before in Section 2.4.5. Configuration information can enable advice to replace traditional conditional compilation in e.g. C-based systems.

Note that, conversely, the aspect language could also be integrated with the build system to make the build system AOP-aware. This would mean that the fine-grained composition offered by aspects would be understood by the build system, such that the build system could directly deal with any of the problems caused by co-evolution of source code and the build system. We do not pursue this further, because replacing the build system in a legacy system is not feasible, as outlined in Chapter 2. Moreover, this integration is not general enough to deal with arbitrary paradigm changes in the source code.

To summarise, we have proposed that the build system information should be integrated with an aspect language for legacy systems, to improve the co-evolution of source code and the build system in the presence of AOP. The next section discusses existing aspect languages for C to evaluate their support for our requirements for aspect languages for legacy systems.

5.2 Evaluation of Existing Aspect Languages for C

To experimentally validate the existence of problems caused by co-evolution of source code and the build system when AOP technology is introduced in a legacy system, we need an aspect language to complement MAKAO. The choice for an aspect language assumes that a particular base programming language has been selected, which restricts the case studies we can perform. De Schutter e.g. has focused on aspects in Cobol [202]. Because no aspect language for Cobol existed, he has designed and implemented his own aspect language, Cobble [142]. This has offered him more control, but at the same time Cobol's extensive language specification with many different dialects made finding a case study much harder. Only industrial Cobol code bases could be found, tied to one particular Cobol

dialect, but no open source systems to experiment with.

To avoid this kind of problems, we have considered other, more recent languages which still have a large share in legacy systems. C turns out to be an ideal trade-off between legacy language and availability. Millions of lines of C code have been written for the Linux/Unix platform, as well as for Windows. Many current open source projects still choose C as their main programming language. The TIOBE Programming Community Index[5], which collects data from major search engines as well as takes into account the worldwide availability of skilled engineers, courses and third party vendors, nominates C as the second most popular programming language in January 2008, with a share of 13.92% (Java reaches 20.85%, Visual Basic 10.96). Although we do not have precise numbers to back this statement, we consider CâĂŹs rank in the industrial popularity poll as very close behind Cobol.

Contrary to the Cobol case, there are many existing aspect languages for C. None of them supports all requirements of goals L1 and L2, and there are many languages for which no weaver implementation readily is available. In this section, we discuss the existing aspect languages for C, including newer ones which have arisen during the course of our research (except for Mirjam, which is discussed in Section 5.3.6). It is our intention to give a fair comparison of available features and to make clear why we eventually have designed our own aspect language for C, Aspicere (Chapter 5).

Our findings are summarised in Table 5.1 and Table 5.2, grouped by weaving time. Each table contains six sets of rows. The first four correspond to the available support for the four requirements of goal L1, the next one to goal L2 and the last set of rows gives characteristics of the languages' weaver implementation. The tables are grouped by category of weaver, i.e. compile-time, run-time and VM weaving. The same order is used in the next subsections to discuss the aspect languages for Cobol and C. We also briefly discuss a Model-Driven Engineering (MDE) approach for weaving C systems.

5.2.1 Aspect Languages with a Compile-time Weaver

These aspect languages are accompanied by a weaver which transforms the base code and all involved aspects into a regular C program. This represents the woven system and can be compiled like any normal C system. Because it has had a big influence on Aspicere, we include a description of Cobble, the aspect language for Cobol designed by De Schutter [202, 142]. Table 5.1 gives an overview of the characteristics of all aspect languages discussed in this subsection.

	Cobble	AspectC	AspectC++	AspectX	C4	WeaveC	ACC
domain	general	kernel	general	general	systems	general	general
base integration	+	+	.h	XML	+	XML	+
preprocessor	N/A	-	-	#include	+	-	-
PCD robustness	LMP	-	regexp	XPath	-	regexp	regexp
(function) pointers	N/A	-	-	-	+	-	`callp`
ITD	-	-	+	+	+	+	+
basic join points	Cobol	AspectJ	AspectJ	+	no `call`	AspectC	+
dynamic join points	-	AspectJ	AspectJ	-	`-(cflow)`	+ (pro)	+
advanced join points	-	-	`callsto/ reachable`	comments	-	-	`preturn`
variable access	+	-	-	+	-	-	+
generic advice	LMP	-	+	+	-	+	-
aspect interaction	build	build	explicit	build	patching	priority	convention
advice interaction	lexical	lexical	lexical	lexical	lexical	priority	lexical
context	+	-	+	+	-	+	+
thisJoinPoint	-	-	+	-	-	-	+
annotations	-	-	-	-	-	+	-
build integration	-	-	-	-	+	-	-
availability	+	+	+	+	+	-	+
weaver type	source	source	source	source	source	source	source
optimisation	-	-	+	-	-	+ (pro)	-
K&R support	-	+	-	+	+	-	+
IDE	-	-	+	-	-	-	-

Table 5.1: Overview of existing aspect languages for Cobol and C (part 1: compile-time weavers). The upper four sets of rows correspond to the provisions for the four requirements of goal L1, the next one for goal L2, and the lowest rows characterise the languages' weaver. A +/- indicates good/bad support for a feature, whereas "N/A" signals when an entry is not applicable to a language. Because every aspect language supports access to function arguments and global variables, a "-" for "context" means that there is no additional means for access to context.

	μDiner	TinyC^2	Arachne	TOSKANA	KLASY	TOSKANA-VM
domain	systems	general	systems	BSD kernel	BSD/Linux kernel	general
base integration	+	+	+	+	XML	?
preprocessor	-	-	-	-	-	-
PCD robustness	-	regexp	+	-	-	?
(function) pointers	-	-	-	-	+	-
ITD	-	-	-	-	+	-
basic join points	+	no `call`	+	no `call`	+	+
dynamic join points	+	-	+	-	-	-
advanced join points	`sequence`	-	+	-	-	-
variable access	+	-	+	-	+	+
generic advice	-	+	-	-	+	?
aspect interaction	deployment	deployment	deployment	deployment	deployment	?
advice interaction	lexical	lexical	lexical	lexical	lexical	?
context	-	-	+	-	+	?
thisJoinPoint	-	-	-	-	-	?
annotations	hookable	-	-	-	-	-
build integration	-	N/A	N/A	N/A	N/A	N/A
availability	-	-	+	-	+	-
weaver type	run-time	run-time	run-time	run-time	run-time	VM
optimisation	-	-	-	-	-	+
K&R support	-	N/A	N/A	N/A	N/A	N/A
IDE	-	-	-	-	-	-

Table 5.2: Overview of existing aspect languages for C (part 2: run-time and VM weavers). The upper four sets of rows correspond to the provisions for the four requirements of goal L1, the next one for goal L2,whereas the lowest rows characterise the languages' weaver. A +/-/? indicates good/bad/unknown support for some feature, whereas "N/A" signals when an entry is not applicable to some language. Because every language supports access to function arguments and global variables, a "-" for "context" means that there is no additional means for access to context.

```
1 USE BEFORE (ADD OR SUBTRACT)
2 AND BIND VAR-ITEM TO SENDER
3 *> Disregard the item if it is a receiving data item, too.
4 AND NOT IS RECEIVER VAR-ITEM
5 MY-SENDER-ADVICE.
6 *> Count the sending data item if it equals zero.
7 IF VAR-ITEM = ZERO
8 ADD 1 TO COUNT-ZERO-ITEMS.
```

Figure 5.1: Cobble aspect which counts the number of zero-valued sending data items [142].

5.2.1.1 Cobble

Cobble is the aspect language for Cobol developed by De Schutter [142, 202]. Supported join points correspond to the major Cobol statements, program execution and procedures. Cobble's weaver[6] is tied to one particular dialect (hence the "-" for "K&R support"), i.e. acucobol. No inter-type declaration is supported by the Cobble prototype, nor more advanced or dynamic join points. Interaction between aspects and advice is determined by the order of weaving and lexical ordering respectively.

An example aspect is shown in Figure 5.1. This aspect counts the number of zero-valued operands of add and subtract expressions. Lines 1–4 form the pointcut (with a comment on line 3), while the actual advice body is given by lines 7–8. The use of LMP makes the pointcut robust to changes in the base code, as the pointcut is expressed in terms of program structure (accessed via selectors like `RECEIVER`) and join point context can be bound freely using `BIND`. If there are multiple solutions for some selector, the weaver backtracks and finds all matches instead of choosing only one. In the current example, the pointcut matches for both operands `A` and `B` of a statement like `ADD A B TO C`. The advice body is also robust to variability of join points because it can access the context captured via the pointcut (`VAR-ITEM`). Any statement which has a sender can be matched (line 2), and the advice is able to adapt to all possible statements. The captured context also enable access to variables.

Cobble is the only aspect language for Cobol we consider. All other related languages target C or C++ as base programming language.

```
1 pointcut allocating_buffers(vm_object_t obj,
2                              vm_pindex_t pindex):
3  execution(vm_page_t vm_page_lookup(obj, pindex))
4  && cflow(execution(int allocbuf(struct buf*, int)));
5
6 around(vm_object_t obj, vm_pindex_t pindex):
7  allocating_buffers(obj, pindex){
8    vm_page_t m = proceed(obj, pindex);
9    if ((m != NULL) && !(m->flags & PG_BUSY)
10      && ((m->queue - m->pc) == PQ_CACHE)
11      && (pages_available() < vfs_page_threshold()))
12     pagedaemon_wakeup();
13   return m;
14 }
```

Figure 5.2: AspectC aspect for page daemon wake-up in the FreeBSD kernel [49].

5.2.1.2 AspectC

AspectC has been the first aspect language for C, inspired by AspectJ's constructs. Figure 5.2 illustrates this, as familiar `pointcut`, `execution`, `cflow` and `proceed`[7] constructs are used. There is no explicit `aspect` construct, as file boundaries are used for this. The join point model is similar to AspectJ-without-objects. This means that procedure calls (`call`) and executions[8] (`execution`) can be advised. The dynamic `cflow` pointcut selects all join points which occur in the control flow of another join point. Contrary to AspectJ, variable accesses cannot be advised and there is no ITD support either.

The advice signature does not specify a return type (line 6). Instead, the aspect developer should determine the right return type when it is needed in the advice body, e.g. to declare local variables (line 8). Regular expressions cannot be used either in pointcuts, which means that for every possible return type a separate pointcut *and* advice has to be written. This has caused the problems of Bruntink et al. discussed in Section 5.1.1. Access to typed context is possible (line 6). Initially [50], aspects were hand-compiled, but later on [49], a real weaver has been built. AspectC seems unmaintained since 2003 without any official releases, but a weaver prototype has been available by request.

```
1  aspect ThrowWin32Errors{
2
3    pointcut Win32API() =  "% CreateWindow%(...)"
4                         || "% BeginPaint(...)"
5                         || "% CreateFile%(...)"
6                         || ...
7    ...
8    advice call(Win32API()): after () {
9     if(win32::IsErrorResult(*tjp->result())){
10      ostringstream os;
11      DWORD code=GetLastError();
12
13      os << "WIN32 ERROR " << code << " : "
14         << win32::GetErrorText(code) << endl;
15      os << "WHILE CALLING: "
16         << tjp->signature( << endl;
17      os << "WITH: " << "(";
18
19      // Generate join point-specific sequence of
20      // operations to stream all argument values
21      stream_params<JoinPoint,
22                    JoinPoint::ARGS>::process(os,tjp);
23      os << ")";
24      throw win32::Exception(os.str(),code);
25    }
26   }
27 }
```

Figure 5.3: AspectC++ aspect which converts return value error codes into C++ exceptions [214].

5.2.1.3 AspectC++

AspectC++ [213, 214] is the most mature and general-purpose aspect language for C++ to date, but since its inception people have tried to use it for C too. Official support for this has never been a priority, however. Non-ANSI C code (so-called "K&R"-code) cannot be parsed by the weaver. It is also not clear which constructs and pointcuts can be used for C and which ones cannot. The AspectC++ weaver is heavily based on template instantiation to reduce memory footprint and execution time. Hence, the woven code is valid C++ which needs a modern C++ compiler. There is a (commercial) IDE plugin.

[5] `http://www.tiobe.com/`

[6] `http://homepages.vub.ac.be/~kdeschut/cobble/`

[7] This enables to resume the advised join point from within an `around`-advice.

[8] A `call` join point corresponds to the actual call at the caller site, whereas an `execution` join point denotes a call at the callee site.

As Figure 5.3 shows, AspectC++ is heavily influenced by AspectJ. There is an explicit `aspect` construct which is similar to a C++ class, hence an aspect needs to be declared inside a special "aspect header file". Join point, advice and pointcut types are comparable to AspectJ. Contrary to AspectJ, advice and inter-type declarations ("slices") can be specified in the same way. Because of this, the join point model is said to be "unified". Join points are implicitly typed, such that the weaver may check that they are only advised by correctly typed advice. Regular expressions can be used to specify pointcuts (lines 3–6 on Figure 5.3). Two new pointcut types are provided. The `callsto` pointcut takes an `execution` pointcut and deduces which call join points can call the join points described by the `execution` pointcut. The `reachable` pointcut is analogous, but it calculates (via static analysis) all join points from which its argument join points can be reached. There are no `set` and `get` pointcuts, i.e. access to variables is not reified as a join point, because of aliasing problems introduced by pointers and because of the unsound semantics of `set` regarding `operator=`. Just like AspectJ, there are precedence directives to derive a total order between aspects. Advice ordering within an aspect is ordered via lexical conventions.

AspectC++ has coined the term "generic advice" [159] to refer to the powerful capabilities of templates for obtaining highly reusable and robust advice. The idea is that AspectJ's distinction between static and dynamic context provided by a `thisJoinPoint`-like construct is generalised to C++'s strong compile-time templating mechanism. Compile-time context can be used to instantiate templated advice and functions, such that there is no run-time overhead to dynamically allocate or access context. Lines 21–22 of Figure 5.3 gives an example of this. Because `JoinPoint` is just a class name and `JoinPoint::ARGS` statically resolves to the correct number of function arguments of the advised call join point, the `stream_params<JoinPoint,JoinPoint::ARGS>` template is instantiated at compile-time using template meta-programming. This mechanism allows for very reusable and robust advice, which varies automatically based on the particular join point and advice context. As a downside, the templates can easily get very complex to understand, especially for C programmers which are used to the simpler semantics of the C preprocessor.

AspectC++'s weaver[9] is based on the PUMA-framework, a C++ source code transformation system [213] with a custom C preprocessor implementation. Aspects are transformed into singleton classes in which advice corresponds to a member function. Generic advice is turned into a template member function with a specific JoinPoint type as template parameter. Each join point shadow is replaced by a call to a wrapper member of a local class which is specifically generated for that shadow and encodes the chain of advice which matches at that shadow. The implementation of advised functions is replaced by a template wrapper, i.e.

[9] `http://www.aspectc.org/`

```
<pointcut name="targetFloatNameGetter" type="src:name"
      constraint="(text()='_flt') and
       (not(contains(following-sibling::text()[1],'=')) or
       (contains(following-sibling::text()[1],'==')))">
 <restriction type="within">
  <pointcutRef ref="anySampleClassExpr" type="src:expr"/>
 </restriction>
</pointcut>

<advice name="targetFloatNameGetter" type="replace">
 <pointcutRef ref="targetFloatNameGetter"
              aspect="PointcutLibrary" type="src:name"/>
 <codeModifier type="codeFragment">
  <xsl>pDB-&gt;getParameterFloat(PD<xsl:value-of
       select="upper-case(current())"/>)</xsl>
 </codeModifier>
</advice>
```

Figure 5.4: Accesses to a float member are replaced by the result of a method call with XWeaver (see `http://www.xweaver.org/`).

a template which inherits from a class and wraps superclass methods with advice functionality. As a result, the overhead of advice invocation is constant, just as the size of woven code. Possible compiler optimisations include inlining of code and removing spurious parameter copies via local alias analysis.

Gilani et al. [98] extend the AspectC++ weaver with a family-based dynamic weaver implementation. They decompose aspect language and weaver features into a product family with associated time and memory costs. Depending on the application domain, features can be stripped or added to decrease or increase the weaving cost. AspectC++ advice simulates dynamic weaving by adding calls to a run-time monitor which determines whether a registered aspect matches the current join point. The dynamic aspects are not described within AspectC++, but implemented in an OO framework instead. Aspects are loaded via a dynamic library at run-time.

5.2.1.4 AspectX/XWeaver

XWeaver[10] [197] is the name of the aspect weaver associated with the AspectX aspect language. It is conceived for tailoring software frameworks to control systems. As quality control is important for this, XWeaver's task is to generate woven code which syntactically resembles the base code layout and even updates comments (to document the woven code) such that the woven code can be manually

[10] `http://www.xweaver.org/`

investigated. XWeaver does not work on the program AST, but on srcML. This is an XML representation of a program in which only high-level program constructs are accessible. Comments and include/import statements are retained. This format makes XWeaver language-independent, in the sense that initial C++ support has been extended to Java once srcML was released for Java. Just as is the case with AspectC++, XWeaver can be used for C systems too.

The AspectX language is also XML-based, as the advice in Figure 5.4 shows. XML Schema is used to type-check the syntax of the XML aspects. AspectX allows the usage of XPath and XSLT technologies in the pointcut (lines 2–4) and advice (lines 14–15) respectively. XPath is able to select the right XML nodes by navigating across the XML tree. In the example, nodes of type `float` are selected (line 2) which do not occur as the left-hand side of an assignment or "equals" condition (lines 3–4). The advice of lines 10–17 replaces (line 10) the selected elements using the XSLT transformation of lines 14–15. This transformation capitalises the name of the advised join point XML node. Hence, the user should have considerable knowledge of the program XML-model. Special symbols need to be escaped, as the "`>`" on line 14 shows. Inclusion of XML documents can be exploited to reuse a library of pointcuts. To summarise, the AspectX language is a very low-level aspect language which resides on the border with pure program transformation.

Join point context (argument types/names, return types, etc.) is accessible via dollar-variables like `${className}`. Under the hood, XWeaver transforms aspects in XSLT transformations, which means that join point context actually corresponds to an XSLT query. Hence, users can extend XWeaver with new context queries. New join point types can be added in a similar manner, and they can be very fine-grained like e.g. if-blocks, return-statements, etc. More traditional join points like `execution` exist, but all of them are purely statically determined based on the AST. There are no provisions for dynamic join points. On the other hand, the focus on program transformation enables syntactic ITD of comments and even include/import statements.

XWeaver is implemented in Java. There is an Eclipse plugin (AXDT) akin to the AspectJ AJDT, but command line or build script access (via Ant) is also possible. XWeaver can generate an Ant file based on a project file (XML). The latter specifies the important directories in the project and also the aspect configuration per subset of base code modules and header files.

5.2.1.5 C4

C4[11] (CrossCutting C Code) [93] aims at replacing the traditional "patch" system by a simplified AOP-driven, semantic approach. Basically, the idea is that a pro-

[11] `http://c4.cs.princeton.edu/`

```
1 void func(int var, char *str) {
2   aspect_around(foobar) {
3     printf("Inside around-advice\n");
4     if(var == 20) proceed(20,"Twenty");
5     else proceed(0,"ZERO");
6   };
7
8   ... /* Function body */
9 }
```

Figure 5.5: C4 aspect which overrides a function body [245].

grammer writes down advice (so-called "woven C4") inline with the base program, which means that there are no pointcuts or separate aspect modules. Figure 5.5 (lines 2–6) shows `around`-advice of the `foobar` aspect. Access to join point context is easy, because the advice can reuse variables available in the surrounding base code.

The C4 unweaver extracts the inline aspects into a separate "unwoven C4" file, i.e. a semantic patch. This can be freely distributed to everyone or (if needed) converted to a plain patch first. The unwoven C4 can be re-woven by other developers to resume development with the latest changes. At compile-time, the unwoven C4 is physically woven with the base code, i.e. all aspect constructs are transformed into pure C code. Physical weaving is implemented on top of the XTC-framework [104], an advanced macro facility for C. To deal with C preprocessor code, developers do not work on the actual C code, but on an intermediate representation called ASTEC [166, 245]. This reifies preprocessor constructs in a slightly higher-level representation.

The unwoven C4 file is in fact a (tweakable) classic aspect written in an AspectC-based dialect. This dialect is capable of ITD in `structs`/`unions` and can also advise global variables. There are no `call` or `cflow` pointcuts in the dialect, primarily because of C function pointers. Instead, one is encouraged to extract base code into new procedures, which can be advised directly.

5.2.1.6 WeaveC

WeaveC[12] [75] is an aspect language developed within the Ideals project as a prototype of the Mirjam [179] aspect language (discussed in Section 5.3.6). Just like AspectX (Section 5.2.1.4), pointcuts and advice are written in XML-files. WeaveC has the same join point and advice types as AspectC, but no `around`-advice. Pointcuts are name-based (which may contain wildcards), and can be restricted in their scope ("*.c" on line 4 of Figure 5.6). Dynamic pointcut types like `cflow`

[12] `http://weavec.sourceforge.net/`

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <aspect id="insert_f_name">
3  <pointcut id="insert function names">
4   <elements files="*.c" identifier="function" data=".*"/>
5   <advices>
6    <adviceapplication id="insert_f_name" type="before"/>
7   </advices>
8  </pointcut>
9  <advice id="insert_f_name"
10       type="function_variable_introduction" priority="200">
11  <code>
12   <![CDATA[ static const char* f_name = "%FUNC_NAME%"; ]]>
13  </code>
14 </advice>
15 </aspect>
```

Figure 5.6: Introduction of a static local variable in WeaveC (example suggested by Pascal Durr).

are only provided in an advanced version of WeaveC[13]. WeaveC can introduce new local and global variables. The latter is illustrated in the snippet of Figure 5.6 (line 12). Line 10 shows that advice is prioritised to handle conflicts at shared join points.

A number of powerful, predefined context variables like `%MODULE_NAME%` are available which represent the advised function's module name, argument types, etc. Variables which appear near the join point shadow can be used freely in the advice body. It is not clear from the documentation whether truly generic advice can be written by using these context variables. WeaveC has limited support for annotations.

WeaveC's weaver is implemented in Java, and transforms an XML-representation of the AST of the preprocessed base program. In the advanced version, CodeSurfer[14] is used to perform the necessary analyses.

5.2.1.7 ACC

ACC[15] [102] is the most regularly released aspect language for C, designed by the people behind TinyC[2] (Section 5.2.2.2). It is in fact an updated version of the original AspectC (Section 5.2.1.2), but it sports a number of improvements. ACC e.g. adds an AspectJ-like thisJoinPoint-struct (`this`) to provide join point context like the names of argument types (lines 8, 10 and 12) or to provide access to func-

[13] The advanced version has never been released.
[14] `http://www.grammatech.com/products/codesurfer/`
[15] `http://www.aspectc.net/`

```
1 before(): call($ $(...)) {
2   printf("%s \" %s \" in function %s \n", this->kind,
3           this->targetName, this->funcName);
4
5   if(this->argsCount == 0) printf("no parameter \n");
6   else{
7     for(int i = 1 ; i <= this->argsCount; i++){
8       printf("arg[%d] = %s  ", i, this->argType(i));
9
10      if(strcmp(this->argType(i), "int") == 0){
11        printf(", value = %d ", *(int *)(this->arg(i)));
12      }else if(strcmp(this->argType(i), "double") == 0){
13        printf(", value = %.2f ",
14                *(double *)(this->arg(i)));
15      }
16
17      printf("\n");
18    }
19  }
20
21  printf("return type = %s \n \n", this->retType);
22 }
```

Figure 5.7: ACC tracing advice (see http://research.msrg.utoronto.ca/ACC/Examples/).

tion arguments (lines 11 and 13–14). Arguments passed to `around`-advice cannot be changed directly by modifying the captured context. Instead, the `this` struct contains pointers to the advised function's arguments through which the original arguments can be modified prior to calling `proceed`. The `result` pointcut should be used to modify the return value of a join point.

ACC provides a number of new pointcuts like `callp` (function pointer calls), `infunc`, `infile` and `intype`, and there are `get`/`set` join points for advising accesses to global variables (not of a `struct` type). New members can be introduced into `struct` and `union` types. Additionally, ACC provides exception handling aspects, which are based on a special pointcut for exception handling called `preturn(errorvalue)`. This construct is based on the delimited continuation join points we have proposed [8] (more on this in Chapter 8). Basically, `preturn(errorvalue)` returns from the enclosing procedure of a `call` join point shadow with the provided error value `errorvalue` as return value.

The ACC weaver is based on an ANSI-C compiler (implemented in C), and it supports K&R-, ANSI- and GNU-C. It expects preprocessed aspects and base code as input. Each advice is transformed into a unique function with as arguments the context parameters. There is a thread-safe `cflow`-implementation based on GCC

```
1 prevent_propagation :[
2   int handle_request(char* req) :[
3     int relay_request(struct req_data* request) :[
4       { #include ``prefetching.h''
5         if(is_prefetching_request(request)){
6           return NO_NEIGHBOUR_HAS_FILE;
7         }
8
9         return continue_relay_request(request);
10      }
11    ]
12  ]
13 ]
```

Figure 5.8: Prefetching policy aspect in μDiner [203].

extensions for thread-local state. Also, `#line`-directives for debugging can be generated, to facilitate stepping through base and advice code. Powerful weaver conventions and build system drivers have been developed to improve weaver usage.

5.2.2 Aspect Languages with a Run-time Weaver

The biggest group of aspect languages for C features a compile-time weaver, primarily because the typical domains where C shines (systems software!) require highly efficient woven code. However, these systems have other desirable properties too, such as availability and debuggability. These are the application areas run-time weavers can be beneficial [190] for.

All aspect languages discussed in this section have a weaver based on instrumentation libraries or techniques like code splicing [78], i.e. tweaking of assembler code to jump to the aspect code instead of the advised base code. As the run-time weavers process binary code, they do not need to parse the actual source code, except for μDiner. The platform-independence of the instrumentation mechanisms used by these weavers is questionable, however. There is also no opportunity to optimise the advised application after the dynamic weaving, such that the woven systems tend to become patch works. All dynamic approaches provide some sort of `around`-advice or a combination of `before` and `after`. They all support procedure call join points. Typically, there is not enough context information available at the binary level. Writing generic advice is impossible, hence even slightly heterogeneous advice results in duplication of advice.

```
1 onexit int retv group * :
2  (int errorno, char* errormsg){
3   if ( retv < 0 ){
4     errorno=ILLEGAL RESULT;
5     errormsg="Result cannot be negative";
6   }
7  }
```

Figure 5.9: Checking return values using TinyC^2 [249].

5.2.2.1 μDiner

μDiner is the sole aspect language with a run-time weaver which needs to parse the base code. It requires base code procedures to be annotated as "hookable" (which expands to `volatile`) to mark them as advisable [203]. A support library needs to be linked with the resulting executable.

In return for this slight inconvenience (one needs to mark in advance which join points can dynamically be advised later on), μDiner offers an advanced kind of pointcut which selects a sequence of nested procedure call join points. As an example, Figure 5.8 shows an advice named "prevent_propagation" which only matches on a call to a procedure `relay_request` (line 3) if it is called from within a call to `handle_request` (line 2). The advice body (lines 4–9) is executed around the matched call join point. The procedure call on line 9 corresponds to a `proceed`-call in AspectJ. The `continue_nameoffunction`-function pointer is automatically defined for a matched join point. Within the advice body, one can access arguments and global variables. Variable access join points (`get` and `set`) can also be advised.

5.2.2.2 TinyC^2

TinyC^2 relies on the DynInst-instrumentation library [249]. An aspect is transformed into a self-contained C application which controls the instrumentation of a running base program through the DynInst API. The pointcut language supports name patterns, as shown on line 1 of Figure 5.9. Instead of `before`- and `after`-advice, TinyC^2 talks about `onentry` and `onexit`, but there is no `around`-advice. Except for arguments and global variables, there is no join point context available.

```
1 seq( call(void* malloc(size t)) && args(allocatedSize)
2                                 && return(buffer);
3      write(buffer) && size(writtenSize)
4                    && if(writtenSize > allocatedSize)
5       then reportOverflow(); *
6      call(void free(void*)) && args(buffer); )
```

Figure 5.10: Buffer overflow detection aspect in Arachne [68].

5.2.2.3 Arachne

Arachne[16] improves on the μDiner framework [68]. The pointcut language has been reworked, inspired by Prolog. The sequence of nested calls has been generalised to an arbitrary sequence of procedure call and/or (in)direct variable access join points, whether or not they are nested. The resulting `sequence` pointcut is a natural means for advising protocol-like behaviour, as each element of the sequence can be advised individually. The advice of Figure 5.10 detects when more data is written into a heap-allocated buffer (lines 3–4) than initially allocated (lines 1–2). In that case, overflow is reported (line 5). The sequence ends when the buffer is deallocated (line 6). The latter is required to avoid that further run-time checks are performed for the buffer allocated at that address.

To increase the expressivity of this language, Loriant et al. [164] later have added the possibility to bind specific context to each instance of a `sequence`. Also, a `fakeEvent` construct has been introduced to simulate calls to an arbitrary procedure when some join point matches. These fake events can then trigger other advice of which the pointcut is expressed in terms of that event.

Clever assembler manipulation techniques are used to instrument a running system without having to pause it. Despite claims of robustness across computer architectures, these techniques did not work on the various test machines we have tested Arachne on[17]. The last public release of Arachne dates back to March 2005.

5.2.2.4 TOSKANA (Toolkit for Operating System Kernel Aspects with Nice Applications)

TOSKANA [78] is targeted specifically at C code running in the kernel mode of BSD-like operating systems. As a consequence, the aspect language has been kept as simple as possible. It e.g. lacks name patterns and abundant join point context. Figure 5.11 shows an example aspect which tries to recover (lines 11–16) from a particular error. On line 6, the `map_aspect` is woven onto the `uvm_map` function. The aspect is deployed explicitly onto the desired join points, as there

[16] `http://www.emn.fr/x-info/arachne/`
[17] Arachne is distributed as a live Linux distribution.

```
1  #include <aspects.h>
2
3  int error;
4
5  void aspect_init(void){
6    AROUND(uvm_map, map_aspect);
7  }
8
9  ASPECT map_aspect(void){
10   error = PROCEED();
11   if (error == ENOMEM) {
12     /* no memory available */
13     kernel_alloc_swapfile(SWAPFILE_SIZE);
14     /* try again */
15     error = PROCEED();
16   }
17
18   return error;
19 }
```

Figure 5.11: Self-healing aspect in TOSKANA [78].

are no pointcuts.

TOSKANA's weaver applies a weaving strategy (named "code splicing") which is similar to the one Arachne uses. There are no public releases of TOSKANA.

5.2.2.5 KLASY (Kernel Level Aspect-oriented SYstem)

KLASY is analogous to TOSKANA, but is targeted at the Linux kernel instead. It allows to advise `struct` member access (e.g. to function pointers) and procedure calls. Available join point context includes access to `struct`s, local variables and arguments [244]. There is no `around`-advice. As shown on Figure 5.12, aspects are XML files. Just like with AspectX (Section 5.2.1.4), special symbols have to be escaped (line 14). Data can be passed from the advice to a user land application via the `STORE_DATA` macro. Notice the special `$pc$` variable on line 13 which gives direct access to the program counter.

KLASY needs a modified GCC compiler to generate symbol information during the weaving process, and the "kerninst" Linux kernel module and daemon to control the dynamic weaving (via a socket). Aspects are compiled into standalone applications which steer the "kerninst" daemon.

```
<aspect>
 <import>linux/sched.h</import>
 <import>asm/page.h</import>
 <advice>
  <pointcut>
   access(task_struct.timestamp) AND
   within_file(sched.c) AND target(arg0)
  </pointcut>
  <before>
   struct task_struct *p = (struct task_struct *)arg0;
   unsigned long long timestamp;
   DO_RDTSC(timestamp);
   STORE_DATA($pc$);
   STORE_DATA(p-&gt;pid);
   STORE_DATA(timestamp);
  </before>
 </advice>
</aspect>
```

Figure 5.12: Process switch tracing aspect in KLASY [244].

5.2.3 Aspect Language with a Virtual Machine Weaver

TOSKANA-VM [79] is a proposal of the researchers behind TOSKANA for an aspect weaver based on a virtual machine running on top of LLVM (Low-Level Virtual Machine). This is a compiler framework with a universal IR and life-long analysis facilities. This infrastructure would be capable of optimising the dynamically woven system based on profiling date. At the same time, the IR is richer in information than assembler code, so the amount of context available at join points would be larger.

TOSKANA-VM's aspect language has not been published, so there are many question marks in Table 5.2. Only some details about the weaver have been published. A micro-kernel (L4) runs directly on hardware, with on top of it a weaver and various LLVM instances. The weaver can monitor the VM instances and dynamically (re)weave the operating systems or user applications which run within the VM instances. No implementation of TOSKANA-VM has ever been released.

5.2.4 Model Driven Weaving

WEAVR[18] [53] is an MDE-system of Motorola which provides an AOP language at the model level, i.e. in terms of UML 2.0 behavioural models. Join points correspond to state transitions in state machines. Weaving corresponds to trans-

[18] http://www.iit.edu/~concern/weavr/

formation of these state machine models. After aspects have been woven in the model, C code can be generated from it. In other words, this aspect language weaves aspects in C systems via a model of the system. In fact, the whole system is generated from that model. WEAVR is especially targeted at checkpointing, tracing, timing, etc.

5.2.5 Evaluation

Given the comparison of features in Table 5.1 and Table 5.2, we now evaluate the degree to which the existing aspect languages satisfy goals L1 and L2 presented in Section 5.1.1 and Section 5.1.2 respectively.

First, for the base code integration requirement of goal L1, we have observed that most aspect languages blend well with the base code, with C4 as an extreme case. Other aspect languages require a custom XML format (AspectX, WeaveC and KLASY). Second, regular expressions are the most widespread mechanism to obtain expressive pointcuts. AspectX and Arachne are more powerful because of their syntactic program transformation and `sequence` pointcuts respectively. Third, except for Cobble and AspectC++, the aspect languages obtain some form of generic advice especially by allowing access to a rich set of join point variables in various ways inside the advice body. Cobble and AspectC++ on the other hand respectively rely on LMP and C++ templates instead. They do not require developers to change the weaver implementation to add extra context for obtaining more expressive pointcuts. Fourth, aspect languages with generic advice all have access to a wealth of join point context, except for TinyC^2. Conversely, ACC does not enable a form of generic advice, but offers access to a rich set of join point context. Overall, AspectC++, AspectX and Arachne conceptually are the best aspect languages for C based on L1.

However, if we consider the integration of the build system with the aspect language, we observe that only Cobble and C4 enable integration of the build system with the aspect language. C4 enables this in an ad hoc way, as aspects are specified inline with the base code. Cobble in principle enables integration by asserting build system information as weave-time meta data, but facts can only be specified inside the aspect. Hence, no aspect language for C fundamentally supports integration of the build system with its pointcuts or advices.

Because AspectC++ does not fully support C (e.g. no K&R parser) and generates C++ code instead of C, AspectX reasons in terms of XML transformations instead of in terms of join points, and Arachne does not provide generic advice and is not platform-independent, none of the languages which fulfill L1 are really suited for the legacy system environment we are targeting. Because they in addition do not satisfy L2 either, we have decided to design and implement a new aspect language for C, i.e. Aspicere. As Cobble's features fare well in the comparison with

	AspectC++	Arachne	Aspicere1	Aspicere2	Mirjam
domain	general	systems	general	general	general
base integration	.h	+	+	+	+
preprocessor	-	-	-	-	-
PCD robustness	regexp	+	LMP	LMP	relational
(function) pointers	-	-	-	-	?
ITD	+	-	-	properties	-
basic join points	AspectJ	+	+	+	+
dynamic join points	AspectJ	+	-	+	-
advanced join points	`callsto/ reachable`	+	-	cHALO	-
variable access	-	+	-	-	-
generic advice	+	-	+	+	+
aspect interaction	explicit	deployment	build	build	build?
advice interaction	lexical	lexical	lexical	lexical	order
context	+	+	+	+	+
thisJoinPoint	+	-	+	+	+
annotations	-	-	-	+	+
build integration	-	N/A	+	+	+
availability	+	+	+	+	-
weaver type	source	run-time	source	link-time	source
optimisation	+	-	-	+	?
K&R support	-	N/A	+	+	?
IDE	+	-	-	-	?

Table 5.3: Comparison between the two incarnations of Aspicere and the most related aspect languages (taken from Table 5.1 and Table 5.2). Mirjam is discussed in Section 5.3.6. The upper four sets of rows correspond to provisions for the four requirements of goal L1, the next one for goal L2, and the lowest rows characterise the languages' weaver. A +/- indicates good/bad support for some feature, whereas "N/A" signals when an entry is not applicable to some language. Because every language supports access to function arguments and global variables, a "-" for "context" means that there is no additional means for access to context.

the other aspect languages, and because it intrinsically supports integration of the build system with the aspect language, we have based Aspicere on Cobble. The next section presents the design of Aspicere.

5.3 Language Design of Aspicere

Aspicere [248] is an aspectual extension to C developed upon the same foundation as Cobble [142]. This section discusses the basic language specification of Aspicere, without taking into account the more advanced features which are discussed during the case studies (in later chapters). Table 5.3 relates Aspicere to the most expressive and powerful aspect languages for C described in Section 5.2, or

presented later on in Section 5.3.6. Aspicere has two weavers, i.e. Aspicere1[19] and Aspicere2[20]. The Aspicere1 weaver does not support some basic features of Aspicere and none of the advanced ones.

We first briefly summarise how Aspicere deals with goals L1 and L2 (Section 5.3.1). Then, we elaborate on Aspicere's join point model (Section 5.3.2), advice structure (Section 5.3.3) and pointcut language (Section 5.3.4). We conclude this section with an example which combines all basic language features (Section 5.3.5). Later chapters provide more advanced applications. Finally, Section 5.3.6 evaluates Aspicere's features with respect to those of a recently proposed aspect language for industrial C systems, i.e. Mirjam. We defer the presentation of the implementation of the two Aspicere weavers to Section 5.4.

5.3.1 How Aspicere Deals with Goals L1 and L2

Aspicere deals with the four requirements of L1 and with the integration of the build system with the aspect language (L2).

First, integration with the base language is achieved by only introducing a dedicated advice and ITD structure. Aspects correspond to normal C modules with the same visibility rules for global and `static` variables and procedures. The second requirement, an expressive pointcut language, is satisfied by choosing Prolog as the underlying language for pointcut specifications, akin to Gybels et al. [108]. Arachne is the only other aspect language for C which does something similar, but in a more restricted way. As Prolog is a logic programming language, it satisfies the pointcut language characteristics suggested by Gybels et al. Pointcuts are composed of various Prolog predicates. These can either be defined in Prolog files (libraries), or inline in the advice definition. To reconcile Prolog's expressiveness with the first L1 requirement, inline pointcut definitions within an advice can make use of well-known conditional operators like "&&", "||" and "!!" instead of ",", ";" and "\+".

The third L1 requirement is fulfilled by enabling context bound during join point matching by the advice's pointcut to be used either as normal procedure arguments or instead as a kind of C++-like template parameter to represent a type[21]. Aspicere1 supports even more powerful template parameters (e.g. to represent a statement), but we have not yet pursued this further. The available join point context (fourth requirement) either corresponds to Prolog facts which model the program structure, or weave-time metadata, similar to Cobble [142, 202].

Weave-time meta data is also the key to integrate the build system with the aspect language (goal L2). It suffices to assert (parts of) the build structure and

[19] `http://users.ugent.be/~badams/aspicere/`

[20] `http://users.ugent.be/~badams/aspicere2/`

[21] Note that besides the fact that Aspicere's template parameters can abstract over types, there are no other similarities with C++ templates. More advanced features like template meta-programming are overkill, and would violate the first L1 requirement.

configuration as logic facts before the actual weaving starts. Pointcuts can freely make use of this meta data, and hence reason about the high-level program architecture or configured features.

To summarise, Aspicere fulfills all requirements of L1 and L2. The remainder of this chapter elaborates on the design of the Aspicere language and on the two existing aspect weavers for it.

5.3.2 Aspicere's Join Point Model

This sections first presents the join points which are supported by Aspicere, and afterwards the provisions for static crosscutting.

5.3.2.1 Supported Join Points

When introducing AOP-techniques into a base language, the first thing to consider is how the base language and the new aspect language should interact. Because C is a procedural programming language, procedures are the ideal abstraction for join points. Just as in AspectJ-like languages, a distinction is made between `call` and `execution` join points. However, because C modules are not mapped to a type, the concepts of `this` (caller) and `target` (callee) do not really make sense.

There are various reasons for distinguishing between `call` and `execution`. One should advise an `execution` join point to make sure that a function is always, except for a limited number of cases, advised by some aspect. If on the other hand only in a minority of cases a function should be advised, a `call` join point is preferable. This is not just a technical argument based on weaver implementation details, as it has other implications. If one wants to advise a library, choosing an `execution` join point will weave the advice into the library itself, making it active anywhere the library is linked with an application. However, source-to-source weavers cannot advise compiled code. They can only advise `call` join points which access the library. This restricts the scope of the aspect to that application, i.e. the advice does not transfer to other systems in which the library is used.

Polymorphism is a second area where the difference between `call` and `execution` becomes visible. In procedural languages like C, polymorphism is obtained via function pointers. These correspond to addresses of functions which can freely be be passed as arguments or invoked. Pointers are generally hard to reason about because of "aliasing". A given (global) variable or function can be accessed directly or via some pointer to it. Because pointer variables can be freely passed as procedure arguments, cast to other types or assigned to other pointers, it becomes impossible to statically predict in all cases which variable is accessed or which function invoked by looking at a (function) pointer.

Aspect languages can do four things to deal with function pointers:

- ignore them;

- treat them like normal procedure calls;
- treat them specially;
- handle them indirectly via `execution` join points.

The first approach is not desirable, as in many systems function pointers are heavily used. The second one seems to be the best one at first sight, but here it is important to take into account the efficiency of the woven code. Whichever weaving implementation is chosen, only run-time checks can determine accurately the actual method which is called, even in the presence of advanced points-to analysis. It is better to give developers more control over this, such that they explicitly can choose to avoid this performance penalty at the expense of less precision. This corresponds to the third approach, as chosen e.g. by ACC (Section 5.2.1.7). Here, a dedicated `callp` pointcut for function pointer calls can be used to ensure that every function pointer is checked at run-time.

However, even without this dedicated support, there is a simple way to avoid overhead while giving up only some expressivity: advising the `execution` join point instead of the function pointer `call`. Indeed, advice on the `execution` join point ensures that every execution of a function is advised, whether or not it has been invoked via a function pointer, but without run-time check overhead. In practice, we have not encountered cases where the reduced expressivity is a real problem. Hence, Aspicere does not have a `callp`-like join point, but instead uses `execution` join points.

Join points for access to variables suffer from a similar problem as the function pointers. However, there are no real alternatives to handle them. Arachne (Section 5.2.2.3) e.g. uses the operating system's page fault mechanism to detect variable access, but this causes extreme performance penalties. Similar to AspectC++ (Section 5.2.1.3), Aspicere does not support variable access join points.

The last kind of behavioural join point is the "delimited continuation join point". This is an advanced kind of join point which abstracts over the remaining execution of a procedure after a join point which has been directly invoked by that procedure, has returned. As such, `around`-advice on a delimited continuation join point can control whether the enclosing procedure continues its execution or not by judicious use of a `proceed`-call inside the advice. For didactic purposes, a more detailed explanation of delimited continuation join points and an application of them is presented in the third case study (Chapter 8 on page 223). Note that Aspicere's continuation join points differ from those of Endoh et al. [77]. These researchers use the same term to refer to a formalisation technique used for defining the fine-grained semantics of their join point model. We, instead, use the term in the sense of an aspect language concept.

Aspicere does not consider the following constructs as valid join points:

macro expansion The lack of type checking and the problems to easily parse preprocessor code makes macro expansions undesirable as join points. We do believe that replacing macros and other preprocessor constructs by aspects makes more sense. This statement is revisited in Chapter 10.

inline assembler code This is definitely too platform-dependent and low-level.

5.3.2.2 Join Point Properties for ITD

For static crosscutting, i.e. ITD, a literal translation of AspectJ would require introduction of new members to C `structs`, `unions` and maybe also `enums`. This is not straightforward in realistic systems, because one needs to make sure that no base code depends on the memory size of a `struct` or `union`, as this is changed by the ITD, and that the whole system uses the modified (advised) data structure instead of the original types.

Aspicere treats ITD differently. The primary reason why ITD is part of AspectJ, is because some crosscutting concerns can only be modeled cleanly when extra state and behaviour can be associated with participating objects. As these objects are passed through the system, they transport this state and behaviour to other join points where advice can access them. A different way to interpret this, is to associate state[22] with a join point and to relate different join points with each other, if needed, to make this state accessible to the other join points.

Aspicere follows this approach by using join point properties [178]. These are (name,value)-pairs attached to individual join points. The familiar `thisJoinPoint`-object of AspectJ can actually be interpreted as a container of system-provided join point properties. Nagy et al. [178] propose to let the user add custom properties to communicate between various advices and, even more importantly, between independently developed aspects. This eases advice interactions, and, if the properties are thread-local, it fits nicely with multi-threading environments too. The main benefit for C is that the extra state can be completely managed by aspects, i.e. the base code is oblivious to them. Hence, the introduced state is much more manageable, and can be implemented efficiently (more on this later).

A prerequisite for this approach to work is that relations between join points can be easily expressed. Aspicere's expressive logic pointcut language and its temporal extension (discussed in Chapter 9) enable this. This is also how Arachne accomplishes ITD. Arachne associates state to a particular sequence of join points via its `bind` construct [164]. This is less general than join point properties, as the latter are accessible to any advice of which the pointcut matches a join point with a property attached to it.

To summarise, the join point model of Aspicere is a compromise between language expression power, ease of integration in the build system and customisation

[22] In C, ITD of behaviour does not make sense, because `structs` only contain state.

```
1 ReturnType safe_ato(TYPE ReturnType,char* Src) around Jp:
2   invocation(Jp,"ato.")
3   && args(Jp,[Src]))
4   && type(Jp,ReturnType){
5     ReturnType dst;
6
7     if(Src == NULL) dst = 0; /* compiler does the cast */
8     else dst=proceed();
9
10    return dst;
11 }
```

Figure 5.13: Aspicere aspect to make standard conversion from strings to numbers null pointer-proof.

to C's typical application areas, in which efficiency and performance play an important role. The next section presents Aspicere's advice model.

5.3.3 Aspicere's Advice Model

As explained in Section 5.3.1, Aspicere aspects correspond to ordinary C modules with a special advice and join point property construct. These constructs are described in the two following subsections.

5.3.3.1 Advice Structure

Figure 5.13 shows an example advice. It secures calls to the standard `atoi`, `atol` and `atof` procedures to prevent a program from crashing when a null pointer is passed as an argument. These three procedures parse a string (`char*`) argument into an `int`, `long` or `double`. One advice suffices to advise all three procedures because of the use of template parameter `ReturnType` (lines 1 and 5). This advice is used later on in the first case study (Section 6.2).

An advice structure specifies:

- the advice return type (useless in case of `before`- or `after`-advice);
- the name of the advice;
- a list of bound context variables[23] which are visible to the advice body;
- the type of advice (`before`, `around` and `after (returning)`);
- (in case of `after returning`) the binding of the return value to a context variable;

[23] Their names always start with a capital letter.

- the name of the variable which contains the join point that is matched by the pointcut;
- a pointcut (behind the colon);
- the advice body.

The advice body contains pure C code enhanced with template parameters, whereas the advice signature is a new construct and the pointcut (see Section 5.3.4) consists of Prolog predicates with C-like conjunction, disjunction and negation operators. This makes our aspects a hybrid of pure C and a Prolog-based pointcut language.

The advice of Figure 5.13 is `around`-advice on join points `Jp` fulfilling the pointcut on lines 2–4. We explain this one later on, but in any case two context variables are bound (`ReturnType` and `Src`) and are available for use as template parameters in the advice body. One can hide variables bound during join point matching by not adding them to the binding list. Bindings are typed. `Src` is a simple string, whereas `ReturnType` represents an actual C `TYPE`. This is a custom (meta)type we have added to Aspicere, because C does not have reflective capabilities. One can use such a type parameter further on in the binding list, as return type of the advice (line 1) and inside the advice body itself (line 5). This enables the weaver to statically check the type correctness of advice instead of deferring this to run-time (with the catch-all `void*`).

Aside from template parameters, an advice body may contain a `proceed`-call, similar to AspectJ. If no arguments are given, the join point's original arguments are passed as is to any remaining advices on the same join point or to the join point itself. If one wants to replace the value of the arguments, one should fill in the arguments of `proceed`:

```
1 ...
2 else dst=proceed(``test'');
3 ...
```

Alternatively, assigning the new value directly to the variable binding works too:

```
1 ...
2 else{
3   Src=``test'';
4   dst=proceed();
5 }
6 ...
```

A third alternative is to access the thisJoinPoint-like struct which is accessible via the join point variable (`Jp`):

```
1 ...
2 else{
3   Jp->args[0]=``test'';
4   dst=proceed();
5 }
6 ...
```

This `struct` contains the following fields with information about the join point:

nrArgs number of arguments

args array of arguments

returnValue pointer to the return value

fileName name of the file in which the advised join point resides

functionName name of the advised function

Contrary to Aspicere1, the Aspicere2 weaver does not support these fields, primarily because this `struct` can be costly to create, store and access. Instead, binding these values as context variables in the pointcut is preferred.

There is a fourth and final way to access and modify only the return value of an advised procedure. When using `after returning`-advice, the return value is automatically bound to a context variable:

```
1 void safe_ato(TYPE ReturnType,char* Src) after Jp
2  returning (ReturnType Res):
3   ...{
4     ...
5     *Res=0;
6     ...
7 }
```

Note that, just like in the context `struct` above, `ReturnType` represents the corresponding pointer type of the actual return type, such that assignment to the bound return value is possible via dereferencing of this pointer (line 4).

5.3.3.2 Join Point Property Declaration

Apart from advice, an aspect may contain a join point property declaration. Figure 5.14 gives an example of this. It attaches an integer identification number to the same `call` join points which are advised by Figure 5.13. The property can be accessed by any advice on these join points if a context variable is bound to the property. The advice on lines 4–8 of Figure 5.14 binds a pointer to the `index` property to the `Index` context variable on line 6. This context parameter is used on line 7 to store the value of a global variable.

```
1 int index on Jp:
2   invocation(Jp,"ato.");
3
4 void count(int* Index) before Jp:
5   invocation(Jp,"ato.")
6   && property(Jp,index,Index){
7     *Index=global_counter++;
8 }
```

Figure 5.14: Join point property associated with the advised join points of Figure 5.13.

This concludes the discussion of Aspicere's advice and join point declaration structures. The next section presents the logic pointcut language.

5.3.4 Aspicere's Pointcut Language

As mentioned in Section 5.3.1, Aspicere's pointcut language is heavily influenced by LMP [237, 30, 202]. The basic idea is that a program is represented as a collection of logic facts and that a Turing-complete logic language is used to reason about these facts. There are two flavours of LMP: a generative and a query-based approach. De Volder's TyRuBa [237] uses template-based meta-programming to generate the woven system based on facts and logic rules which model program fragments and aspects respectively. More in particular, the weaving happens by means of the logic rules which generate new facts and modify existing ones. Gybels' CARMA [108] (then: Andrew) on the other hand, uses a hybrid Smalltalk/Prolog language (SOUL) as a pointcut language to query the base code for join points. Weaving is then performed imperatively using Smalltalk's meta-programming facilities instead of via the logic rules. Similar to Cobble, Aspicere applies the query-based LMP approach.

Basically, the base code is reified as a database of logic facts (Prolog in our case). Pointcuts can be expressed in terms of those facts and hence are able to compose powerful patterns based on program structure using conjunction, disjunction and or negation (by unprovability). Ultimately, any kind of fact source could be used [185], such as data from execution control flow, meta-information from annotations, etc. As argued in [202], legacy languages like C lack sufficient structure or reflective capabilities to be able to write crisp and robust pointcut patterns and advice. LMP is a good solution for these mismatches. Adding new pointcuts boils down to defining new logic predicates. A clean, layered pointcut language can be constructed by carefully specifying more advanced predicates in terms of more primitive ones. By raising the level of abstraction, pointcut predicates can be brought closer to the actual problem domain.

We can now understand the pointcut of Figure 5.13. This advice matches all

`calls`[24] (line 2) to procedures of which the name matches the regular expression "ato.", i.e. the name starts with `ato` and the fourth letter is arbitrary. Because Aspicere does not allow advising a call via a function pointer (see Section 5.3.2), this advice only matches explicit function calls. To connect the `invocation` predicate to the ones on lines 3 and 4, Prolog's unification allows us to reuse the previously bound join point variable (`Jp`). This is actually a very natural way to express that two bound variables should be equal[25]. The `args` predicate binds the sole argument of the advised calls[26] to the `Src` variable and also captures the procedure call's return type as `ReturnType`. Of the bound variables, the interesting ones can be exported to the advice via the binding list on line 1.

The unification has an interesting side-effect. Consider the following pointcut:

```
1 invocation(Jp, _) && args(Jp,Params) && member(P, Params)
```

This corresponds to the following Prolog rule:

```
1 /*rule name*/:-
2   invocation(Jp, _),
3   args(Jp,Params),
4   member(P, Params).
```

The unification semantics of Prolog dictate that if a variable can have multiple values, each one is eventually used to find a complete solution for the logic rule. In this case, `Params` is a list of function arguments. The `member` predicate on line 4 binds every argument in the list in turn to `P`, which can be used as a template parameter in the advice body. Because there are potentially many function arguments passed to the `Jp` call join point, the rule, and hence the pointcut, matches multiple times. This particular advice matches as many times as there are function arguments.

Aspicere's logic facts contain base program structure information or may represent weave-time meta data. The recorded program structure consists of the existing procedure and variable definitions and declarations of the base code, together with arguments, file information, etc. The meta data can contain any user-specified information. It can record information obtained via reverse-engineering and make it accessible in other advice which is used to re-engineer the system, or convey design information, information about the actual composition of source modules (is one executable built or multiple libraries?), information of base code modules which have been selected by the user, etc. As argued for the L2 goal, weave-time meta data is very important for AOP in legacy systems.

Logic facts are useful to store meta data separately (loose coupling) from the base code, but sometimes meta data should be attached to specific program entities.

[24] Because of name clash issues in our weaver implementations, we use `invocation` instead of `call`.
[25] Prolog also allows one to express that certain variables can have an arbitrary value, by writing an underscore.
[26] `Src` stands between `[...]` because procedures can have in general a list of arguments.

Because of this, we have added an annotation facility to C. Annotations are well-known by now as AspectJ 1.5[27] has had them for some time. Put briefly, they correspond to meta data which is associated directly to program elements by means of a kind of tags. They convey information, especially semantical, which is not expressible in the programming language itself. As C does not have annotations by default, we abuse comments for this, similar to the style in Java before annotations were turned into first-class entities. Our annotations are associated with the first statement or (nested) expression which follows them:

```
1 int main(void){
2   /*@repeat(``Jack'',66)*/
3   printf(``This line will be repeated.\n'');
4 }
5
6 int repeat_printf(char* Name,int Iter) around Jp:
7   invocation(Jp,"printf")
8   && annotation(Jp,repeat,[Name,Iter]){
9     int i=0,res=0;
10
11    printf(``Hi %s,\n'',Name);
12    for(;i<Iter;i++) res+=proceed();
13
14    return res;/*do not care*/
15 }
```

Based on meta data attached to a call to `printf`, this advice greets the user and repeats the call a number of times based on the meta data. The annotation on line 2 has a name (`repeat`) and may contain an arbitrary number of attributes. Based on a join point and the annotation name, the list of attributes can easily be retrieved in a pointcut (line 8) for use in the advice body. In Chapter 8, a more interesting application of annotations is presented.

Aspicere supports the dynamic `cflow` pointcut of AspectJ. To provide more advanced dynamic pointcuts, Aspicere's pointcut language has been extended into a history-based language, i.e. pointcuts can be expressed in terms of temporal operators to relate two or more distinct join points in time. This means that more precise and advanced pointcuts can be developed. Aspicere's temporal extension is named cHALO and is discussed in the fourth case study (Chapter 9). The next subsection combines the presented elements of Aspicere's join point, advice and pointcut model into a larger example.

5.3.5 Aspicere at Work: Database Error Recovery

In this section, we give a more complete example to illustrate various facets of Aspicere, and some of its shortcomings. Consider an application which accesses

[27] `http://www.eclipse.org/aspectj/`

```
1 int recover(int ErrNr, int Retry, char* ErrStr) around Jp:
2   critical_call(Jp)
3   && sql_redo(ErrNr,Retry)
4   && sql_code(ErrNr,ErrStr){
5     int res=0, i=0;
6     for(i=0;i<Retry;i++){
7       res=proceed();
8       if(res!=ErrNr) return res;
9     }
10
11    printf(``Recovery did not succeed: %s\n'',ErrStr);
12    return res;
13 }
```

Figure 5.15: Database error recovery advice.

```
1 sql_code(-666,'Time limit exceeded.').
2 %> Some other codes with their description...
3
4 sql_redo(-666,2).
5 %> Some other codes with the relevant number of
6 %> retry attempts...
7
8 critical_method(``monthlyPayment'').
9 critical_call(Jp) :-
10   invocation(Jp,``_iqcftch''),
11   critical_method(Name).
12   execution(CMethod, Name),
13   cflow(CMethod,Jp).
```

Figure 5.16: Prolog predicates associated with the database error recovery advice of Figure 5.15.

a database. We would like to catch database errors in such a way that failed SQL-queries are retried a number of times, depending on the specific error code. This recovery procedure should only be active if the error occurs during execution of an important method. As the concept of recovery is the same across all error codes, we can encode this knowledge as meta data and use this information directly in a pointcut.

The resulting advice is shown in Figure 5.15 and its accompanying Prolog rules are shown in Figure 5.16. The latter file specifies a number of meta data facts which declare the error explanations (lines 1–2) and the number of retrial attempts (lines 4–6) for various error codes. These facts are bound in the pointcut on lines 2–4 of Figure 5.15 to variables `ErrNr`, `Retry` and `ErrStr` respectively. Within the advice body, a loop retries the failed function invocation the prescribed number of times for the specific error corresponding to `ErrNr`. If all attempts fail, the advice exits with the correct error message. Otherwise, i.e. no error has occurred (line 8), the advice exits normally.

The particular join point on which the recovery advice applies is described on line 2 as a `critical_call`. The definition of this predicate is listed on lines 9–13 of Figure 5.16. A `critical_call` is a call to the `_iqcftch` procedure (part of the SQL library) which occurs in the `cflow` of the `execution` of a procedure `CMethod`. `CMethod` has to be registered explicitly as a `critical_method`. There is only one such method, i.e. `monthlyPayment`. In other words, only calls to `_iqcftch` in the `cflow` of the `monthlyPayment` method are advised. This pointcut makes explicit use of weaving meta data. Alternatively, one could annotate the critical methods instead of listing them as facts.

Because of unification during join point matching, each advised call to `_iqcftch` is matched as many times as there are registered error codes. In the ideal case (no error), each advice only executes up to line 8. If there are on average a retrial attempts per error code and there are n different error codes, then the database query will be run a^n times in the worst case: for every iteration of the outermost recovery advice instance, the next instance will at most iterate a times, etc. This corresponds to the notion of "handler chains" mentioned by Lohmann et al. [160], i.e. multiple advices are chained together on one and the same join point, each checking for one particular thing. As advice usually is transformed by a static weaver into a procedure, inlining of the advice (like Aspicere2 does) is a necessity to avoid the potential overhead of a^n procedure invocations.

A more efficient solution could be based on De Schutter's generative aspect approach [202]. Instead of using the meta data to determine how many instances of the advice match, the meta data could be used to construct the advice itself:

```
1 int recover() around Jp:
2   critical_call(Jp){
3     [for each match of sql_redo(ErrNr,Retry)]
```

```
4       int retry_ErrNr=Retry;
5       [end]
6
7       while(1){
8         res=proceed();
9         [for each match of sql_code(ErrNr,ErrStr)]
10        if(res==ErrNr){
11          if(retry_ErrNr--) continue;
12          else{
13            printf(``Recovery did not succeed: %s\n'',
14                   ErrStr);
15            return res;
16          }
17        }else
18        [end]
19        return res;
20      }
21  }
```

This kind of advice template is currently not supported by Aspicere, but Mirjam [179] (discussed in the next section) does. It is similar to e.g. JavaServer Pages[28], except that the variables accessed in the templates (lines 3 and 8) are context variables bound by a pointcut. This style has some similarities with C++ templates or "class morphing" [122].

5.3.6 Comparison with an Industrial Aspect Language for C: Mirjam

To put the design of Aspicere in perspective, we compare it in detail to the design of a new aspect language for C which has been proposed[29] during the writing of this dissertation [179], independently from our work. It originates from the "Ideals" project. This project was concerned amongst others with dealing with crosscutting concerns in the industrial C code base of ASML (more on this in Chapter 8). Many researchers have explored various facets of the re-engineering of the system with aspects [36, 33, 34, 35, 75, 32]. This has led to a number of desired aspect language features, on which Mirjam has been based [179]. Mirjam is meant to be an industrial-strength, general purpose aspect language for C suited for large-scale software development. WeaveC (Section 5.2.1.6) is the actual weaver back end of Mirjam. As the Ideals report is the first written description of Mirjam, which means that Mirjam is state-of-the-art, we consider Mirjam to be the ideal reference point to evaluate the design of Aspicere. The design of Mirjam confirms the requirements we have proposed in Section 5.1.

[28] http://java.sun.com/products/jsp/

[29] Only a language description has been published, but no weaver implementation has been released yet.

```
1  context{
2    #include <stdio.h>
3  }
4
5  aspect database{
6    advice printParams(Variable@JP[] Args)
7                                    before (FunctionJP JP){
8      printf("Function %s in module %s executes with arg(s) "
9             @StringConstant<
10             [arg in Args:
11              strConcat(arg.name, "=",formatString(arg))],
12             "; " //semicolon as separator
13            >@
14            ,JP.name
15            ,JP.module.name
16            ,@VarargExpression<[arg in Args:
17                                formatExpression(arg)]>@);
18   }
19
20   query Q5() provides (FunctionJP JP, Variable@JP[] V){
21     JP: |JP.formalParameters()| >= 1;
22     V : V == JP.formalParameters();
23   }
24
25   binding{
26     foreach (thisFunc, vars) in Q5{
27       apply on thisFunc {
28         printParams(vars);
29       }
30     }
31   }
32 }
```

Figure 5.17: Mirjam example adapted from Nagy et al. [179] (their Listing 5.9).

Figure 5.17 shows an example aspect written in Mirjam. Aspects do not align with file boundaries. Instead, there is an explicit `aspect` construct (line 5) and the usual C declarations and inclusions have to be specified within a context declaration closure (lines 1–3), which is preprocessed before weaving starts. Within the `aspect` declaration, `advice`, `query` and `binding` constructs may occur, which can all be referenced from the outside as the `aspect` defines a name scope. This increases reusability, at the expense of more additions to the C base language.

The `advice` on lines 6–18 prints out the argument names and values before an advised `execution` join point is resumed. Mirjam provides `before` and `after` advice on `call` and `execution` join points. It is not clear how function

pointers are handled. The advice's signature (lines 6–7) specifies that the advised join point `JP` is an execution join point (`FunctionJP`) which may expose a list of formals or global/local variables bound to context variable `Args` in scope at the join point (`Variable@JP[]`). In addition to bound context, the advice body can also access other context like the name of the join point (line 14) or the enclosing module's name (line 15). Note that the "."-notation enables easy navigation through the structure of the context and also enables access to methods. Aspicere on the other hand only provides one-level deep access to join point variable context, unless the right context is bound directly to a variable in the pointcut.

The two expressions between `@...<...>` (lines 9–13 and lines 16–17) are advice generators, which are similar to De Schutter's generative aspect approach [202]. Because Mirjam does not provide `around`-advice, we could not rephrase the example from Section 5.3.5. Instead, the first generator in the advice of Figure 5.17 (lines 9–13) generates a string (`StringConstant`) which consists of the concatenation (with `;` as separator) of statements of the form "argument name = argument value", one for each captured argument. The latter is achieved via the iterator on lines 10–11 (between `[...]`). The `strConcat` and `formatString` methods are executed at generation-time. Lines 16–17 contain a similar generator, but this one returns a comma-separated list of all argument values. Each list element returned by method `formatExpression` matches the corresponding format string generated by `formatString` on line 11. Advice generators are more powerful than Aspicere's template parameters, because they generate code expressed in terms of names of variables instead of in terms of the build-time value of bound context. This is a consequence of the generative approach compared to a traditional aspect language.

Mirjam pointcuts are queries which return for each match a tuple consisting of join point locations and context, analogous to Aspicere. Contrary to Aspicere, context variables of pointcuts are explicitly typed. The query on lines 20–23 returns a tuple of two elements: an execution join point `JP` and a tuple `V` of variables which are accessible at `JP` (with the type `Variable@JP`). The query is actually a relational query which returns all tuples (`JP`,`V`) for which `JP` has at least one formal parameter (line 21) and `V` is the complete list of formal parameters (line 22). An equivalent predicate in Aspicere would look like:

```
1 Q5(JP,V):-
2   execution(JP,_),
3   args(Jp,V),
4   nth0(0,V,_).%>=1 elements
```

Finally, the `binding` closure on lines 25–31 contains binding definitions like the one on lines 26–30. This explicitly allows to bind a particular join point to concrete advice, by capturing the results of a query (line 26) and applying a particular advice (line 28) to each captured join point (line 27). The `apply`-construct on

lines 27–29 is a "binding entity". There can be multiple binding entities within the same `binding` definition. Other kinds of binding entities can control the order of advices at shared join points, or generate warnings/errors. State can be declared within a `binding` closure for use by all `binding` definitions (and hence advice), or within a binding entity to restrict access to the state to particular join points only. The former corresponds in Aspicere to global variables within aspects, while the latter is related to join point properties.

Mirjam provides annotations to specify weave-time meta data. Nagy et al. [179] acknowledge the important role of build-time meta data (goal L2):

> In our investigation, we recognized that information derived from the build process is also used in the idioms that realize crosscutting concerns in software. [...] Hence, information about product and platform can serve as variation points; these are typically originated from the configuration of the build processes.

Mirjam provides information from the build process to advice via the join point variable and the "."-notation. WeaveC/Mirjam has been integrated into the build system of ASML [179], but no more details have been given about this process. Two concerns (tracing and timing) have been implemented as aspects thusfar on more than one thousand base modules.

From this detailed discussion, we can conclude that Mirjam in various ways is more powerful than Aspicere, especially regarding the generative advice, but at the expense of a more complex aspect language (pointcuts and advice). In any case, the fact that Mirjam satisfies almost all our requirements is an important indication about the validity of the L1 and L2 goals. Real validation of these requirements, both for L1 and L2, is done in the five chapters following this chapter. These are introduced in Section 5.5, but first the next section presents the two weaver implementations of Aspicere.

5.4 Two Weaver Implementations for Aspicere

Two aspect weavers have been built for Aspicere. The first one, which we name "Aspicere1"[30], is a source-to-source weaver [248] like AspectC and AspectC++. The second one, "Aspicere2"[31], is a link-time weaver [8]. This section discusses their main characteristics without going into too much detail. More information about their internals can be found as documentation distributed with the source code of these weavers.

[30] `http://users.ugent.be/~badams/aspicere/`
[31] `http://users.ugent.be/~badams/aspicere2/`

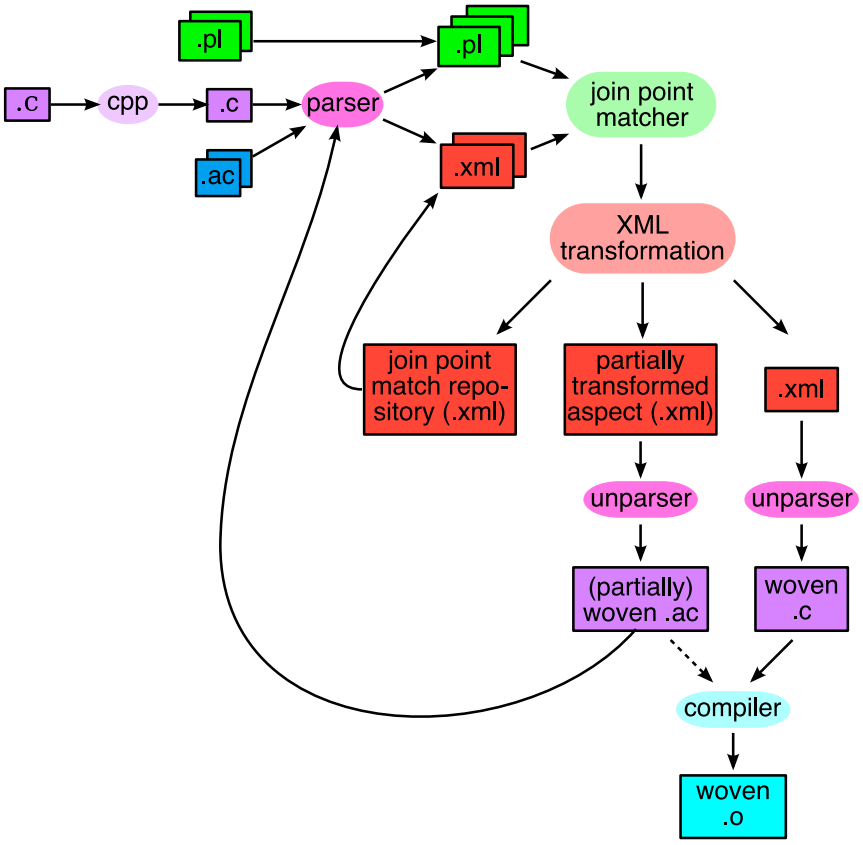


Figure 5.18: Architecture of Aspicere1. The .ac-files represent aspects, while .pl-files contain logic predicates.

5.4.1 Aspicere1, a Source-to-source Weaver

Figure 5.18 shows the architecture of Aspicere1. It is a pure source-to-source weaver, which takes as input base code (one module at a time), aspects (.ac) and Prolog modules (.pl), and generates woven C code which can be compiled using the normal base compiler. This means that the weaver has to be executed before every compiler invocation, as we will see later on. Aspicere1 is actually built on the same basic framework as Cobble [142], except for the parser. This one is based on an existing robust ANTLR-parser[32] which is capable to parse K&R-, ANSI- and GNU-style C.

Inside the weaver, a parser and unparser convert the C code to/from an XML representation of the AST (analogous to AspectX in Section 5.2.1.4). The Pro-

[32] http://www.antlr.org/

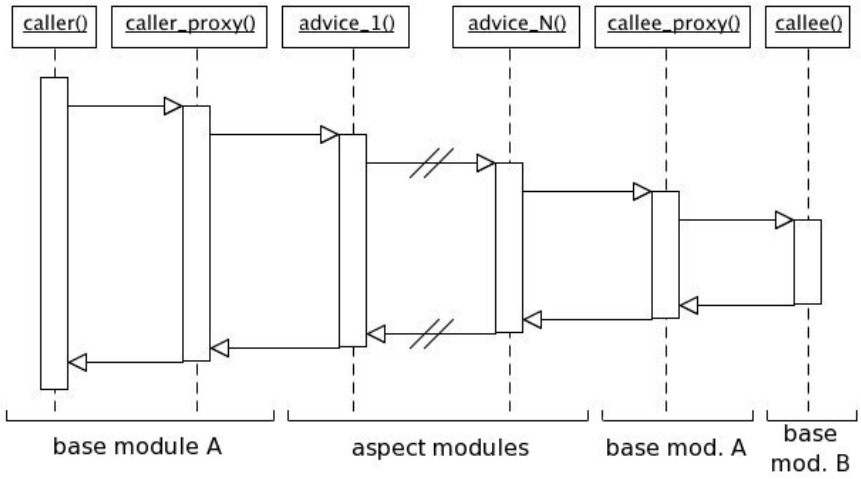


Figure 5.19: Sequence diagram (with methods instead of objects) showing the control flow through an (`around`-)advice chain in Aspicere. The respective owners of the various procedures are also shown.

log modules and the pointcuts (transformed into Prolog rules) are used to locate the right join point shadows [118], i.e. the appropriate XML nodes of the AST which correspond to function calls or definitions. Once these have been found, the XML tree representing the base code module can be transformed. The aspects themselves are also transformed (more on this later). The woven base code module is converted back to C code, whereas the transformed aspect possibly needs to be transformed further when weaving other base code modules. Eventually (dashed arrow), the resulting transformed aspect is compiled and linked with the transformed base code.

Aspicere1 uses a Java-based Prolog engine, i.e. TuProlog [59]. One of its features is the interaction between logic predicates and Java libraries, which enables us to make use of Java's extensive XML facilities. Instead of reifying the whole XML tree as logic facts, we use a layered system of predicates of which the lowest ("hyperprimitive") layer looks up XML nodes on demand using XPath queries. Unfortunately, even with the use of a cache, this turns out to be very slow (see Chapter 6). The middle ("primitive") layer defines primitive pointcuts like `call` and program structure navigation queries. This layer immediately accesses the hyperprimitive layer. Finally, the upper ("aspect-specific") layer is meant for predicates written by users, tailored to a particular domain [30]. TuProlog also makes it possible to store meta data inside XML files or a relational database and access this from a predicate as if this data is stored as a normal Prolog fact.

The XML transformation is based on a couple of principles. Advice is converted into multiple C functions, one per combination of type parameters used in the advice[33] body. These generated functions are named "advice instances", and they correspond to the XML nodes which are added to a (partially) transformed aspect (the loop in Figure 5.18). However, care is needed to manage access to static and global variables.

We have decided to reduce code bloat by decoupling advice instances from particular sequences of advice on a shared join point. Instead of hardcoding a call to the next advice or the advised join point inside each advice instance, we make the instances eligible for reuse by encoding the advice order inside a list of function pointers, which is stored in the join point `struct` pointed at by the advice's join point variable. This `struct` forms the sole argument of an advice instance.

The resulting chaining of advices is shown on Figure 5.19. This sequence diagram illustrates both the order of woven advice instances and the order of advice execution on a shared join point. The list of function pointers is initialised inside a generated `caller_proxy` method called at the join point shadow. This also marshalls the join point's context into the context struct argument of the advice instances. Calls to `proceed` inside the advice are converted into generic function pointer arithmetic which selects the next advice instance in the chain until a `callee_proxy` is invoked. This one demarshalls and restores the (possibly modified) arguments of the join point. Because the `caller_proxy` and `callee_proxy` are both added by the weaver to the base module, even `static` methods can be advised and `static` variables accessed. There are some special cases, like calls to `exit`, which may lead to memory leaks when the memory allocated by caller proxies for a context `struct` is not deallocated.

Aspicere1 does not support annotations, join point properties or dynamic pointcuts like `cflow`.

5.4.2 Aspicere2, a Link-time Weaver

C is all about writing efficient code, but AOP has been known to add overhead to build and (most importantly) run-time execution [14, 73] compared to a tangled and scattered implementation. The people from the AspectBench Compiler for AspectJ (abc) [15] have acknowledged this by creating a common workbench to experiment with new features, optimisations and analyses for AOP in Java. It provides an alternative implementation of an AspectJ weaver with an open architecture, enabling easy extensions at every level (join points, pointcuts, advice, weaving, etc.) and speeding up woven code via sophisticated, whole-program static analysis and optimisation [14, 15]. We have considered a similar workbench for C to be favourable for the following reasons:

[33] Aspicere1 does not implement `before`- and `after`-advice, only `around`-advice.

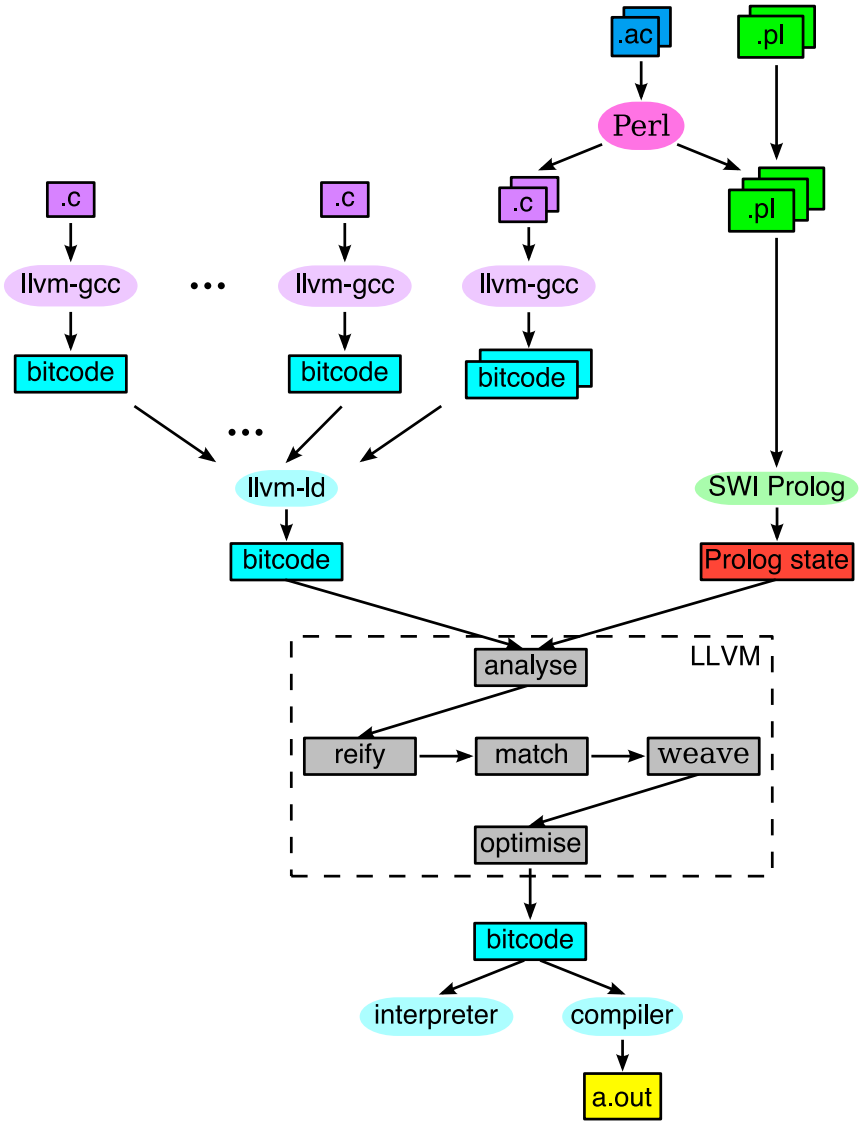


Figure 5.20: Aspicere2 architecture.

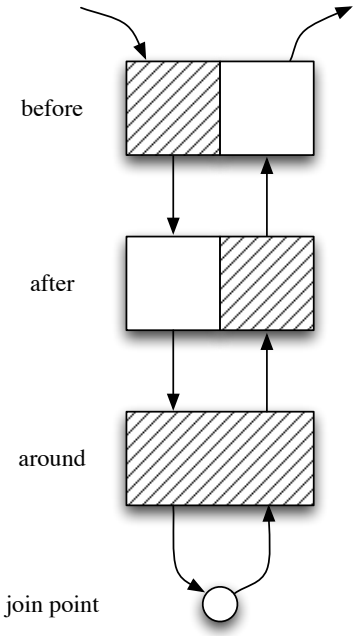


Figure 5.21: Precedence of advice on shared join points with Aspicere2.

- AOP-researchers in C systems do not need to re-implement basic weaving functionality from scratch;
- existing optimisations can be applied to C to check their validity and to examine whether they can be specialised;
- new analysis and optimisation techniques for typical C features like macros and (function) pointers can be developed and applied directly.

Figure 5.20 shows the resulting architecture of the Aspicere2 weaver. Instead of a source-to-source weaver, weaving happens at link-time to facilitate whole-program analysis. This architecture is inspired by abc [15], which consists of a front- (Polyglot[34]) and back end (Soot[35]). Polyglot has provisions to treat the aspect language as an extension of Java with adequate type checking, while Soot offers analyses and transformations at link-time to optimise the weaving process based on whole-program knowledge. abc also has a powerful re-weave facility, which allows to do an initial weaving pass, to analyse the weaving results, to undo the weaving and then to re-weave based on the analysis results of the first weaving attempt.

[34] `http://www.cs.cornell.edu/Projects/polyglot/`
[35] `http://www.sable.mcgill.ca/soot/`

For Aspicere2, the role of Soot and Polyglot is played by a framework targeted specifically at C code. We have opted for LLVM (Low-Level Virtual Machine)[36], i.e. the framework TOSKANA-VM (Section 5.2.3) is built on[37]. Instead of using LLVM in a VM setup, we apply it in a traditional compile-link setup, compatible with classic development traditions. One only needs to replace the normal C compiler by `llvm-gcc` and the standard linker by the combination of `llvm-ld` and LLVM. These tools generate or operate on LLVM bitcode instead of on machine code. LLVM bitcode is a compact SSA-form intermediate representation (IR) similar to Soot's Jimple IR. LLVM offers tools, analyses and optimisations to build powerful compilers or bitcode transformers. Memory footprint and execution time are significantly lower than GCC allows. By invoking LLVM at link-time, we obtain a whole-program view of the base code. This enables us e.g. to reduce the number of join point shadows where dynamic residues for a `cflow`-pointcut should be put by using points-to analysis and control flow graphs, or to perform aggressive dead code elimination or inlining of the woven bitcode.

We now consider the various components shown on Figure 5.20. A GCC-based front end compiles C code to LLVM bitcode. The C code and the advice bodies of aspects (.ac files) are converted into normal C files and then transformed into bitcode, while the pointcuts are turned into Prolog rules (.pl files). Together with user-provided Prolog modules and facts extracted from annotations in the source code (which has been elided from Figure 5.20), the predicates are compiled by the SWI Prolog[38] compiler into a binary state file[39]. Parsing and transformation is done via a Perl script, because LLVM's companion High-Level Virtual Machine framework (HLVM), comparable to abc's Polyglot, is still in its infancy. HLVM is a framework on top of LLVM which will accommodate interpretation of dynamic languages like Ruby, but (more interesting for Aspicere) will allow easy addition of new language-level features to a language via generation of extra lexer and parser functionality.

When all source files have been converted into bitcode, the resulting bitcode files are glued together at link-time into one link module and the Prolog state is loaded into an embedded Prolog engine. First, some analyses can be run (if required by later passes) before a reification pass traverses the link-time module and asserts all interesting program entities as Prolog facts (inspired by the design of the LogicAJ aspect language for Java[40]). Libraries for which no bitcode is provided cannot be woven into. During the matching phase, all pointcut logic rules are queried to find join point shadows which may give rise to a pointcut match.

[36] `http://llvm.org/`
[37] CIL (`http://cil.sourceforge.net/`) is an alternative framework we have considered.
[38] `http://www.swi-prolog.org/`
[39] Such a file contains binary representations of predicates and facts for fast loading by the SWI engine.
[40] `http://roots.iai.uni-bonn.de/research/logicaj/`

As a side note, Aspicere2 still uses a vanilla Prolog engine as back end for Aspicere's pointcut language, although Hajiyev et al. [109] claim that Prolog is not scalable enough for this. To be able to verify this claim once a robust Datalog implementation becomes available, the Prolog manipulation components of Aspicere2 have been encapsulated in dedicated classes such that they can easily be replaced.

Armed with the set of pointcut matches, a weaving pass transforms the IR representation of the whole program. The transformation for `around`-advice is similar to Aspicere1's advice chaining strategy (Figure 5.19), except that advice instances are not reused. They explicitly call the next instance in the chain or the advised join point. This causes extra code bloat compared to Aspicere1, but it removes the need for caller and callee proxies, does not require run-time checks and (most importantly) opens the door for aggressive static optimisations like inlining, dead argument elimination, etc. Aspicere1's generic function pointer arithmetic prohibits this kind of optimisations. Contrary to Aspicere1, `before`- and `after`-advice are implemented. They are transformed into calls before or after the join point shadow. Join point properties are implemented as global or local variables, based on the join point (shadow) they are attached to. To implement `cflow`, we first construct a control flow graph via a standard LLVM analysis. We use this control flow graph to deduce which join shadows always yield a match and which ones possibly do, similar to the approach of Avgustinov et al. [14].

The order of advice on shared join points is determined as shown on Figure 5.21, which is easy to remember. One just needs to imagine the lexical order of the definitions of all advices which advise a given join point, and stack the advices like rectangles[41]. Then, depending on the kind of advice (`before`, `after (returning)` or `around`), either the left part, right part or the whole advice rectangle can be marked. If control flow is depicted as shown on Figure 5.21, then the crossing of an arrow and a marked rectangle represents execution of advice. Hence, in the example of Figure 5.21, a sequence of `before`-, `after`- and `around`-advice results in execution of the `before`-advice, followed by the `around`-advice and eventually, if the `around`-advice exits, the `after`-advice. This precedence model is intuitive, whereas it is still flexible. Aspicere1 implements a similar precedence model.

Once the bitcode has been transformed by the transformation passes, we first optimise the woven code by inlining, dead code elimination, etc. The end result is a bitcode representation of the whole, woven program, which can either be interpreted, compiled directly into machine code or translated back to (very low-level) C code.

Because of LLVM's flexibility, Aspicere2 supports much more features of Aspicere than Aspicere1 does, e.g. those of Chapter 8 and Chapter 9. Calls via

[41] The order of aspects given to the weaver determines the order of advices from distinct aspects.

function pointer can now be detected more accurately, but we have not looked at this in more detail yet. Similarly, support for incremental weaving has not been investigated either. Contrary to a source-to-source weaver, a change to a C file not only requires re-compilation and re-weaving of that single file, and re-linking of the complete system. The whole linked system has to be re-woven, i.e. analysed, reified, etc. To resolve this, Aspicere2 should be able to trace back from woven program artifacts to the original base code and aspects. This is considered in the future work section of Chapter 11.

5.5 Validation of Goals L1 and L2

The next five chapters present five case studies performed with Aspicere and analysed with MAKAO. Their goal is to prove the existence of problems caused by co-evolution of source code and the build system in legacy systems in which AOP technology is introduced, to correlate these problems with the four roots of co-evolution and to validate MAKAO and Aspicere. The latter validation focuses both on the degree to which goals L1 and L2 have been met.

Each case study first presents the application of aspects in the source code to solve a given problem. Two of the cases involve reverse-engineering activities, while the last three target re-engineering of a legacy system (extraction of exception handling into aspects, system composition using an architectural description language and extraction of conditional compilation into aspects). After the discussion of L1, the impact of the introduction of aspects on the build system is considered, i.e. build problems which have occurred and an explanation whether or not we have been able to resolve them. Afterwards, we validate the applicability of the four roots of co-evolution as explanation of the build problems, the use of MAKAO to understand and manage the co-evolution, and the degree to which Aspicere fulfils goal L2. A summary of the validation results of all case studies is given in Chapter 11.

5.6 Conclusion

This chapter has derived two categories of requirements, i.e. for aspect languages for legacy systems in general (L1) and for language support to deal with co-evolution of source code and the build system (L2). We have evaluated existing aspect languages for C, and have found that none of them fulfils all requirements. Instead, we have designed and implemented Aspicere, i.e. an aspect language for C with an expressive logic pointcut language, provisions for generic advice and access to any kind of weave-time meta data. The latter is the key for integrating the build system with the aspect language. We have evaluated the L1 and L2 re-

quirements w.r.t. a recently proposed industrial aspect language for C. Finally, we have discussed the two weaver implementations of Aspicere.

The next chapter presents the first case study, which reverse-engineers a legacy system using aspects.

Substituting gce for gcc in a makefile can range from very easy to extremely difficult. [...] In systems that use many makefiles, or that are specifically dependent on the value of the $CC variable, the substitution task may be considerably more difficult, and additional modifications of the makefile may be required.

Instructions of the SWAG toolkit [222]

6

Case Study 1: Reverse-engineering of the Kava System using Aspects

THIS chapter[1] discusses a first case study in which AOSD, i.e. Aspicere, is introduced in a legacy system. It serves as a validation of the suggestion made by two of the four roots of co-evolution (RC1 and RC2) about the occurrence of co-evolution problems between source code and the build system in AOSD systems. At the same time, MAKAO (goal T2) and Aspicere (goal L1) are validated as well.

The case study uses dynamic analysis [248] to extract knowledge about the internal composition of the Kava system (Section 3.5), a C code base of 453 kSLOC. A tracing aspect has been applied to facilitate a web mining technique [246], for measuring the degree of coupling between compilation units, and for the visualisation [247] of the results of "Frequency Spectrum Analysis", aimed at assessing cohesion within modules. This chapter does not go into detail about these analyses or the results. Instead, we focus on the role of aspects and the build system.

The next section describes the context of the case study, followed by a discussion of the aspects we have applied (Section 6.2). Then (Section 6.3), we report on the build system problems we have encountered to facilitate the aspects in the source code. Section 6.4 validates the prediction of two of the four roots of co-evolution about co-evolution of source code and the build system in legacy systems in which AOSD has been introduced. Section 6.5 evaluates the ability of MAKAO

[1] Parts of this chapter are based on [248].

to deal with this. Finally, Section 6.6 presents the conclusions of this case study.

6.1 Rationale behind the Case Study

Contrary to static analysis, dynamic analysis does not consider a system's source code, but rather a specific program trace obtained by making the system execute a well-defined, well-chosen scenario. This trace can be analysed off-line. The developers at Kava have pointed us to the so-called *TDFS*[2] batch application. They use TDFS as a check to see whether adaptations in the system have any unforeseen consequences. As such, TDFS should be considered as a functional application — it outputs a detailed invoice of all prescriptions, ready to be sent to the health care insurance institutions—, but also as a form of regression testing. As an added bonus, the program trace could be reused to calculate test coverage of the Kava system.

There are many ways to obtain a program trace. The easiest one is to run a program like "strace", which writes out all system calls invoked by an application. However, system calls are in general too low-level and do not contain any correspondence with the actual source code. At the other extreme, adding `fprintf` statements throughout the code enables the human tracer to print out any desired info to a file, at the expense of huge manual effort and an unevolvable tracing implementation. In practice, dedicated tools like DTRACE [38] or ATOM [217] are the preferred way of gathering sufficiently detailed tracing information in a custom format. Still, new mechanisms for tracing could be interesting to further lower the barrier of obtaining a dynamic program trace. AOP is one of those mechanisms. From an AOP point of view, a program trace can be obtained via a basic tracing aspect. Such an aspect is the prototypical example of a homogeneous [51], extremely scattered aspect which matches join points throughout the whole system to record information about the program. Unless it has to replace an existing, inconsistently used base code implementation of tracing [32], it is very straightforward to implement and it can be easily customised to record the tracing information the user needs in the right format. Additionally, if the aspect is written well [54], the original semantics of the application are left untouched, which makes the woven system easier to understand. As such, a tracing aspect resembles the typical "Hello World" program people use as a first example of a new programming language.

We have seized the opportunity given by the Kava case study to:

- assess the feasibility of using AOP for future re(verse)-engineering tasks;
- stress-test Aspicere's template parameters and pointcut language;
- integrate Aspicere1 with Kava's build system.

[2] "*T*ariferings*D*ienst *F*actuur en *S*tatistiek", or "Tarification Service for Invoices and Statistics".

```
1 ReturnType trace(TYPE ReturnType,char* FileStr) around Jp:
2   invocation(Jp, ``^(?!.*printf$|.*scanf$).*$'')
3   && type(Jp,ReturnType)
4   && !!is_void(ReturnType)
5   && trace_file(FileStr) {
6     FILE* fp=fopen(FileStr,"a");
7     ReturnType i;
8
9     fprintf (fp,"before ( %s in %s ) \n",
10              Jp->functionName,Jp->fileName);
11    fflush(fp); /* call sequence */
12
13    i = proceed (); /* continue normal control flow */
14
15    fprintf (fp,"after ( %s in %s ) \n",
16              Jp->functionName,Jp->fileName);
17    fclose(fp); /* return sequence */
18
19    return i;
20 }
```

Figure 6.1: One of the two applied tracing aspects, i.e. the one for non-void procedures.

The last bullet deserves the most attention, as it touches the core topic of this dissertation: How does one reconcile the introduction of AOP technology at the source code level with the build system? The next section discusses the application of AOP in the source code, whereas Section 6.3 considers the implications on the build system.

6.2 Application of AOP in the Source Code

We first present the aspects which have been used in the Kava case study. Then, we evaluate whether Aspicere has fulfilled the L1 goal.

6.2.1 Trace and Pointer Guard Aspects

To enable the dynamic analyses and the test coverage to reconstruct the program call tree, we only need a program trace which contains call- and return-sequences, i.e. a message upon entry into a procedure, and one on exit. The easiest way to do this is by writing a `before`- and an `after`-advice. To stress-test Aspicere1's template parameters, we have used `around`-advice instead. Figure 6.1 shows one of the two resulting advices. We need two advices, because we have to store the procedure's return value into a temporary variable (line 13). This only works if

there is a return value, i.e. the return type is not `void`. The latter condition is explicitly added to the pointcut on line 4.

The body of the tracing advice is straightforward, as it only needs to write out a message to a file before (lines 9–11) and after (lines 15–17) the advised join point (line 13). We have not dealt with errors originating during the execution of the advised join point (call to `proceed`). The join point context `struct` (bound to `Jp`) is used to access the name of the advised function and the file in which it is implemented. The reason why the file pointer is opened (line 6) and closed (line 17) during each advice execution will be explained later on. Note that the name of the trace file is not hardcoded into the advice, but instead is specified separately as meta data and bound during join point matching on line 5. This makes the advice body reusable. The pointcut uses a complex regular expression on line 2 which states that calls to any procedure except for those of the standard `printf` and `scanf` families should be advised. Although we are interested in calls to the standard library, these two families of procedures introduce too much noise. More elaborate control of the scope of aspects can be obtained by using weaving meta data, as we show in Chapter 8. The choice to advise `call` join points instead of `execution` join points is governed by the need to record access to libraries. Because no function pointers were used in the code base, this did not introduce additional impreciseness.

As a side track, during the case study we have discovered the usefulness of an aspect to guard calls to `atoi`, `atol` and `atof` against null pointer arguments. On the original UnixWare platform on which the Kava system had been developed, null pointer arguments for these procedures were gracefully ignored. After migration to Linux, subtle errors appeared. It turned out that the C standard library there did not cope well with null pointer arguments. Kava initially solved the problem ("temporarily") by redeclaring `atoi` and its colleagues as a macro which checks the arguments first before calling the actual Linux implementation. This approach clearly works, but is not maintainable or evolvable. The advice of Figure 5.13 on page 173 obtains the same effect, but in a localised and concise way. It represents a simple, non-intrusive refactoring of the base code using aspects.

6.2.2 Validation #1: Aspicere Meets Goal L1

All four requirements of goal L1 have contributed to the aspects applied in the Kava case study. Aspicere's pointcut language and generic advice have enabled the concise notation of tracing and pointer guard aspects, and weave-time meta data has been applied to make abstraction of the name of the trace file.

This is promising for future reverse-engineering experiments, which should try to gather more focused tracing data. The current aspects trace every procedure call in the system, which generates too much information. By encoding the re-

sults of earlier dynamic analysis as Prolog facts and by using dynamic pointcuts like `cflow` or even temporal ones [70], the scope of the tracing aspects could be better controlled and more concrete information could be retrieved. Besides more focused tracing, reverse-engineering aspects could of course perform more work themselves than only writing out messages to a file. Care is needed, however, to prevent the aspects from disturbing the actual things which are measured (Heisenberg principle). The next section discusses the impact of AOP on the Kava build system.

6.3 Impact on the Build System

This section considers the consequences of the introduction of AOP technology on the build system. Whereas the Kava case has confirmed the design choices behind Aspicere (Section 6.2.2), it is especially the interaction between the Aspicere1 weaver and the development environment which has sparked the most interesting results. Aspicere1 is a source-to-source weaver which transforms one C compilation unit at a time (Section 5.4.1). The input file has to be preprocessed first to resolve all `#include`-statements, conditional compilation, etc. The output of each weaver invocation is passed to the normal C compiler. Eventually, the system linker combines all compiled modules into a library or executable. Hence, integration of Aspicere1 into the Kava build system requires the weaver to be invoked in between the C preprocessor and the C compiler. This section shows that this is not as easy as it may seem.

More in particular, we have observed the following five problem areas:

1. physical integration of Aspicere1 with the build process
2. the notion of "whole program"
3. the influence of C language features
4. build time increase
5. run-time overhead

We discuss these five issues in detail. Afterwards, the next two sections summarise how this case study validates the four roots of co-evolution and MAKAO's ability to support in understanding and managing the co-evolution of source code and the build system.

6.3.1 Integration of Aspicere1 with the Build Process

The Kava system uses GNU Make to automate the build process. Historically, all 272 makefiles have been hand-written by several developers. During a recent

```
1  $(CC) -c -o file.o file.c
```

(a)

```
1  $(CC) -E -o tempfile.c file.c
2  cp tempfile.c file.c
3  aspicere -i file.c -o file.c \
4           -aspects $HOME/kava/aspects.lst
5  $(CC) -c -o file.o file.c
```

(b)

Figure 6.2: (a) Original makefile snippet for .c files. (b) After transformation.

```
1  .ec.o:
2     $(ESQL) -c $*.ec
3     rm -f $*.c
```

(a)

```
1  .ec.o:
2     $(ESQL) -e $*.ec
3     chmod 777 *
4     cp `ectoc.sh $*.ec` $*.ec
5     $(ESQL) -nup $*.ec $(C_INCLUDE)
6     chmod 777 *
7     cp `ectoicp.sh $*.ec` $*.ec
8     aspicere -verbose -i $*.ec -o
9              `ectoc.sh $*.ec`
10             -aspects $HOME/kava/aspects.lst
11    $(CC) -c `ectoc.sh $*.ec`
12    rm -f $*.c
```

(b)

Figure 6.3: (a) Original makefile snippet for .ec ("esql") files. (b) After transformation.

migration operation from UnixWare to Linux, a considerable number of makefiles, but not all, have been re-generated with the help of "automake" (see Section 2.1.3.2). As a consequence, the structure of the makefiles is still heterogeneous, which is a typical phenomenon in practice. This heterogeneity has complicated the integration of Aspicere1 into the Kava build system, because we did not have a tool like MAKAO at our disposal at that time. Hence, the only way to understand the makefiles was by manual inspection.

6.3.1.1 Necessary Changes to the Makefiles

The makefiles clearly show that the build system uses a "recursive make", as makefiles in directories which only contain subdirectories explicitly transfer build con-

trol to the makefiles in the subdirectories. This pattern has been acknowledged later when we applied MAKAO on the Kava build system, as reported in Section 3.5.1.1. However, we had no idea about the actual structure of the source code and the build system, i.e. the mapping of source code to build system components. Although we only had to focus on the TDFS application, the actual extent of this system was not clear. We did not know which executables were part of TDFS, which object files and libraries were linked into it, etc. Hence, we conservatively decided to weave the tracing aspect of Figure 6.1 in the entire code base. A second problem was the uncertainty about the tools in use. Occasionally, we have encountered unknown messages and errors in the build trace. While some of these had to do with dead code (Section 3.5.2.1), others originated from tools like e.g. Informix "esql" (embedded SQL) or report generation tools. Only after manual inspection of the build trace the complete collection of build tools became clear. To summarise, we only had a partial understanding of how the Kava build system worked.

6.3.1.2 Wrapping the Compiler does not Work

The uncertainty about the build system has turned out to be a big problem when trying to physically integrate Aspicere with the C compiler. We have first tried the obvious solution: creating a wrapper script around the C compiler (GCC) which first invokes Aspicere1 before pursuing the actual compilation. This is one of the most popular and conceptually simple ways of integrating source-to-source or bytecode weavers like AspectC++, KLASY, ACC, AspectJ, etc. into a build system. The reason for the success of wrappers is the seemingly ease of using them instead of the wrappee (in this case: GCC), as this often involves just changing a build variable like `CC` or a symbolic link. The C4 people e.g. have been able to weave advice in the Linux 2.6.12.3 kernel like this[3]. However, there are some pitfalls too. The instructions of the SWAG toolkit [222], a utility kit from the research group of Michael Godfrey [228], explicitly warns that "In systems that use many makefiles, or that are specifically dependent on the value of the $CC variable, the substitution task may be considerably more difficult, and additional modifications of the makefile may be required.". A second problem is that most wrappers begin as simple Bash scripts, but quickly become very complex, especially when trying to make them robust w.r.t. the accepted program arguments and to the various versions and platforms supported by the wrappee. Third, when wrappers are being integrated, it is often not clear whether the wrapper has been installed exactly where it is needed, or whether some commands still call the wrappee directly. Fourth, there can be strange, undocumented interactions between wrappees which precludes correct functioning of the wrappers.

[3] `http://c4.cs.princeton.edu/files/Patches/Makefile.patch`

In the Kava system, we have encountered most of these problems. First of all, the non-uniformity of the makefiles did not guarantee that assigning the wrapper to `CC` would suffice to redirect all invocations of GCC to the wrapper. Some compiler invocations do not use environment variables to abstract over the compiler name, whereas other ones use different environment variables than `CC`. This problem could be circumvented by physically replacing the actual compiler programs themselves by symbolic links to our wrappers. This would cause more serious issues, however. The most important one is that the other compilation tools which were used besides GCC, like "esql", internally call the original C compiler. As we had replaced this by our own wrapper, the embedded SQL files eventually would be woven twice, i.e. once by the "esql" wrapper and a second time by the GCC wrapper script. We eventually gave up on using a wrapper script.

6.3.1.3 Regular Expression-based Transformation lacks Context

Instead, we have built a small tool to parse the makefiles via regular expressions and to make the necessary adaptations. A typical example of this transformation is shown in Figure 6.2a and Figure 6.2b. What the transformation does is that instead of the single compilation command of Figure 6.2a, the preprocessor should be called first (line 1), followed by a destructive[4] renaming of the preprocessed file to the original C file (line 2), invocation of Aspicere1 (lines 3–4) and the original compilation command (line 5). The aspect configuration, i.e. the configuration of which aspects which should be woven by the weaver, was stored in a central location.

The context-free regular expressions were not able to deal with minor and major variations in compiler commands, whereas we had to extract the name of the input file from the commands to compose the backup and Aspicere commands. This was an important source of problems. The situation becomes even more difficult when e.g. Informix "esql" preprocessing needs to be done (see Figure 6.3a and Figure 6.3b). As Aspicere1 requires preprocessed source code as input, we need to perform the extra commands shown on lines 2–7 and line 12 in Figure 6.3b to only invoke esql's preprocessor, followed by compilation with the normal C compiler. Notice how we assume that the environment variable $(C_INCLUDE) has been defined on line 5. If this variable is not defined or another variable is used to abstract over header file directories, the regular expression script fails.

Using our scripts to alter the makefiles took a few seconds to run. Detecting where exactly our tool failed (due to heterogeneously structured makefiles) and making the necessary manual adaptations took several hours. To truly automate the above integration of Aspicere1, we would have needed at least the following context information:

[4] We had backed up the original source code, so in case of an error we could easily rollback.

- the mapping of source code components to build system components
- names of all tools in use
- environment variables defined in each build script

6.3.1.4 MAKAO is able to Help

In Chapter 3, we have seen how MAKAO can provide us with this information. We have described the Kava build system in Section 3.5.1.1. The graph visualises the build-time dependencies of the TDFS application, i.e. the object files and libraries it depends on. This information would suffice to determine the scope of weaving, assuming that no dynamic libraries are used or other executables invoked. Probably, filtering would be needed to filter out too much detail from Figure 3.4a. The names of tools in use and the defined environment variables can be identified via querying, similar to the Tool Mining query in Section 3.5.2.2. In fact, the re-engineering example presented in Section 3.5.5 on page 100 could be applied directly to the integration of Aspicere1 in the Kava case. The join points (C compiler invocations) should be restricted to the extent of the TDFS system identified above. Before each join point, advice should be executed which performs the makefile code of Figure 6.2b or Figure 6.3b. The robustness of the MAKAO aspects to join point variability makes errors during the integration of Aspicere1 less likely, but verification (Section 3.5.4) could be applied to be absolutely sure about this. We are confident that if MAKAO had been around at the time of the case study, the integration problems discussed above could have been solved easier. Unfortunately, there were more problems to solve, which we discuss in the next section.

6.3.2 The Notion of "whole-program"

The Kava experiment gives evidence of how the changes introduced by AOSD on RC1 (Section 2.4.2) and RC2 (Section 2.4.3) have an impact on the build system, and how inadequate support to deal with this impact in the build system may backfire on the aspects at the source code level. We illustrate this via the problems encountered with Aspicere1's partially transformed aspects.

6.3.2.1 An Illustration: Partially Transformed Aspects

As we have seen in Section 5.4.1, Aspicere's weaver transforms aspects into C compilation units. The advice constructs are transformed into C procedures (the so-called "advice instances"), whereas all advised procedure calls in the base code are replaced by (indirect) calls to the right chain of advice instances. So far, so good. An aspect module has been transformed completely if all base code modules

with which the aspect has been composed have been woven into. The resulting transformed aspect module should be compiled and linked with every object file it has been composed with, as these contain calls to the aspect's advice instances. The biggest advantage of this weaving system is that aspect state is unique and is accessible to all advice instances. In our case, this would mean that a global file pointer variable could be added to the aspect which could remain open throughout the whole program execution and would only close during program shutdown.

However, this mechanism is fundamentally flawed, for a number of reasons. The most important one is that knowledge of the build system architecture is required to realise this weaving approach, i.e. a concrete overview of how generated build artifacts like libraries and executables are organised and used in the build system. We give an example of this. Consider a subdirectory of which the source code yields a library, and a second subdirectory of which the source code is linked with the library into an executable. By virtue of the adaptations in Figure 6.2a and Figure 6.2b, the tracing advices (and the null pointer guard) are woven into all source files of the first subdirectory. Then, the generated object files and the partially transformed tracing aspect are linked into the library. Next, the source files of the second subdirectory have to be woven. Because the same tracing aspect is used throughout the whole system (it has been stored centrally), the partially transformed aspect of the library can either be reused during weaving of the source files of the executable, or can be thrown away and rebuilt from scratch.

The first approach is the easiest one to implement, as it requires no special makefile transformation in addition to those of Section 6.3.1. However, because part of the transformed aspect overlaps with the version linked into the library, we encounter duplicate symbol definition problems[5]. Even if the partially transformed aspect would not be linked into the library, but only into executables, the partially transformed aspect eventually would grow too big to be practical. This would defy the actual point of partially transformed aspects, i.e. sharing state between advice and reuse advice instances. Hence, the first approach is not useful.

The second approach starts a new partially transformed aspect within each directory. Unfortunately, directories can yield more than one executable or library, with several source code modules reused among them. On the other hand, source code of multiple directories can be combined into one system too. Making sure that each application gets its own copy of a partially transformed aspect and is linked with it, drastically complicates the regular expression-based makefile transformer. Without precautions, some systems can end up with three global file pointers instead of one, each used within a subcomponent of the system without access to the other ones. Each of them would be writing to the same physical file, causing synchronisation problems. It is clear that this has semantic consequences for the

[5] An error which happens during linking if multiple definitions of the same symbol are found. In C, multiple declarations will be accepted by the compiler, but there should only be one definition of a procedure.

source code.

Taking all these factors into account, we eventually have taken the decision to add a "legacy mode" to Aspicere1. When using this mode, no partially transformed aspect is used anymore. The generated advice instances are added to the specific base code module which is woven into. This eliminates all possibilities for reuse of advice instances, but avoids too complex makefile changes. No base code module needs additional aspect modules anymore. Unfortunately, this rules out globally shared state as well. This is the reason why the advice of Figure 6.1 manages its own file pointer instead of reusing a global file pointer variable. In other words, build system problems backfire at the aspect in the source code.

6.3.2.2 Defining the Notion of "whole program"

These problems seem to be caused by questionable weaver implementation choices, but there is actually a much more fundamental cause. Aspect composition can be described as inversion of dependencies (see Section 2.4.3). Source-to-source and bytecode weavers (like AspectJ) actually reinstate a tangled and scattered version of the composed system by distributing the aspect logic back across the base code. The latter action of distributing is what causes the problems outlined in this section, as it implicitly assumes knowledge of the range of the distribution. This range is either the predefined scope of the aspect (if this has been explicitly specified), or the "whole program" [135], i.e. Griswold et al.'s "global configuration" [105].

In the first case, the scope makes the issue of using partially transformed aspects easy to solve. The second case on the other hand requires knowledge about the source code components which constitute the "whole program". RC2 has shown that programming-in-the-large [60, 84] usually is the build system's responsibility, by outlining the boundaries of programs and libraries, i.e. the build architecture. This means that this precious information is buried inside the makefiles. If the notion of "whole program" is not made visible somehow, it should be reverse-engineered or re-specified. Without tool support, wrong assumptions can be made in doing this, which cause the problems described in Section 6.3.2.1. This has a direct impact on the semantics of the aspects, as the new notion of "whole program" may either acknowledge or refute the aspect developer's assumptions. To summarise, the combination of whole-program reasoning (RC1) and inversion of dependencies (RC2) require explicit knowledge and control over the build architecture to support the aspect developer.

6.3.2.3 Supporting the Notion of "whole program"

Most source-to-source and bytecode weavers have the additional disadvantage that even when the notion of "whole program" is defined, this cannot directly be exploited by them because they process only one file at a time. The latter approach

has been the default mode of compilers for years, and blends with the "trickle-down" [9] recompilation offered by "make". Weavers which try to continue this tradition either have to make assumptions during each compilation, or they should store relevant state in between compilations, contrary to the usual stateless nature of compilers. A straightforward way to tackle this problem is to incorporate inter-weaving communication into the weaver, like e.g. the join point repositories of AspectC++ and Compose*, or Aspicere1's partially transformed aspects. As such, the weaver can keep track of matched join points, generated program artifacts, etc.

The management of the repository can prove difficult, however. First, it is almost impossible to know whether the repository state is consistent with the current compilation of the system or dates back to previous, possibly corrupted build runs. It is very common to halt the build when a developer notices compiler errors flying by. Removing the repository file before each build is not an option, as this likely erases valuable information for incremental weaving or for resuming interrupted builds. Another problem is that subcomponents of a system might require separate repositories, as we have discussed for the partially transformed aspects. The right location of the repository could be dependent (and hence vary) on the particular configuration of the system, i.e. which files should be built, and the resulting component (library versus executable). Upon switching between configurations, old repository files of earlier compilations could accidentally become active again. Unless inter-weaving communication is tightly controlled, non-trivial build or even run-time errors might ensue. Hence, RC1 forces source-to-source and bytecode weavers to emulate the notion of "whole program" by using techniques like inter-weaving communication. Without knowledge of the build architecture, these techniques are hard to manage.

Note that there are other kinds of weavers besides source-to-source and bytecode-weavers. Run-time weavers (Section 5.2.2) and load-time (VM) weavers [25, 24, 26, 100, 76] are dynamic weavers, in the sense that they operate during program execution and hence offer rapid feedback. Unfortunately [79], binary code is too low-level to provide sufficient join point context or to enable portable run-time weaving. Load-time weavers do not have these problems, but they are not used commonly in legacy systems. Standard linkers typically exhibit the same limitations as run-time weavers, but Aspicere2's choice for a higher-level bitcode IR instead of machine code improves platform-dependence and the richness of the program representation (Section 5.4.2). Hence, static link-time and source-to-source weavers are well-founded choices in legacy systems. This means that the problems they face during integration in the build system are representative for introduction of AOSD in legacy environments.

6.3.3 The Influence of C Language Features

Even though the Kava system has been migrated to a modern Linux platform, some remains of the original non-ANSI C implementation are still visible in the system. In non-ANSI C (also called "K&R" C), procedure declarations with an empty argument list are allowed, even if the procedure expects arguments. The actual declaration of the arguments can be postponed to the procedure definitions. This is not just a minor detail, as this means that source-to-source weavers have to infer the type of procedure call arguments from the context of the call, unless the weaver can access the procedure definitions. This, however, requires extra care in integrating the weaver into the build system, because typical source-to-source weavers only process one file at a time (see Section 6.3.2.3). Type inferencing on the other hand increases build time and risks imprecise or even incorrect type information.

For the reasons outlined in the previous two sections, Aspicere1 uses the type inferencing approach to deal with K&R C code. However, because of time restrictions, we have not pursued handling every possible type inference case and instead have tolerated a number of join points to be "skipped", i.e. not advised. Especially complex `struct` members passed as procedure argument have caused us trouble. Eventually, we have advised 367 source files, of which 125 contained some skipped join points. Of the 57015 discovered join points, there were only 2362 filtered out, or a minor 4 percent. Several screenings of the code confirmed that calls to the same small group of procedures were responsible for skipped join points. These procedures turned out to be very low level, not even part of the business logic, and therefore ignoring them did not impact our analyses.

Note that macros and conditional compilation did not cause any trouble for Aspicere1, as Aspicere1 works on preprocessed code. This however requires pointcuts to be expressed in terms of the base code into which macros expand instead of in terms of the macros. In the Kava case, this was not a restriction.

To summarise, undisciplined features of legacy programming languages like incomplete procedure declarations or preprocessor usage (more on this in Chapter 10) represent problems for weaver technology. The compromises which have to be taken on this level have repercussions on the aspect developer's understanding of the program.

6.3.4 Build Time Increase

The original compile cycle of the Kava application suite (407 C modules corresponding to 453 kSLOC) takes more or less 15 minutes[6]. When Aspicere1 is included in the build process, it takes 17 *hours* and 38 minutes to complete. The reason for this substantial increase in time can be attributed to several factors. The

[6]Timed on a Pentium IV, 2.8 GHz running a vanilla Slackware 10.0 system.

most fundamental one is that a source-to-source weaver adds additional functionality on top of a build in the form of transformation of the base code. This means that the code is parsed, transformed and written out to file twice.

Apart from this inherent slowdown, Aspicere1 is a research prototype, and no production system. The framework on which it is built [202] combines many different Java-based technologies (XML, Prolog engine, etc.), and was especially geared towards rapid prototyping. We did some initial profiling, and the biggest time loss occurs during join point matching, i.e. when the Prolog engine kicks in. Aspicere1's on-demand program reification and heavy backtracking slow down this phase considerably. A second source of slowdown (and memory consumption) is the XML generation and processing. We have used the standard XML DOM implementation, so there is still room for improvement in this area.

Because we did not frequently alter the base and aspect code, but rather spent time with debugging Aspicere1, incremental weaving would not have helped us to reduce the weaving time. The time stamps of object files were always newer than those of source code files, but this did not tell anything about the correctness of the woven code produced by Aspicere1. Hence, if a weaving error was detected, the source code was synchronised with a backup and weaving was performed from scratch again. RC3 was not an issue in the Kava case study.

6.3.5 Run-time Overhead

Not only the compilation time has been influenced by our aspect weaving process. The run-time speed has heavily slowed down as well. The TDFS scenario we used normally runs in about 1.5 hours. With the tracing advice, it took 7 hours to complete due to the significant file I/O. This file activity is inherent to a tracing aspect, but the inability to reuse a unique file pointer throughout the application has had a large influence on this as well. The overhead of the function pointer manipulation inserted by Aspicere1 is negligible compared to the file I/O overhead. The combination of long build and execution time has a negative impact on the debuggability of an AOP system.

6.4 Validation #2: Roots of Co-evolution Experimentally Confirmed

The five build problems described in Section 6.3 have given evidence of RC1 and RC2, and also have shown how MAKAO is able to support understanding and management of co-evolution of source code and the build system. This section summarises the evidence of the two roots of co-evolution, whereas the next section considers tool support.

The main lesson from the Kava case study is that there is an impedance mismatch between the notion of modules in traditional "make"-based build systems and the "whole program" implied by AOP (RC1). The former implies a build style based on Parnas' notion of modules [187] in which each module is compiled in separation and type-checked based on imported interfaces. Only at link- or class loading time, compiled modules are composed. Aspects on the other hand, require a view of the whole system, not just one module at a time. We have seen how the definition of "whole program" is specified by the build system and as such is not easily accessible or understandable. Redefinition of this concept without tool support introduces impreciseness which has consequences for the semantics of the aspects, as illustrated by the desired global file pointer. This mismatch can either be resolved by adapting a build system to aspects (AOP-aware build system), aspects to the build system or finding another common base. The middle solution is the one in use today, but it requires numerous compromises between language semantics and the integration of AOP in the development environment.

Second, the inversion of dependencies (RC2) offered by AOSD, combined with whole-program reasoning (RC1), has consequences for weaver technology. The popular source-to-source weavers, which continue the tradition of compiling one source file at a time before linking the results together, have problems to deal with this. They require workarounds like inter-weaving communication to have knowledge about the whole build configuration while processing each source file. These workarounds are not easy to manage, as they require explicit control and understanding of the build system. Without this, compromises have to be made with possibly disadvantageous effects on the semantics of aspects, as shown by the partially transformed aspects in Section 6.3.2.1.

To summarise, the Kava case has given evidence that RC1 and RC2 indeed lead to build problems when AOP technology is introduced into a legacy system.

6.5 Validation #3: MAKAO Achieves Goal T2

We have seen how ignorance about the build architecture (RC1 and RC2) has caused problems for physical integration of the weaver in the build system, understanding and definition of "whole program" and management of inter-weaving compilation. During the actual Kava case study, MAKAO did not yet exist. Section 3.5 has shown however, how MAKAO's visualisation, querying, filtering, verification and re-engineering support the understanding and management of co-evolution of source code and the build system. Visualisation offers an intuitive overview of the possible definition of "whole program", which can be improved by filtering out redundant dependencies or subsystems. Querying provides access to specific build context like compilers and values of build-time variables. Re-engineering support facilitates invasive, robust changes to the build system. The

correctness of the re-engineering can be verified with the verification support, but the latter has not been done yet. Many build problems could have been prevented by the use of MAKAO.

6.6 Conclusion

This chapter has identified evidence of problems of co-evolution of source code and the build system caused by RC1 and RC2 in a legacy system in which AOSD is introduced. Many of these problems could be solved by MAKAO. On the source code level, Aspicere fulfils goal L1.

The Kava case study has focused on a reverse-engineering case, i.e. an evolution step which introduces AOP technology into an existing build system to gather information from the system, possibly as preparation of further re-engineering activities. The next chapter presents a similar case study, but uses Aspicere2 instead. MAKAO is used from the start to mitigate build problems.

Divide et impera!

Philippus II of Macedon

7

Case Study 2: Component-aware Reverse-engineering of Quake 3 using Aspects

THIS chapter describes a second case study in which aspects are introduced into a legacy system, i.e. the Quake 3 video game. Similar to the Kava case study of the previous chapter, the aspects are used for reverse-engineering. Instead of Aspicere1, the Aspicere2 link-time weaver is used in combination with MAKAO. Evidence of RC1, RC2 and RC3 is found and the tool and aspect language support provided by MAKAO (goal T2) and Aspicere (goals L1 and L2) is evaluated.

Similar to the previous chapter, we first describe the case study (Section 7.1). Then, we discuss the actual application of aspects in the source code (Section 7.2). Afterwards, we discuss the impact on the build system of the introduction of AOP technology (Section 7.3). The experimental evidence of co-evolution problems explained by RC1, RC2 and RC3 is summarised in Section 7.4, followed by the evaluation of the support offered by MAKAO for goal T2 (Section 7.5) and by Aspicere (Section 7.6) to deal with co-evolution of source code and the build system (goal L2). Section 7.7 presents the conclusions of the Quake 3 case study.

7.1 Rationale behind the Case Study

As introduced in Section 3.5, Quake 3 Arena is a popular, commercial 3D video game [191][1] written in C. Like older versions of the game, it has been released as open-source. A community has formed to maintain and improve the released system[2]. We have focused on revision 1041 (released in February, 2007) made by this community.

Similar to the Kava case study, we were interested in reverse-engineering the internal structure of Quake 3 via dynamic analysis. Contrary to the Kava case, we did not intend to unravel the architecture of the whole system. Rather, we have focused on each component of the Quake 3 engine in separation, to find out the most important functionality they provide and the internal cohesion of the components. The same analyses as for the Kava system have been used for this, based on one execution trace per main component. We do not elaborate on the results of this analysis. Instead, we discuss how the main components have been determined and how we have integrated Aspicere2 into the build system. The next section highlights the identification of Quake 3's main system components and discusses the aspects which have been used.

7.2 Application of AOP in the Source Code

First, we discuss how we have identified the main components of the Quake 3 system. Then, we present the tracing aspect we have used. Afterwards, we evaluate whether Aspicere has been able to fulfil goal L1.

7.2.1 Determining the Main System Components

Before embarking on the Quake 3 case study, we had no previous experience with the source code and build system. Available documentation [240] and skimming through the source code suggest that the system consists of a client-server architecture and contains components for rendering, artificial intelligence (AI), sound, etc. There is even a VM to prevent game cheaters from bringing down a running game engine. However, there was no clear overview of dependencies and internal structure.

To resolve this, we have compiled Quake 3 and have studied the build dependency graph of Quake 3 with MAKAO. We have shown this build DAG in Figure 3.6a on page 92. Despite the source code complexity (222 kSLOC spread over 487 files, without counting assembler code), the Quake 3 build system only has six build scripts (2000 SLOC), of which only one contains the actual composition

[1] `http://www.idsoftware.com/games/quake/quake3-arena/`
[2] `http://ioquake3.org/`

dependencies of the whole system in the form of a "non-recursive make". This seems a much simpler case than the Kava one (which had more than 45 times as many build scripts), but without MAKAO we would not have realised as quickly how the build architecture is structured.

As explained in Section 3.5.1.3, Quake 3 has a very clear high-level structure consisting of three dynamic libraries and one main executable. From the names of the source code files making up these libraries and executable, it seems that the "ui" library implements the logic for the main and in-game menus (client side), "game" provides layout and AI (server side), "cgame" manages communication with the server and drawing (client side), and the executable provides rendering, sound, etc. (client side). As these four components constitute the complete Quake 3 system, it makes sense to consider them as the main systems components and to apply the dynamic analyses on each of them. Based on the resulting recovered knowledge, subcomponents could be identified and used as the target of further analysis.

It is clear that to be able to gather information for each component in separation, the knowledge of the extent of the build architecture has to be used to define the scope of the reverse-engineering aspects. A separate trace should be generated per dynamic library or executable. The particular scenario to execute is easy to determine: playing a game. This more than likely exercises all Quake 3 components and gives a representative idea about the dynamic behaviour. The next section presents the aspects used to gather the four execution traces.

7.2.2 The Tracing Aspect

To capture the four program traces, we principally have reused the Kava tracing aspect of Figure 6.1 on page 197. However, the clear overview of the build architecture has made it easy to identify the high-level build components and hence, to understand and define the notion of "whole program" in the Quake 3 system. As a consequence, many of the problems discussed in the previous chapter could be avoided, especially the absence of a global file pointer. As we need four program traces, one for each Quake 3 component, we could either declare four file pointers and weave the aspect into all four components at once, or restrict the notion of "whole program" to the individual components. The former approach is not reusable, as it would be specialised to the Quake 3 system and would not cater for systems with e.g. six components. Hence, we have chosen for the second approach, which considers each component as a "whole program".

The resulting aspect is shown in Figure 7.1. Lines 4–23 contain the file pointer management code. The idea is that the `aspicere2_getFilePtr` method provides access to the file pointer. This lazily initialises the file pointer of line 4 if needed. During initialisation (lines 13–16) a cleanup handler is registered (line 14) which flushes the log file and closes it if the file pointer has ever been initialised

```
#include <stdlib.h>
#include <stdio.h>

FILE* aspicere2_fp=0;

void aspicere2_cleanup(void){
  if(aspicere2_fp){
    fflush(aspicere2_fp);
    fclose(aspicere2_fp);
  }
}

FILE* aspicere2_init(char* fileName){
  atexit(aspicere2_cleanup);
  return fopen(fileName,"a");
}

FILE* aspicere2_getFilePtr(char* fileName){
  if(!aspicere2_fp){
    aspicere2_fp=aspicere2_init(fileName);
  }
  return aspicere2_fp;
}

ReturnType trace(TYPE ReturnType,char* FileName,
            char* FunctionName,char* LogFile) around Jp:
  execution(Jp, FunctionName)
  && type(Jp,ReturnType)
  && !!is_void(ReturnType)
  && program_name(LogFile)
  && filename(Jp,FileName){
    FILE* myfp=aspicere2_getFilePtr(LogFile);
    ReturnType res;

    fprintf (myfp,"before ( %s in %s ) \n",
                  FunctionName,FileName);
    res=proceed();
    fprintf (myfp,"after ( %s in %s ) \n",
                  FunctionName,FileName);

    return res;
}

/* Similar advice for void-procedures. */
```

Figure 7.1: The build architecture-aware tracing aspect applied to Quake 3.

by the code on lines 6–11.

The advice for non-`void` procedures is shown on lines 25–42 of Figure 7.1[3]. It now accesses the global file pointer on line 32 via a lazy getter method. A second difference with the Kava aspect is the use of context obtained via a binding instead of via the join point context `struct` (lines 35–36 and 37–38), as this is more efficient (speed and memory). Third, we advise `execution` join points instead of `call` join points, because the Quake 3 code contains a lot of function pointer callbacks. The most important difference with the Kava advices, is the determination of the right log file name based on the component into which the advice is woven (line 30). To accomplish this, we need to know for each join point shadow the build component it belongs to. This information is only available at build time. Hence, we need Aspicere's support for integration of the build system with the logic fact base to communicate the build system architecture to the advice. In the advice of Figure 7.1, the `program_name`-predicate (line 30) binds to the `LogFile` variable the name of the library or executable to which the current join point shadow belongs, which is used on line 32 as the name of the file into which the advice will write its output. The library and executable names form a part of the reification of the build architecture as logic facts.

7.2.3 Validation #1: Aspicere Meets Goal L1

The integration of the build system with the logic fact base enables to write compact advice which can exploit knowledge of the high-level structure of the base code. In the Quake 3 case study, this has enabled tracing advice to vary its output based on the particular build architecture. This advice can be reused in other systems as long as the corresponding build system is reified as logic facts and is accessible during weaving. The next section discusses the build problems and issues involved in introducing Aspicere2 into the Quake 3 build system.

7.3 Impact on the Build System

This section discusses the impact of the introduction of AOP on the build system. More in particular, we provide evidence of problems attributed to three roots of co-evolution (RC1, RC2 and RC3) and show how MAKAO and Aspicere have helped in understanding and managing build problems.

We consider the following build problems:

1. integration of Aspicere2 with the build process

2. communication between Aspicere2 and the build system

[3] Just as in the Kava case study, we do not show the advice for `void` procedures, as this is rather straightforward.

```
CC=gcc
...
DO_CC=$(CC) $(NOTSHLIBCFLAGS) $(CFLAGS) -o $@ -c $<
DO_SMP_CC=$(CC) $(NOTSHLIBCFLAGS) $(CFLAGS) \
          -DSMP -o $@ -c $<
...
$(B)/client/cl_cgame.o : $(CDIR)/cl_cgame.c; $(DO_CC)
...
$(B)/ioquake3.$(ARCH)$(BINEXT): $(Q3OBJ) $(Q3POBJ) \
                                $(LIBSDLMAIN)
    $(CC) -o $@ $(Q3OBJ) $(Q3POBJ) $(CLIENT_LDFLAGS) \
                $(LDFLAGS) $(LIBSDLMAIN)
```

Figure 7.2: Original build commands and rules in the Quake 3 makefiles.

```
override BUILD_DIR=aobuild
override LDD=link.sh \
             -aspects $(BUILD_DIR)/../aspects/aspects.lst \
             -modules $(BUILD_DIR)/../aspects/modules.lst
override LD=lto.sh
override CC=llvm-gcc -fno-builtin -g -emit-llvm
override O_FLAG=-O0
override ASPECTS=`paste -s -d\  \
                  $(BUILD_DIR)/../aspects/aspects.lst`
```

Figure 7.3: Local makefile which overrides the important makefile variables.

3. the influence of C language features
4. build time increase and incremental weaving
5. run-time overhead

7.3.1 Integration of Aspicere2 with the Build Process

The combination of MAKAO, the consistent build architecture of Quake 3 and Aspicere2's link-time weaving have made the integration of Aspicere2 into the Quake 3 build system much easier than for the Kava case. In fact, because the build system is easy to manage, we did not have to use MAKAO's re-engineering capabilities (Section 3.5.5). MAKAO's reverse-engineering features, especially visualisation and querying, have sufficed to find out the right places to change the makefiles and to obtain the required context. The latter was easy to find, because environment variables are consistently used, and there are many makefile comments.

```
1 $(B)/ioquake3.$(ARCH)$(BINEXT): $(LIBSDLMAIN) \
2                                 $(B)/ioquake3.$(ARCH)$(BINEXT).bc
3     $(LD) -o $@ $(B)/ioquake3.$(ARCH)$(BINEXT).bc \
4                 $(CLIENT_LDFLAGS) $(LDFLAGS) $(LIBSDLMAIN)
5 ...
6 $(ASPECTS): ;
7 $(B)/ioquake3.$(ARCH)$(BINEXT).bc: $(Q3OBJ) $(Q3POBJ)\
8                                    $(LIBSDLMAIN) $(ASPECTS)
9     $(LDD) -o $(B)/ioquake3.$(ARCH)$(BINEXT) $(Q3OBJ) \
10            $(Q3POBJ) $(CLIENT_LDFLAGS) $(LDFLAGS) \
11            $(LIBSDLMAIN)
```

Figure 7.4: Modified build commands and rules in the Quake 3 makefiles for weaving into the libraries and the executable.

Figure 7.2 shows the most crucial makefile variables and rules. Compiler names are hidden behind standard build variables (line 1) and combined with other variables into a complete compiler command, one for normal builds (line 3) and one for SMP[4] builds (lines 4–5). Object files are compiled via a simple rule like the one on line 7. They only depend on their corresponding source file and are compiled via the single `$(DO_CC)` command. Libraries and executables are built via a rule similar to the one on lines 9–12. They depend on a number of object files (`Q3OBJ` and `Q3POBJ`) and the SDL[5] library for graphics and sound support. The normal compiler's (`$(CC)`) internal linker is used to link the object files and library into a library or executable (lines 11–12).

The Quake 3 build system is really developer-friendly, as there is built-in support for overriding existing build variables in a makefile called "Makefile.local" (Figure 7.3). This has made it easy to override the `CC` variable to use the LLVM front end instead of GCC (line 6). This front end generates bitcode instead of machine code. Because the `CC` variable was also used for invoking the linker, we have replaced the use of `$(CC)` for linking by a new variable (`LD` on line 5) which points to an Aspicere2 script (`lto.sh`) which does the actual link-time weaving. As such, the original build rules for libraries and executables have almost remained the same (lines 1–4 of Figure 7.4). We only have moved the dependencies on the object (actually: bitcode) files and the SDL library to a separate rule (lines 7–11), which compiles the aspects, compiles the Prolog state (see Section 5.4.2 on page 187) and combines all bitcode modules into one link module (see Figure 5.20 on page 188). This is handled by the `LDD` variable defined on lines 2–4 of Figure 7.3. This rule also depends on the aspects which should be woven. The complex shell expression on lines 8–9 of Figure 7.3 merely concatenates all aspect

[4] For multiprocessor computer architectures.
[5] `http://www.libsdl.org/`

paths in the aspect configuration file with a space between them (more on this in Section 7.3.4).

To summarise, integration of Aspicere2 into the Quake 3 build system has been easy because of the clean build system, MAKAO and the link-time weaver. The latter simplified the integration because we only had to focus on the four rules for the executable and the three libraries instead of on the compilation of each source file. Note that a source-to-source weaver would have been easier to integrate than in the Kava case as well. However, the partially transformed aspect should have been used instead of the legacy weaving mode to obtain the global file pointer, which would have required more integration effort than with Aspicere2.

7.3.2 Communication between Aspicere2 and the Build System

To enable the integration of the build system with the aspect language, build-time data should be passed to the weaver. Ideally, the whole build dependency graph constructed by "make" should be reified as weave-time meta data. Likewise, all the available configuration data should be communicated. In general, this is not straightforward to achieve, as it requires modification of e.g. GNU Make to be able to extract the build dependency graph at build-time. A second problem is that the graph should be filtered to only retain the build components which correspond to actual source code components. For this, the semantics of the build architecture should be taken into account. Also, the extraction of the dependency graph at build-time should be able to deal with the "recursive make" idiom, in which multiple "make" processes are invoked without clear links between each other. Apart from efficiency concerns, this means that the weaver should be provided with the architecture of the whole build system, rather than the data of one "make" process. We have not yet embarked on an attempt to solve these problems in the general case.

In the Quake 3 system, however, we did not need the whole build architecture and configuration, only the names of the libraries and the executable which is currently woven into, i.e. the current "whole program" (RC1). As Aspicere2 is only called at link-time, it gets the name of the constructed library or executable as one of its program arguments. It suffices to change the `link.sh` script (used on line 2 of Figure 7.3) to parse the arguments with which it is invoked for the library or executable name[6] and to assert this as a logic fact when the Prolog state (see Figure 5.20) of the Aspicere2 invocation is compiled. This is straightforward to do and still generic, as libraries and executables are by definition high-level build components.

To summarise, the combination of restricting the notion of "whole program" to the library and executable which is currently being woven into, and asserting the

[6]Technically, it should only look for the string behind the `-o` switch.

```
1 $(B)/ioquake3.$(ARCH)$(BINEXT): $(Q3OBJ) $(Q3POBJ) \
2                                 $(LIBSDLMAIN)
3     $(WEAVE) -o $@ $(Q3OBJ) $(Q3POBJ) $(CLIENT_LDFLAGS) \
4               $(LDFLAGS) $(LIBSDLMAIN)
```

Figure 7.5: Original build rule and command for building a library in the Quake 3 makefiles.

name of the "whole program" as a logic fact before weaving starts has enabled us to integrate a restricted portion of the build architecture with the logic fact base. The restriction only passes names of build components (RC1), but not dependencies between them (RC2). Although limited, this case has shown that this kind of integration facilitates development of compact, reusable advice expressed in terms of the high-level base code architecture.

7.3.3 The Influence of C Language Features

Because Aspicere2 operates on a uniform bitcode representation, it does not need to deal with assembler code, the preprocessor, etc. The potential for duplicate definitions caused by name clashes is drastically reduced as well, as during weaving identity is determined by memory address, not by names. Type inferencing of K&R code is automatically taken care of by the LLVM framework.

This robustness to typical C problems has been valuable in the Quake 3 system, because it features e.g. 1 kSLOC of assembler code. Most of this is part of a custom VM implementation on top of which Quake 3 is built. This VM interacts either with the dynamic libraries or with a 32-bit RISC pseudo-assembly bytecode format. Its purpose is to form a kind of sandbox with a limited number of system calls, with the aim of preventing cheating or hacking[7]. Client and server logic contain system calls to the code which is running in the VM. Complex pointer arithmetic is used to dispatch to the right function definition and to fetch the system call arguments. This is all hidden from Aspicere2 because of the LLVM IR.

7.3.4 Build Time Increase and Incremental Weaving

Not all is perfect, however. In Figure 7.4 we have split the rules for linking libraries and executables in two. One is concerned with generating the bitcode link module (lines 7–11), while the other one does the actual weaving (lines 1–4). Originally, there was only one rule and one weaver command as shown on Figure 7.5. However, during development of the aspect and the debugging of some weaver errors, we have noticed that if none of the bitcode files which is part of the link module

[7] Quake 3 is mostly targeted at online multiplayer gaming.

has been regenerated (no source code changes), the rule would still link all bitcode modules together again and then weave the aspects into the new link module. The rule did not detect either whether a change had been made to some aspect. Hence, we have decided to split the rule in two parts. If the weaver fails, we do not need to rebuild the bitcode link module again. Only if a base code module or an aspect would change, relinking is necessary. We have added the dependencies on the aspects via the `ASPECTS` variable of Figure 7.3 (lines 8–9).

Unfortunately, two problems remain. First, the linking of bitcode modules is slow. This is an LLVM issue, but it can get tedious very quickly. From our measurements, weaving takes six times as long as normal compilation. A significant part of the time is spent while linking the bitcode modules. Second (and more fundamental), making a link-time weaver implement incremental weaving is not straightforward (RC3). Splitting the build rules in two has allowed to decouple weaving from linking, but if at least one base module has changed, the whole system needs to be relinked and re-woven. In a source-to-source weaver, the system would also need to be relinked, but only the changed base code module would have to be re-woven and recompiled. This "trickle-down" weaving [9] behaviour is automatically enabled by "make". However, this incremental weaving behaviour cannot be achieved for a link-time weaver, as linking corresponds to only one build action. Hence, whereas a source-to-source weaver benefits from file-level support by "make", as it aligns with single files, a link-time weaver has to deal with incremental weaving by itself.

Aspicere2 currently does not provide incremental weaving. The biggest hurdle to implement it, is traceability from the link module back to base code modules and the ability to unweave previously woven code. The former is needed to identify the regions inside the link module which have changed, whereas the latter tracks down the changes, reverts previous weaving attempts and only re-weaves in the affected regions. An additional problem is the invalidation of interprocedural static analysis results [52], which is used e.g. to implement the `cflow` pointcut [14]. If analysis previously has shown that a join point shadow will e.g. never lie in the `cflow` of another join point, these results could have become invalid by now. Hence, the weaver should now suddenly add code to invoke the advice at that shadow. This means that base code modules which have not been changed, can still be affected by the invalidated analysis results. This makes incremental weaving in a link-time weaver hard to implement. As incremental weaving is a necessity for practical use, there are still many opportunities for research in this area.

7.3.5 Run-time Overhead

Contrary to the build speed, the optimisations inside LLVM and the global file pointers per build component have resulted in relatively fast, playable woven code.

While generating the traces during a game, there was a noticeable delay, but the game was still possible to play. We have not made concrete measurements of the run-time overhead (like e.g. the number of frames per second). As tracing is an extremely scattered aspect with lots of file I/O, this is very promising with the eye on re-engineering aspects (as presented in the next case studies).

7.4 Validation #2: Roots of Co-evolution Experimentally Confirmed

We have found evidence for build problems and issues explained by RC1, RC2 and RC3 in the Quake 3 case study. The explicit knowledge of the architecture of the system has facilitated making an appropriate choice for the "whole program" boundaries in the system (RC1 and RC2), in this case the three libraries and the executable. The usage of a link-time weaver has resolved the problems encountered in the Kava case with the sharing of aspect state (global file pointer), hence this kind of problems associated with RC2 has not occurred. Reification of the build system components (RC1), but not dependencies (RC2), has allowed to specialise the tracing advice to the particular build component it was woven into.

On the other hand, incremental weaving (RC3) has shown to be a problem. Traditional "trickle-down" compilation [9] does not provide opportunities for speeding up the link-time weaver, as linking looks like only one atomic activity for "make". Incorporating incremental weaving into Aspicere2 is not straightforward, however, and requires means for mapping back to the base code modules, undoing previous weaving and dealing with static analysis results.

7.5 Validation #3: MAKAO Achieves Goal T2

MAKAO has enabled us to identify the main architectural components of the Quake 3 system, i.e. the three libraries and the executable. This knowledge has enabled us to estimate how Aspicere2's link-time weaver could be integrated into the build system, by virtue of visualisation and querying. Manual changes sufficed instead of applying MAKAO's re-engineering support. MAKAO has also helped to determine how build system information could be passed to Aspicere2 to facilitate the integration of the build system into the logic fact base.

7.6 Validation #4: Aspicere Meets Goal L2

Integration of build system information with Aspicere's logic fact base has enabled us to write compact, reusable advice for tracing a particular build component's

behaviour. This has shown great promise for advanced applications, where besides build components, also build dependencies should be reified as logic facts.

7.7 Conclusion

Just like the Kava case, the Quake 3 experiment corresponds to a reverse-engineering activity which is used to evaluate the effects of co-evolution when AOP technology is introduced in the build system. Evidence has been found of build problems associated with RC1, RC2 and RC3. Except for the latter problem (incremental weaving), MAKAO and Aspicere have been able to mitigate the problems caused by co-evolution of source code and the build system. Integration of aspect technology has been very straightforward in this case, both on the source code and build level. The high quality of the very compact non-recursive build system has also played a key role in this. Incremental weaving in a link-time weaver has shown to be a problem which cannot be solved from the outside.

In the next chapters, we consider three re-engineering case studies with aspects. Just as the past two case studies, they are used to find experimental evidence of co-evolution problems caused by the four roots of co-evolution, and to evaluate the ability of MAKAO and Aspicere to understand and deal with these problems.

There are still several crosscutting concerns for which no modular implementation strategy could be developed. The most obvious example is exception handling, which seems to defy a modular implementation because of its very detailed interaction with the control flow of the base program.

"Ideals" project end report [194]

8

Case Study 3: Extracting the Return-code Idiom into Aspects

THIS chapter[1] focuses on the refactoring of an idiom-based exception handling strategy in an existing C system using aspects. This work proposes an alternative solution for a problem described by Bruntink et al. [35]. They have unmasked various idioms used in the 15 MLOC C code base of ASML, the world's biggest lithography machine manufacturer, as crosscutting concerns [36, 33, 34, 35, 32]. A scalable aspect-based implementation for these idioms has not been identified yet. We have come up with a concise aspect-based implementation for ASML's exception handling idiom by judiciously combining join point properties, annotations and type parameters, to which we have added Aspicere's concept of *(delimited) continuation join points*. Our solution takes care of the error value propagation mechanism (which includes aborting the main success scenario), logging, resource cleanup, and allows for local overrides of the default aspect-based recovery. The highly idiomatic nature of the problem in tandem with the aforementioned aspect language concepts renders our aspects very robust and tolerant to future base code evolution.

We have not yet evaluated this approach on the actual ASML system. Nevertheless, application on the simple example program used by Bruntink et al. [35] already teaches us valuable lessons about the existence of build problems explained by the roots of co-evolution and also about the possible application of MAKAO

[1] This chapter is based on [8].

(goal T2) and Aspicere (goals L1 and L2) to solve these problems. First (Section 8.1), we describe the general context of the case study, followed by a detailed account of the aspect refactoring at the source code level (Section 8.2). Section 8.3 discusses the build system issues and problems which arise from the aspect refactoring and how they relate to the roots of co-evolution. Finally, validation of problems explained by the roots of co-evolution (Section 8.4), and of the ability of MAKAO (Section 8.5) and Aspicere (Section 8.6) to deal with these problems is examined. Section 8.7 concludes this chapter.

8.1 Rationale behind the Case Study

Exception handling is one of the more fundamental issues encountered in software development, especially for legacy programming languages like C or Cobol. C in particular lacks any dedicated means to tackle exceptions, hence over time people have resorted to all kinds of tricks to emulate them [131]: `setjmp`/`longjmp`, global error variables, signals, returning error values, etc. Bruntink et al. [35] have discussed a variant of the so-called "return-code idiom" as the preferred means for exception handling in the 15 MLOC C code base of ASML, the world's biggest lithography system manufacturer. There are estimates that this idiom represents up to 20% of the total source code, while on average two errors per kSLOC are related to it. The return-code idiom is part of a global idiom-based software development strategy for safeguarding the machines' reliability and functioning. There are also other idioms for checking null pointer function parameters, logging, timing, etc.

However, for idioms to work properly, developers need to be disciplined enough to use them correctly and consistently [34]. Bruntink et al. [32] have studied the variability of the trace idiom implementation in four components of the ASML system. They have found a surprisingly large variation e.g. in the way tracing is invoked (7.5% of functions have done it correctly) or types are converted into a string representation (57.5% traced correctly). While some of this variability corresponds to developer errors ("accidental variability"), much of it is intentional ("essential") to adapt the idioms to local needs. A second problem with idioms is that the choice for one particular idiom ties the code base almost exclusively to the chosen pattern, making it very hard to migrate to or experiment with other approaches. These problems suggest that something should be done to enforce correct usage of idioms, and to decouple idioms from the source code.

Many of the idioms, in particular the return-code idiom, have been identified as typical crosscutting concerns before [34, 157], as they exhibit excessive scattering (throughout the whole system) and tangling. Exception handling e.g. severely obscures normal flow. Aspects would in theory be able to enforce consistency among idioms instead of having to rely on disciplined programming, and to allow for experimenting with other implementations, as idiom implementation are

modularised. No proof has been made yet about the resistance of aspects to the introduction of new variability in the extracted idiom implementations, however. In any case, an aspect implementation of one of these idioms requires explicit means to model essential variability, because duplicated advices are unmaintainable [32].

We have come up with an aspect-based implementation of ASML's exception handling strategy. Our findings partially confirm the quote at the beginning of this chapter, because abstraction of the return-code idiom into an aspect requires a combination of concepts which are not mainstream in aspect languages. Without these features, writing a useful, flexible exception handling aspect for systems written in legacy programming languages is hard to achieve. Note that Bruntink et al. [32] independently have proposed new aspect features to implement the tracing idiom as aspects. These do not overlap with our features, but they are actually highly compatible with Aspicere, especially with its combination of a logic pointcut language and template parameters.

In this chapter, we describe our aspect-based implementation of the "return-code idiom" in Aspicere. More in particular, we:

- analyse the various concerns related to the ASML exception handling idiom;
- explain Aspicere's delimited continuation join points, which are required to model the idiom using aspects;
- present the aspects which co-operate to implement the idiom-based exception handling pattern;
- discuss the implications of this aspect-based approach on the co-evolution of source code and the build system.

8.2 Application of AOP in the Source Code

First (Section 8.2.1), we introduce the specific details of ASML's return-code idiom using the running example of [35], eventually identifying the core aspect of the problem. This concern is modeled using delimited continuation join points, which are introduced and applied on the running example in Section 8.2.2. Section 8.2.3 treats the logging part of the idiom using join point properties and annotations, whereas Section 8.2.4 looks at the resource cleanup concern. Finally, Section 8.2.5 validates whether or not Aspicere satisfies goal L1.

8.2.1 The Return-code Idiom

As its name implies, ASML's return-code idiom is based on the dedicated use of return values to pass error values up the call stack. As an extra requirement, errors have to be logged in a so-called "event log" for off-line exception analysis. In the

```
1 int f(int a, int** b){
2   int r = OK;
3   bool allocated = FALSE;
4   r = mem_alloc(10, (int**) b);
5   allocated = (r == OK);
6
7   if((r == OK) && ((a < 0) || (a > 10))){
8     r = PARAM_ERROR;
9     LOG(r,OK); /*root error*/
10  }
11  if(r == OK){
12    r = g(a);
13    if(r != OK){
14      LOG(LINKED_ERROR,r); /*linked error*/
15      r = LINKED_ERROR;
16    }
17  }
18  if(r == OK) r = h(b);
19  if((r != OK) && allocated) mem_free(b);
20  return r;
21 }
```

Figure 8.1: Running example which applies the return-code idiom [35].

following subsections we dive deeper into the actual code behind this idiom, based on the extensive reports of Bruntink et al. [35, 33].

8.2.1.1 Specification of the Idiom

Each procedure allocates a special local variable in which the current error status (initially `OK`) is stored. More than one variable is necessary in some cases like parallel execution or errors during resource cleanup. Whenever an error occurs directly inside a procedure, i.e. *not* within a called procedure, the developer should react in one of the following ways:

- recover from the error immediately, or
- abort the procedure and propagate the error back to the caller.

In the latter case, the error should be logged in an entry (“root error”), before transferring the control flow to the procedure’s caller. Figure 8.1 shows an example of this idiom (based on Bruntink et al.’s running example [35]). The variable called `r` on line 2 holds the error status of procedure `f`. On line 7, parameter `a` is checked to see whether it lies in the range of $[0, 10]$. If not, the error status gets updated (line 8), and this error is logged as a root error (line 9). The remaining logic is then skipped until line 19, where cleanup takes place.

When a procedure call returns an error, the calling procedure has a similar choice of either handling the propagated error or passing it on. In the latter case, the calling procedure can possibly add extra context information by replacing the original error value by another, (possibly) more meaningful one. This change of error value has to be logged, and this kind of entry is now called a "linked error" because it links a higher-level error value to a lower-level one. The resulting sequence of linked errors gives rise to an "error link tree", i.e. an exception trace for a particular error. Line 14 of Figure 8.1 demonstrates the logging of a linked error. The original error has occurred during the execution of procedure `g` on line 12. Note that a root error is a special case of a linked error, as it links an error to the `OK` value. If ultimately no exception handler is found, the system will probably go down.

Aside from error variable management, control flow transfer and logging, resource (memory) cleanup plays an important role (line 19 on Figure 8.1). If an error has occurred, any previously allocated memory in the current procedure needs to be de-allocated. This concern is not treated in [35], but we consider it to illustrate how the various concerns fit together.

It is clear from the example that the idiom's logic seriously overcrowds the procedure's main control flow. There is both serious tangling and scattering, as the idiom is applied system-wide. Even for such a simple example as Figure 8.1 it is hard to deduce e.g. what are all possible execution paths which get through the if-checks of lines 18 or 19. This of course hampers any maintenance and/or re(verse)-engineering efforts. From the analysis of Bruntink et al. [35], it turns out that most developer errors regarding the idiom are caused by erroneous guards (i.e. checks) on the error variable `r` and inconsistencies between the logged error value and the one assigned to `r`. Other common errors include forgetting to return an error value, returning the wrong value, logging incorrect things, etc.

We would like to use aspects to relieve the base code developer from the return-code idiom burden, while still giving him or her the power to override default exception handling if desired. In order to do this, we first take a closer look at hidden patterns in Figure 8.1 which suggest a possible solution.

8.2.1.2 Distinguishing between all Crosscutting Concerns

What contributes most to the program complexity is the delicate interplay of main logic, exception handling and resource (memory) cleanup. Moreover, exception handling actually encompasses various subconcerns: error variable management, control flow transfer, logging linked errors, detecting root errors, etc. To make these patterns stand out in the code, we have rewritten Figure 8.1's abbreviated programming style into the equivalent, more canonical Figure 8.2. From this, we can clearly identify the various concerns:

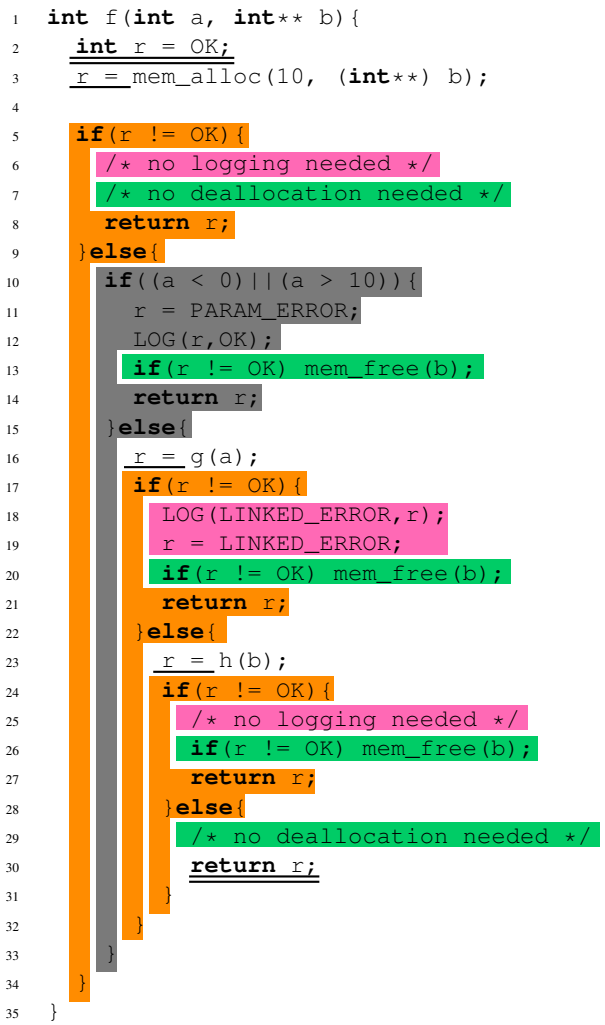


Figure 8.2: Restructured version of the running example of Figure 8.1. Crosscutting concerns are marked with colored F-shapes, rectangles and underlining.

- the declaration, initialisation and returning of a unique error variable (the doubly underlined code on lines 2 and 30),
- the "assign return values to the error variable"-concern (singly underlined lines),
- the control flow transfer which either continues normal execution of a procedure or aborts it by returning the error value (the three orange F-shapes),
- the logging of linked errors (the three pink rectangles),

```
1 /*@range("a",0,10)*/
2 int f(int a, int** b){
3   mem_alloc(10, (int**) b);
4
5   /*@log("LINKED_ERROR")*/
6   g(a);
7   h(b);
8 }
```

Figure 8.3: The main logic from Figure 8.1 to which all aspects and their logic rules and facts presented later in this chapter are applied.

- memory cleanup (the green rectangles),
- the argument range checking concern (the gray F-shape), and
- the main logic (remaining code).

Our intention is to extract all concerns into aspects such that the base code can be reduced to its main concern, as shown on Figure 8.3 (ignoring the comments on lines 1 and 5 for now). We impose the following restrictions, which we revisit later on:

- The procedure signatures remain unaltered to facilitate gradual, stepwise migration from the original idiom-based implementation to an aspect-based one without breaking interfaces. All procedures retain their integer typed return value, which can be used as an error value by some aspect-based exception handling implementations, while others may ignore it.
- In our aspects, we remain as close to the spirit of the return-code idiom as possible, again keeping in mind a gradual, stepwise migration. However, once the original idiom has been extracted into aspects, one can freely experiment with other aspect implementations based on e.g. `setjmp`/`longjmp` or global error variables.

8.2.2 Control Flow Transfer with Delimited Continuation Join Points

From the list of concerns, the one governing the transfer of control flow strikes us as being the most fundamental and hardest one. We therefore discuss it first.

8.2.2.1 The Core Problem of Control Flow Transfer

The following pseudo-code forms the heart of the control flow transfer problem:

```
1 int f(int a, int** b){
2   int r = OK;
3   bool allocated = FALSE;
4   r = mem_alloc(10, (int**) b);
5   allocated = (r == OK);
6   if((a < 0) || (a > 10))
7     ROOT_LOG(PARAM_ERROR,r);
8   LINK_LOG(g(a),LINKED_ERROR,r);
9   NO_LOG(h(b), r);
10  if((r != OK) && allocated)
11    mem_free(b);
12  return r;
13 }
```

Figure 8.4: Macro solution for the return-code idiom in Figure 8.1.

```
1 if(r != OK){
2   return r;
3 }else{
4   /* continue */
5 }
```

After each idiomatic (i.e. returning an error value) procedure call, the execution of the enclosing procedure either halts or continues depending on the call's returned error value (see the orange F-shapes of Figure 8.2).

This is actually a poor man's version of exception throwing in e.g. Java or C++. It is much easier to use and manage than C's `setjmp`/`longjmp` (which is a kind of interprocedural `goto`), but its downside is the heavy tangling with the main control flow and the fact that it is not generally portable. A classic trick to avoid this, is by using macros, as abstraction of language constructs as syntactic sugar is one of the common uses of macros [80]. Figure 8.4 shows the macro-based solution presented by Bruntink et al. [35] for Figure 8.1's example procedure. Macros `ROOT_LOG`, `LINK_LOG` and `NO_LOG` hide assignments to the error value as well as any needed `LOG`-calls. Cleanup code has not been elaborated on [35] as this is considered as a separate concern. The macros already significantly improve the program's clarity, but they still need to be called manually. Hence, people are still not enforced to do exception handling in the right way, and the code is still closely tied to the return-code idiom. One cannot plug in another exception handling strategy.

We have tried to model the core of the idiom (Java-style exception handling) using aspects instead, but found out that this behaviour is not possible to achieve with traditional aspect technology for legacy languages like C. A naive approach would be to put advice around every procedure call in order to only continue the

call if no error has occurred yet in the enclosing procedure. When an error happens, all remaining calls within the current procedure are then short-circuited, i.e. the advice detects the error and skips the call (i.e. does not call `proceed`). Unfortunately, accesses to local variables, arithmetic calculations (more than likely containing bad operands by now), loops, gotos, etc. cannot be advised, hence they are executed and potentially lead to catastrophe. Adding extra join point types for loops, gotos, variable accesses, etc. and short-circuiting them would help, but the continuous checks to find out whether to skip a join point or not incurs a lot of overhead. It looks also more like a hack than a fundamental solution.

Exception handling based on unwinding the stack is actually an example of an "escape continuation"[2]. A "continuation" at any point in the execution of a program can be informally defined as the future execution of that program from that point on. E.g. in:

```
1   printf (getLine ());
```

The initial continuation is the entire program. The continuation after application and evaluation of `getLine` is the application and evaluation of `printf`, i.e. a function which maps a value to the end result of the full program. An escape continuation is a limited form of a continuation, which can only be used to jump back to the context where it has been created (like `longjmp`). Somewhere in between full and escape continuations, lie delimited or partial continuations [90, 123]. These are continuations which do not generate the result of the whole program, but rather an intermediate value. In other words, the scope of the continuation has been restricted and the boundary between the rest of the program and the continuation's scope is called the "delimiter" or "prompt" (e.g. the body of a particular procedure). This concept can e.g. be used [136] to model a context switch within an operating system or to obtain transactional file systems. One can "capture" a continuation to resume it later, possibly more than once. This is mainly used to return to the point at which the continuation has been made [72].

If we would be able to capture the delimited continuation of a procedure call and to choose between resuming or aborting it based on the procedure call's return value, the return-code idiom's control flow transfer would be modeled completely. The approach that comes closest to this is to rewrite procedures in continuation passing style [107] such that the call to the continuation argument can be circumvented by withholding a proceed-call. Migrating to this style of programming is as huge an undertaking as would be abandoning the act of returning an error value. Instead, we have integrated continuations with aspects via Aspicere's *delimited continuation join points*. We have already very briefly introduced these in Section 5.3.2, but the context of the return-code idiom makes them more tangible and easier to explain. We first consider the definition and syntax of delimited contin-

[2] Another interpretation would be the error or exception monad [238].

```
1  void f(void){
2    printf("A");
3    do_something();
4    printf("B");
5  }

6  int main(void){
7    f();
8    printf("C");
9    return 0;
10 }
```

Figure 8.5: Small example which highlights the differences between continuation join points and delimited continuation join points.

uation join points, and then apply them on the implementation of the control flow transfer concern.

8.2.2.2 Definition of Delimited Continuation Join Points in Aspicere

In the context of AOP, we first turn the concept of a continuation into a join point:

> The *continuation* of a join point *p* is a join point representing the future execution after conclusion of *p*.

The reification of the continuation of a join point as another join point is what makes it possible for this continuation to become the target of aspectual advice. A continuation join point is defined in terms of a second join point.

Let us apply this definition to the code in Figure 8.5. Without any advice this program outputs "ABC". Short-circuiting the continuation of the call to `do_something` on line 3 yields "A" only, as the entire remainder of the program is skipped. The return value of the continuation corresponds to the return value of the program (i.e. `int`) and could e.g. hold a meaningful error value.

As it is, this construct is too strong for our purposes. Indeed, it captures the entire future execution of a program, which is too coarse-grained. In the current case study e.g. we are only interested in this future execution up to the end of the current executing procedure, i.e. a delimited continuation. We therefore introduce a reduced version:

> The *delimited continuation* of a join point *p* is a join point representing the future execution after conclusion of *p*, limited to the scope of the procedure in which *p* is active.

Let us again apply this to the code in Figure 8.5. Short-circuiting the *delimited* continuation join point of the call on line 3 now yields "AC", as only the remaining execution within procedure `f` is skipped. Hence, `f` is the prompt of the delimited continuation of the call to `do_something`. There is no return value here, as `f` is a `void`-procedure.

In practice, the concept of a delimited continuation join point allows us to capture the "remainder of the execution of a procedure", which can be used for

an exception handling mechanism. Only `around`-advice on delimited continuation join points of calls or variable accesses seems to be useful. An execution join point's delimited continuation corresponds to nothing (or rather a no-op), while `before`- or `after`-advice on a call's delimited continuation is identical to `after`-advice on the call or its enclosing execution join point respectively.

A delimited continuation join point can be advised in Aspicere as follows. Given a join point `Jp`, the delimited continuation of that join point is:

```
1   delimited_continuation(ContinuationJp, Jp)
```

In the spirit of the pointcut language of Aspicere2, `delimited_continuation` is a logic-based predicate. It takes any join point (`Jp`) and deduces the associated delimited continuation join point (`ContinuationJp`). The value for this is then available for further use in the pointcut and advice. In the case of `around`-advice, calling `proceed` will activate the join point (and hence the continuation) as expected. This (re-)activation can in theory be done as many times as needed, including zero. The latter situation is at the heart of the exception handling mechanism. Aspicere2 does not support multiple activations of a delimited continuation join point, because we have not yet found a need for this and because single invocations can be implemented more efficiently (more on this later).

Instead of having to advise delimited continuation join points, a new keyword can be added to the aspect language for use inside advice: `break(some_value)`. Its semantics are analogous to the `break`-keyword used to jump out of loops, except that it stops the advised join point's enclosing procedure. We have not yet extended Aspicere with such a keyword.

8.2.2.3 Application to the Control Flow Transfer concern

Aspicere's delimited continuation join points give us everything we need to implement the exception handling concern in the example of Figure 8.2. The resulting exception handling aspect is shown in Figure 8.6, while its accompanying logic rules are depicted in Figure 8.7. For now, we focus on the control flow transfer advice of lines 37–45 (code with the shaded background). Advice `error_code_passing` implements the three orange F-shapes of Figure 8.2. Indeed, the advice body on lines 41–44 is nearly identical to the pseudo-code of Section 8.2.2.1. The thing to note here is that the advice superimposes on join points `Jp` (line 37) which represent the delimited continuation of join points `JpCall` (line 40). The latter are so-called "idiomatic calls" (line 38), of which exception handling should not be manually overridden (line 39).

What exactly are idiomatic calls? It is very likely that standard library procedures which accidentally return an integer do not have anything to do with exception handling. Likewise, it is possible that some in-house modules or external libraries deliberately do not take part in the return-code idiom. These two groups

```
1  /*necessary imports*/
2
3  int error_var() on Jp:
4    idiomatic_proc(Jp);
5
6  int error_code_mgmt(int* R) around Jp:
7    idiomatic_proc(Jp)
8    && property(Jp,error_var,R){
9     *R=OK;
10    proceed();
11    return *R;
12 }
13
14 void error_code_resetting(int* R) after Jp:
15   idiomatic_call(Jp,R)
16   && manual(Jp){
17    *R = OK;
18 }
19
20 void error_code_logging(int* R,
21  int ErrorCode) after Jp returning (int* Return):
22   idiomatic_call(Jp,R)
23   && log(Jp,ErrorCode){
24    if(*R != OK){
25      LOG (ErrorCode, *R);
26      *R = ErrorCode;
27      *Return = ErrorCode;
28    }
29 }
30
31 void error_code_update(int* R)
32  after Jp returning (int* Result):
33   idiomatic_call(Jp,R){
34    *R=*Result;
35 }

37 int error_code_passing(int* R) around Jp:
38   idiomatic_call(JpCall,R)
39   && !!manual(JpCall)
40   && delimited_continuation(Jp,JpCall){
41    if(*R!=OK)
42      return *R;
43    else
44      return proceed();
45 }
```

Figure 8.6: The idiom-based exception handling aspect. The shaded area corresponds to Figure 8.2's orange F-shapes (control flow transfer).

```
1  error_code("LINKED_ERROR",0).
2
3  int_invocation(Jp,FName):-
4    invocation(Jp,FName),
5    type(Jp,Type),
6    type_name(Type,"int")
7    .
8
9  idiomatic_proc(Jp):-
10   execution(Jp,_),
11   filename(Jp,"main.c")
12   .
13
14 idiomatic_call(JpCall,R):-
15   int_invocation(JpCall,FName),
16   \+wildcard(".*printf",FName),
17   enclosingMethod(JpCall,JpEncl),
18   idiomatic_proc(JpEncl),
19   property(JpEncl,error_var,R)
20   .
21
22 manual(JpCall):-
23   annotation(JpCall,manual,_)
24   .
25
26 log(JpCall,ErrorCode):-
27   annotation(JpCall,log,[ErrorName]),
28   error_code(ErrorName,ErrorCode)
29   .
```

Figure 8.7: Accompanying Prolog meta data of the aspect in Figure 8.6.

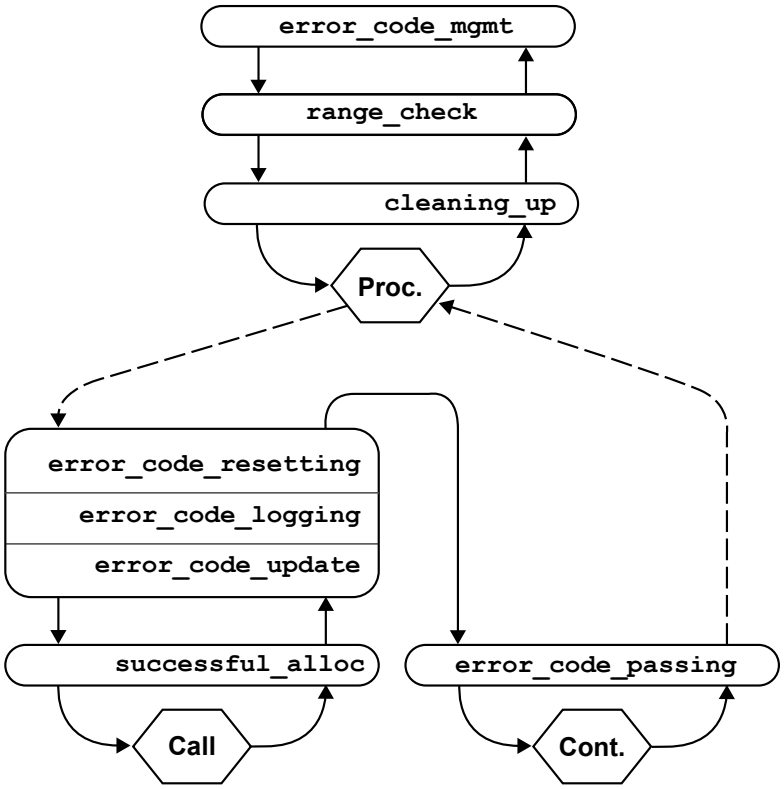


Figure 8.8: Schematic order of execution of all aspects woven into an idiomatic procedure like the one of Figure 8.1. Full arrows denote execution flow, while dashed ones indicate that the sequence in between them can show up zero or more times. Rounded rectangles represent advice, while hexagons indicate execution, call and delimited continuation join points. The advice name is centered for `around`*-advice, while it is right-aligned for* `after`*-advice.*

of procedures should be excluded, and this happens via the `idiomatic_call` and `idiomatic_proc` predicates shown in Figure 8.7. The former states (lines 14–20) that idiomatic calls are nothing more than invocations (calls) returning an integer and located inside idiomatic procedures (lines 17–18). We will discuss line 19 in Section 8.2.3. We can avoid calls to standard library procedures which accidentally return integers by excluding them through a wildcard pattern like the one on line 16. Idiomatic procedures specify the in-house modules which take part in the return-code idiom (lines 9–12). The `idiomatic_proc`-predicate limits the aspects' scope to the relevant modules (e.g. "main.c" on line 11). This is actually build configuration data which should be integrated into the logic fact base, similar to the build dependency graph information in Section 7.3.2.

To summarise, whenever an error value is returned from an idiom-participating procedure call, the remaining execution of the enclosing procedure is skipped. Otherwise, normal execution is resumed. This is clearly illustrated in Figure 8.8, which gives an overview of the role of each advice described in this chapter and the join points it applies at. The `error_code_passing`-advice can decide between resuming the delimited continuation (`Cont.`) or aborting the current procedure (`Proc.`). The other aspects are explained in the next two sections.

8.2.3 Logging and Overriding with Join Point Properties and Annotations

The previous section's control flow transfer advice is only part of the story. In this section we consider the logging concern, as well as the possibility of overriding the default aspect behaviour. Afterwards, the next section discusses resource cleanup.

8.2.3.1 Challenges for Logging and Overriding

As discussed in Section 8.1, developers should be able to override flow transfer in case the default idiom does not suffice. Sometimes, one needs to perform extra resource cleanup or to actually handle an expected error right after the erroneous procedure call. We also need a way to automatically log linked errors, and here again developers should be able to decide whether linking is necessary and if so what the new error value should be. Both cases are concerned with giving the developers the necessary power to control advice execution.

A second issue relates to advice interaction. How does each advice know when an error has occurred and what its value is? Using a global error variable (or a stack thereof) will lead to race conditions in multi-threaded architectures. This problem even gets worse for the resource cleanup aspect. Also, it is at odds with the actual semantics of an error status, as these are tied to procedures.

Overriding the default behavior is achieved using the concept of annotations (Section 5.3.4), whereas we use join point properties (Section 5.3.2) to deal with the communication between advice on shared join points. We prefer manipulation of advice through annotations to custom advice written by a base code developer for various reasons:

- During examination of advice interaction in the whole system, one can focus on the few known system-wide aspects which are guided by developer annotations instead of having to understand (possibly) hundreds of small advices added by individual developers every day. This makes proving correctness of the aspect-enabled system more reasonable.

- Annotations can be checked by a tool to detect errors in their usage.

- Existing development tools (IDEs) do not need to change. Care must be taken that the weaver for system-wide aspects can be integrated nicely into the compiler chain (cf. the Kava and Quake 3 cases).
- Aberrant exception handling behaviour remains local to the place at which it occurs.

We now apply these concepts to the aspects for logging and overriding of the basic exception handling. We use Figure 8.8 to chronologically discuss the various advices.

8.2.3.2 Implementation of Logging and Overriding

The exception handling aspect of Figure 8.6 shows the application of join point properties and annotations for the developer overriding and logging advices (lines 1–35). Lines 3–4 contain the declaration of a join point property called `error_var` which represents the error variable[3] associated with each idiomatic procedure `Jp`. We could just as easily have named this property `r` (in line with the original code in Figure 8.1), but this is less self-documenting. The variable should be initialised to the `OK` status every time a procedure starts executing and should be used as the procedure's return value at the end. This is handled by advice `error_code_mgmt` (lines 6–12). The advice accesses the right property instance at the current execution join point `Jp` (line 7) by binding `R` to a pointer to the instance of the `error_var` property attached to `Jp`. A join point property is really a part of a join point's context, and hence the `error_var` property is conceptually equivalent to a local variable of the procedure it is attached to.

As discussed in Section 5.4.2, the lexical order of advice determines the precedence at shared join points. This is illustrated on Figure 8.8. The lexical order of e.g. the three advices on `idiomatic_call` join points in Figure 8.6, directly corresponds to the execution order on the shared call join point (`Call`). However, because these three advice are `after`-advice, the first one in the aspect is the last to be executed, as the arrows indicate. This means that advice `error_code_update` (lines 31–35) is actually the first (of the exception handling aspect) to execute after an idiomatic call. Its task is to record the call's error status into the enclosing procedure's `error_var` property. As such, it provides the shared join point property with the right error value to let the other advices do their job.

After `error_code_update`, annotations kick in. First, logging of linked errors is only meaningful if a developer provides a new error value. Hence, a log annotation can be used to specify the new error value. The absence of a log annotation can be interpreted as if logging is not necessary. Line 5 in Figure 8.3 shows

[3] In Aspicere2, properties and return values are accessible from within advice by pointers instead of by their real type to enable modification of their value.

```
1  int f(void){
2    int tmp=OK;
3    ...
4    /*@manual()*/
5    tmp=g();
6
7    if(tmp==EASY_TO_FIX_ERROR){
8      /* recover manually */
9    }else if(tmp==INITIAL_CLEANUP_ERROR){
10     ...
11     rethrow(tmp);
12   }
13
14   ...
15 }
16
17 int rethrow(int a){
18   return a;
19 }
```

Figure 8.9: Small example which illustrates overriding of exception handling by developers.

such an annotation, which consists of the name "log" and one attribute with the new error value ("LINKED_ERROR"). Advice `error_code_logging` (lines 20–29 on Figure 8.6) tells us that when an annotated (line 23), idiomatic procedure call (line 22) returns, there will be a linked error log (line 25) if the `error_var` property signals an error (line 24). Both the error variable and the call's return value need to be updated to the newly provided error value (lines 26–27). Annotations ensure that logging only happens when needed.

To override the `error_code_passing`-advice of lines 37–45 for manual recovery (lines 14–18), it suffices to reset the error variable (line 17) when a developer uses a `@manual`-annotation. Indeed, as this restores the error variable to `OK` again, the `error_code_passing`-advice will not notice anything and just proceed (see Figure 8.8). The developer should then add a check in the base code to find out whether or not the procedure call returns an expected error return value, and recover from the error the way he or she wants. Figure 8.9 illustrates this mechanism on lines 7–8. Hence, control flow transfer happens by default, while logging and manual overriding should be explicitly asked for.

What if the developer just wants to do some initial recovery manually, followed by the default exception handling behaviour ("rethrowing" the exception)? This is actually easy to deal with (lines 9–12 of Figure 8.9): we just need to use the `@manual`-annotation and to call the identity procedure (aptly called `rethrow`,

```
1 /*necessary imports*/
2
3 int range_check(int* R,int Arg,
4  int LowerBound,int UpperBound) around Jp:
5   idiomatic_proc(Jp)
6   && property(Jp,error_var,R)
7   && range(Jp,Arg,LowerBound,UpperBound){
8    if((Arg < LowerBound)||(Arg > UpperBound)){
9      *R=PARAM_ERROR;
10     LOG(*R,OK);
11     return *R;
12   }else{
13     return proceed();
14   }
15 }
```

Figure 8.10: Parameter range checking aspect.

```
1 range(Jp,Arg,LowBound,UpBound):-
2   annotation(Jp,range,[ArgName,LowBound,UpBound]),
3   nth_arg(Jp,Arg,ArgName)
4   .
```

Figure 8.11: Accompanying Prolog meta data of the aspect in Figure 8.10.

see lines 17–19) after local recovery has been done. This again triggers the aspectised exception handling on the original error value, or a higher-level one. The advantage of a `rethrow` statement to just using a `return` statement, is that the latter ties the base code again to the specific return-code idiom. A `rethrow` is a generic exception handling concept which is equally applicable when using return values or e.g. `setjmp`/`longjmp`.

Although not necessarily a part of the return-code idiom, we can extract the argument range checking concern as well, which yields the aspect of Figure 8.10 and its accompanying Prolog file (Figure 8.11). It only involves looking for `@range`-annotations on procedures (see line 1 of Figure 8.3). These annotations specify the relevant argument name and the two bounds. The advice body is similar to Figure 8.2's gray F-shape. Figure 8.11's predicate allows easy access to the annotation info (line 2) and the checked argument `Arg` (line 3).

```
1  /*necessary imports*/
2
3  int memory_op_success() on Jp:
4    memory_allocation(Jp,_,_);
5
6  Type memory_op_alloc(TYPE Type) on Jp:
7    memory_allocation(Jp,_,Type);
8
9  void successful_alloc(int* SuccessVar,
10  TYPE Type,Type AllocVar,Type Actual)
11  after Jp returning (int* Code):
12   memory_allocation(Jp,Actual,Type)
13   && property(Jp,memory_op_success,SuccessVar)
14   && property(Jp,memory_op_alloc,AllocVar){
15    if(*Code==OK){
16      *SuccessVar=1;
17      *AllocVar=*Actual;
18    }
19 }
20
21 void cleaning_up(int* R,int* SuccessVar,
22  TYPE Type,Type AllocVar) after Jp:
23   idiomatic_proc(Jp)
24   && enclosingMethod(JpCall,Jp)
25   && memory_allocation(JpCall,_,Type)
26   && property(Jp,error_var,R)
27   && property(JpCall,memory_op_success,SuccessVar)
28   && property(JpCall,memory_op_alloc,AllocVar){
29    if((*R!=OK)&&(*SuccessVar==1)){
30      mem_free(AllocVar);
31    }
32 }
```

Figure 8.12: Memory handling aspect.

```
1 memory_operation("mem_alloc").
2 memory_operation("mem_calloc").
3
4 memory_allocation(Jp,Actual,Type):-
5   invocation(Jp,FName),
6   memory_operation(FName),
7   nth_arg_type(Jp,1,Type),
8   nth_actual(Jp,1,Actual)
9   .
```

Figure 8.13: Accompanying Prolog meta data of the aspect in Figure 8.12.

8.2.4 Memory Cleanup with Join Point Properties and Type Parameters

Having tackled the error propagation and logging concerns, we now focus on the cleanup of resources in case of errors, and more in particular on the cleanup of any dynamically allocated memory. The exact problem involves detecting which variables refer to allocated memory (need to be cleaned up) and which ones are just dangling pointers which have not been initialised yet (no cleanup). It suffices to store the memory allocation procedure's return value and to use it when an error is going to be thrown to identify dynamically allocated memory in the current procedure. Of course, there should also be a way to access the allocated memory. For both these purposes, join point properties on memory allocation procedure calls are a good fit.

Figure 8.12 shows the resulting memory cleanup aspect. The two mentioned join point properties are declared on lines 3–4 and 6–7. The first one (initialised by default to `OK`) signals whether the corresponding memory allocation has been a success, in which case the second one holds a reference to the allocated memory area. The properties are associated with `memory_allocation`s, i.e. calls to `memory_operation`s (lines 4–9 on Figure 8.13) of which some examples are given on lines 1 and 2. For each call, both the second[4] actual `Actual` (line 8) as well as its type `Type` (line 7) are captured. The latter is needed to deal with allocation of both simple and compound types. As C lacks polymorphism or something similar, and `void`-pointers are not type-safe (although they would work in this case), we apply Aspicere's type parameters (Section 5.3.1) here.

Advice `successful_alloc` (lines 9–19) catches the return error value of allocation procedure calls (line 11) and stores a reference to the allocated memory area (line 17), but only if no error occurred. Advice `cleaning_up` (lines 21–32) frees allocated memory after the continuation has been skipped (the `error_var`-property differs from `OK`) for each successfully allocated variable (line 29). This advice makes heavy use of Aspicere's logic pointcut language. Backtracking ensures that for every idiomatic procedure `Jp` (line 23), all `memory_allocation`-calls within it (`JpCall` on lines 24–25) will trigger a match of the advice on `Jp`, each time with the right instances of the `memory_op_success` and `memory_op_alloc` properties. The logic facts enable easy navigation through the program structure (line 24).

Notice on Figure 8.8 that memory allocation and cleanup do not take place when a range violation occurs, as the `range_check`-advice does not `proceed` in that case. This is semantically correct, as careful investigation of Figure 8.2 learns. On line 13, the `mem_free`-call will always be run to cleanup the allocation of line 3 (which has succeeded when a range violation happens). As this is

[4] Indexing in Prolog starts from zero.

a redundant transaction (allocate, detect range error and de-allocate), our aspect just ignores memory allocation and cleanup in this case. An alternative which adheres better to the idiom would be to advise the last `memory_allocation`'s delimited continuation join point, provided that all such allocations occur at the beginning of procedures.

What happens if the call to `mem_free` on line 30 of Figure 8.12 goes wrong? Aspicere2 does *not* allow advice on advice, which would let this error go unnoticed. Fortunately, replacing calls to `mem_free` by invocations of a wrapper procedure around it solves this if the wrapper is part of the base code. Indeed, our aspects will advise either the wrapper or the `mem_free`-call it contains to provide default exception handling. If some more specialised recovery actions should be needed in this case, one could put them into an additional aspect or just add it directly to the wrapper.

8.2.5 Validation #1: Aspicere Meets Goal L1

The novel combination of delimited continuation join points, annotations, join point properties and template parameters is able to extract exception handling and associated memory cleanup into reusable aspects. The resulting base code is much cleaner and requires explicit annotations before the default exception handling algorithm can be overridden. This non-standard behaviour can be automatically detected with a simple tool, if desired. The next subsections discuss the scalability of our aspects, run-time overhead and adoption of our delimited continuation join points.

8.2.5.1 Scalability of the Aspects

Looking back at all aspects and Prolog files we have presented, one could argue that the original code of Figure 8.1 is much shorter. At first sight, it is. However, we should keep in mind that our aspects have to be written only once and are superimposed throughout the whole software system. They form in fact an initial investment which pays itself back once a whole source code module or component is refactored. The only thing developers need to do is to annotate their code or (in some cases) override default exception handling behaviour. Bruntink et al. [33] have measured that in a representative module of 20 kLOC exception handling accounted for 9% of the module size (1716 LOC), whereas our aspects (together with the Prolog files) account for 122 LOC. For each logged linked error, there will be a `@log`-annotation. If developers want to recover manually, there will be another extra line of code, i.e. the `@manual`-annotation. More precise numbers can only be obtained when applying our aspects in practice, but a rough estimation suggests that our approach allows for a substantial code reduction. This enhances base code readability and facilitates software evolution.

8.2.5.2 Run-time Overhead

At run-time, the aspects adds some extra complexity which were not there in the tangled implementation:

- advice is transformed into procedures to which various calls are issued;
- the cleanup aspect adds extra local variables to idiomatic procedures and contains extra calls to the `succesful_alloc`-advice.

On the other hand:

- join point properties can be mapped onto local variables;
- advice `error_code_passing` can be inlined efficiently, resulting in the same code size as the original base code;
- Figure 8.10's aspect actually optimises the current implementation as it avoids unnecessary memory (de-)allocation;
- the bitcode optimisation passes can eliminate lots of redundant code.

The most natural implementation strategy for an `around`-advice on a delimited continuation join point D of some join point J is to physically extract the remaining procedure instructions following J into a new procedure and to replace `proceed`-calls by invocations of this new procedure. Unfortunately, this can turn out difficult as soon as multiple `if`/`else`-constructs, loops, variable argument lists, etc. show up. Also, efficiency is seriously influenced by the frequent procedure calls.

Aspicere2 takes a simpler, but slightly restricted approach: the advice body gets inlined into the base code. If instruction N follows the join point shadow of J, then the advice body is pasted right before N and any advice calls to `proceed` are replaced by jumps to N. The advice's return statements become return instructions of the enclosing procedure of J's shadow (and N). This is very fast (both for build- and run-time), but makes the size of the woven code larger. Fortunately, optimisation passes can reduce the code size. A second drawback is that any advice statement which physically follows a `proceed`-statement (by now replaced by a jump instruction) is never executed, as there is no jump back from the end of the procedure to the advice instruction following the invoked `proceed`-call. Hence, the procedure's return instruction exits the procedure without ever running any advice code following `proceed`-calls. As a consequence, multiple `proceed`s after each other in advice do not work either, hence a delimited continuation join point can only be resumed once with Aspicere2. However, for most purposes this restriction can be worked around by splitting the intended advice into multiple advices chained one after the other. In general, inlining makes the woven code almost identical to the canonical, tangled implementation of Figure 8.2.

8.2.5.3 Adoption of Delimited Continuation Join Points

In the meantime, other researchers have adopted our delimited continuation join points for exception handling. Delimited continuation join points have been incorporated into recent releases of ACC (Section 5.2.1.7) via the `preturn` construct, which is identical to the special `break`-keyword mentioned in Section 8.2.2.2. It exits the advice in which it occurs and the enclosing execution join point (procedure) of the advised join point. ACC has generalised the statement such that it can also occur in e.g. `before`-advice instead of only in `around`-advice. Apart from `preturn`, ACC also has a `try–catch–throw` mechanism at the advice level. A `throw`-call within an advice can generate an exception (integer value) that can be caught by an advice which advises another join point higher up the stack. These exception handling constructs are only available within aspects, not in the base code. The underlying exception implementation uses `setjmp`/`longjmp`.

This completes the description of this advanced re-engineering application with aspects. The next section considers the implications on the build system.

8.3 Impact on the Build System

This section discusses the impact of the introduction of AOP technology on the build system, similar to the previous two chapters. We focus on:

1. integration of Aspicere2 with the build process
2. migration to the re-engineered system
3. build time increase

8.3.1 Integration of Aspicere2 with the Build Process

We have not yet evaluated our approach on the actual ASML system, only on the running example of Bruntink et al. [35]. Hence, we do not have concrete data on integration into the build process, except that integration of the configuration system with the logic fact base is needed to communicate the scope of the return-code idiom to the aspects (`idiomatic_proc` in Figure 8.7), i.e. which modules do not apply the exception handling idiom. This is similar to our remarks in the Quake 3 case, but now it considers configuration data instead of build data (RC4).

This configuration data heavily relies on a succinct definition of the notion of “whole program” (RC1) and on knowledge of the interactions between components (RC2). Otherwise, part of the frames on the program call stack could correspond to procedures which have been advised to support exception handling, whereas others do not know anything about exceptions. As in the worst case these two categories of stack frames could be interleaved, exception handling would

become unreliable. Hence, clear boundaries of the "whole program" should be defined (RC1), and interactions between build components (RC2) should be examined to ensure that exception handling is semantically consistent across the whole system.

More detailed analysis of build problems is only possible on a concrete case. Unfortunately, it is impossible to test our full aspect solution on other systems, because these do not use the exact same combination of idioms. Nevertheless, the aspects are robust enough to be easily integrated into other systems. We are planning to apply our aspects on Quake 3 (OpenGL calls) and on a typical application which uses OpenSSL[5]. There are some known problems with error handling with the latter[6], which we aim to address. These cases will give us a better idea on build process integration problems.

8.3.2 Migration to the Re-engineered System

From the discussion in this chapter it becomes clear that migration from a tangled implementation of exception handling to an aspect-based approach is not straightforward. Contrary to logging or tracing, exception handling is a functional concern which is deeply intertwined with the base code's control flow. The migration to an aspect-based exception handling implementation is a huge evolution step, even between two different paradigms (procedural and AOP). It is not hard to foresee that this will have serious consequences on the build system too, i.e. the co-evolution phenomenon of source code and the build system becomes very apparent.

Bruntink et al. [32] note on this topic that migration from a tangled implementation of an idiom to an aspect-based one inevitably will be an incremental process [31]. Functional equivalence between the original system and the new one has to be verified at every step, especially because of the variability within the tangled idiom implementation. As the correct usage of idioms cannot be enforced, there are various small discrepancies. Bruntink et al. identify a trade-off between on the one hand reducing the risk of introducing errors by maintaining the available variability, and on the other hand reducing the idiom variability with the risk of introducing errors in the system that way. Variability makes verification hard to do.

Keeping this incremental nature in mind, this means that the system has to migrate step by step to a full AOP implementation. Hence, the build system will gradually have to evolve from a purely traditional one, i.e. based on the classic notion of modules, to an aspect-aware system. In between, the two technologies need to co-exist. Configuration of which base modules have to be advised and which ones not is crucial (RC4). The `idiomatic_proc` predicate is the easiest way to communicate this to the weaver.

[5] `http://www.openssl.org/`
[6] `http://blogs.sun.com/alvaro/entry/openssl_error_handling_problem/`

However, the configuration of these modules also has direct consequences on the scope of the weaver. As seen in the Kava and Quake 3 cases, knowledge of the interplay between object files, libraries and executables is a prerequisite for being able to assess when (source-to-source, link-time, etc.) and where (which directory, etc.) a weaver should be invoked. Language-level features like shared state rely on this. If the source code, and hence the build system, is constantly in flux, managing the changing architecture becomes a full-time concern (RC1 and RC2), especially if we take into account the possible inconsistency problem mentioned in the previous subsection. Without proper tool support, this becomes an impossible task. A powerful testing infrastructure is required to assess the semantic consequences of this. As the build system is in charge of unit and integration testing and the presence of aspects might require a different testing infrastructure than the base code, a lot of communication and co-ordination is needed to keep both subsystems operational.

During each migration step the migrated base code components need a thorough conversion. First, the actual main concern should be recovered from the tangled representation it is held captive in (cf. Figure 8.2). Second, the relevant error values should be looked up. Third, all error handling code should vanish and annotations should be inserted to guide the aspects. This is indeed a major effort, requiring extensive test suites to validate the migration results and the remaining old subsystems. The automated approach for concern verification presented in [33] could be applied here. Automatic extraction of pointcut and advice, however, is unnecessary. Indeed, our major pointcuts are based on "semantic" concepts like returning an integer, delimited continuation join points, annotations and join point properties.

As return values were used to indicate an error status in the original implementation, they are now pretty much useless in the base code. Instead of changing them into `void`, keeping them adds extra possibilities for aspect implementations. The two restrictions of Section 8.2.1.2, i.e. keeping the procedure signatures fixed and implementing the original idiom in the aspects, were aimed at limiting the migration effort and to leverage the base code developers' understanding of the return-code idiom during the migration. Once this is completed, and behaviour preservation has been verified, one can switch to other aspects and teach the new exception concepts to the base code developers. The resulting code of Figure 8.3 could in fact be the starting point for any exception handling strategy, hence the cleanup effort is not a waste of time. Changing exception handling strategies is now just a matter of writing and using another aspect. The build system should be capable of flexibly switching between different strategies, perhaps even per subcomponent of the system. If certain exception handling algorithms cannot coexist together (in different components of the system e.g.), the aspect configuration should be verified to not break these constraints (RC4).

8.3.3 Build Time Increase and Incremental Weaving

We have not applied our aspects in practice yet, except for the running example of Bruntink et al. [35] where they behave as explained in this chapter. Because delimited continuation join points are implemented by Aspicere2, the same incremental build problems as in the Quake 3 case will occur.

Second, during migration from the original system to the AOP incarnation, the configuration of the base code modules into which the exception handling aspects should be woven actually corresponds to a high-level incremental weaving approach. More in particular, the choice to weave aspects into some parts of the systems and to migrate other parts later on, corresponds to pruning the dependency graph, in this case not for speeding up but to migrate stepwise. This incremental weaving conflicts with the desire to keep the build dependency graph intact, i.e. problems related to RC3 ensue. Knowledge of the interactions and dependencies between components are required to resolve these problems, but this knowledge is buried in the source code or can only be detected by examining the semantics of base code modules. Tool support, for build systems and source code, is indispensable to deal with these issues.

8.4 Validation #2: Roots of Co-evolution Experimentally Confirmed

This section discusses evidence in this chapter's case study of build problems according to the four roots of co-evolution. All four roots are applicable. RC1 and RC2 surface because developers need to have a clear definition of "whole program", and of the dependencies and interactions between build components to keep exception handling semantically consistent across the system. This is especially a challenge in the face of migration from the old system to the aspect implementation, as the collection of components which do or do not rely on the aspects constantly changes. Keeping track of this is a challenge. Tool support, both for managing the build process and the source code, is a prerequisite.

At each moment during the migration, the partition of the build dependency graph into regions in which aspects are applied, others which still use the original idiom implementation and even components which do not use any idiom for exception handling at all, has important consequences on the completeness of the dependency graph. Dependencies between base components are in some regions replaced by dependencies between aspects and base code, but elsewhere not. Hence, from the perspective of the weaver, the dependency graph misses various dependencies, as if the system is incrementally woven (RC3). Indeed, the partitioning of the build dependency graph at all times has to ensure that no semantic inconsistencies arise in the source code. This is not straightforward to manage.

Finally, the build configuration has serious implications on the aspects too, as the `idiomatic_proc`-predicate in Figure 8.7 relies on this information to steer the advices. Hence, this configuration information should be passed from the build system to the logic fact base. A second implication of RC4 is that the configuration of base code and aspect modules should respect the semantic interactions and dependencies between them, to ensure consistency in the exception handling semantics (similar to above).

8.5 Validation #3: MAKAO Achieves Goal T2

Even though we have not applied the aspects in practice, the explanation has shown parallels with the earlier two case studies, i.e. knowledge of the build architecture is indispensable to manage the scope ("whole program") of aspects, detect structural dependencies between build components and to give at all times an up-to-date overview of the migration of the system and of the testing framework. MAKAO's visualisation and querying are indispensable in this regard. Given the scale of the system, filtering is a powerful feature as well. We estimate that application of MAKAO's re-engineering and verification can be important too, although this depends for a great deal on the consistency of the build system. As observed in the Quake 3 case, a disciplined build system can handle some drastic changes better.

8.6 Validation #4: Aspicere Meets Goal L2

Just as in the Quake 3 case, we have exploited the integration of the build system into the logic fact base. This time, configuration information is required instead of build layer data. The actual mechanism to communicate between the configuration layer and the aspect weaver depends on the technology used. It may range from asserting file names from a configuration specification to extracting this information from makefiles or custom build system infrastructure. Any mechanism suffices, as long as the logic fact base is synchronised with the build configuration.

8.7 Conclusion

We have shown how a combination of carefully crafted aspects relieves the developers from the return-code idiom administration, unless they explicitly choose to do manual recovery themselves. At the heart of our approach lies the concept of delimited continuation join points, giving aspect developers the power to skip the remainder of a procedure at any (join) point during its execution. Join point properties are used to decouple advices from each other, while annotations allow developers to override the default exception handling scheme. The latter, together

with type parameters and the fact that all procedures keep returning an integer error value, results in fairly robust pointcuts and advice. We have argued that the aspects considerably improve code readability, understandability and evolvability, by extracting all exception control flow and (in general) reducing code size. Compared to the original code, the woven application should have similar run-time efficiency. These claims still need to be backed by practical application to a real-world case, but conceptually goal L1 has been satisfied.

On the build system level, we have explained that because the migration from a tangled exception handling strategy to an aspect-based one is extremely invasive, the demands for the build system are equally high. It needs to evolve from a traditional build architecture to an aspect-aware one in a gradual fashion. As a consequence, both philosophies need to co-exist and co-operate at each migration step, while they are constantly changed. Configuration of aspects, controlling the scope of weavers and keeping two distinct testing frameworks operational are only some of the challenges ahead. This has consequences which can be understood based on all four roots of co-evolution, and contrasts with the Kava and Quake 3 cases, where the weaver had to be integrated at once into a "stable" build system. MAKAO (goal T2) and Aspicere (L2) are able to support the migration to the re-engineered system. The following chapter discusses a second invasive change to an existing system.

Beware of little expenses; a small leak will sink a great ship.

Benjamin Franklin

9

Case Study 4: Temporal Pointcuts to Support the Re-engineering of the CSOM VM

THE fourth case study is concerned with the re-engineering of the implementation of the CSOM virtual machine (VM) based on a domain-specific architectural description language (ADL) [168] named VMADL [112]. Aspicere is used as the underlying implementation of the VMADL compiler. To support the expressive VMADL descriptions, we have added history-based (also called "temporal" or "stateful") pointcuts to Aspicere. The resulting temporal pointcut language extension is named "cHALO", from the HALO [115] aspect weaver implementation it is based on. The primary incentive behind cHALO is to make memory and runtime overhead associated with temporal pointcut languages in C more explicit to the aspect developer, and to enable him or her to extend cHALO with new kinds of pointcuts which exploit the developer's knowledge about existing join points to reduce overhead. This chapter[1] describes in detail the rationale behind cHALO and its implications on the build system.

First (Section 9.1), the context of the case study is described, followed by the discussion of the design of cHALO (Section 9.2) and its application in the CSOM VM implementation. Section 9.3 considers the implications on the build system of the introduction of AOP technology (VMADL and Aspicere) in CSOM. Section 9.4 validates the existence of build problems suggested by the four roots

[1] This chapter is based on [6].

of co-evolution, and Section 9.5 evaluates MAKAO's ability (goal T2) to support the understanding and management of co-evolution of source code and the build system. The conclusions of this chapter are presented in Section 9.6.

9.1 Rationale behind the Case Study

The context of this case study are virtual machine (VM) implementations [208] like Java HotSpot or Parrot (see Chapter 10). A VM is a kind of infrastructure software, like operating systems, which provides services to user-level programs running on top of them. As such, application programmers do not need to worry about the underlying hardware, or the internal complexity of exception handling strategies or garbage collection algorithms. In other words, non-functional requirements and concerns of applications are fulfilled or implemented in the VM. Because the end-user is not interested in the VM itself, its implementation should be as inconspicuous as possible, i.e. fast and with a small memory footprint.

As VMs by definition make abstraction of the underlying hardware, they are usually heavily ported to multiple platforms like desktop computers, cell phones or PDAs. Because the capabilities of the underlying operating systems on these platforms vary quite a lot, the optimal garbage collection algorithm or multithreading implementation for a VM varies as well. Unfortunately, the internal structure of VM implementations erodes very easily, even if the VM services originally were modularised well [48, 112]. The need for blazing speed and reduction of memory usage forces VM implementers to bypass the original module interfaces. More fundamentally, the fine-grained interaction between e.g. object layout and garbage collection is impossible to modularise with traditional programming language technology. To solve these problems, Haupt et al. [112] have proposed an AOP-based approach to specify the architecture of a VM, i.e. the Virtual Machine Architectural Description Language (VMADL).

VMADL is a domain-specific ADL geared towards declarative specifications of VM implementations. It considers a VM as a collection of "service modules" ("service" in short), i.e. subsystems which provide a specific functionality to user-level programs. Like normal modules, service modules have an API which declares the module functionality exposed to other services as methods. On top of this, services also have a Crosscut Programming Interface (XPI), i.e. a collection of pointcuts to advertise the join points which are guaranteed to exist within the service implementation and hence are safe to advise. XPIs correspond to a possible design rule to decouple aspects from base code implementation details (see Section 2.4.2). They form a contract between the service implementer, which promises not to invalidate the exposed join points, and the client, which does not need to worry about fragile pointcuts if he or she only advises an exposed pointcut or invocations of methods of the API. Services can interact in much more fine-

grained ways with each other by advising specific service-exposed join points. This composition is declaratively specified by a VMADL program, which selects the right service modules to combine and determines how they should be connected via their public APIs and XPIs.

Because of the intricate interactions between services, dynamic switching of exposed XPIs based on the state of the VM and the fact that a service interface can have different implementations, the pointcut language within VMADL should be very expressive. It should be able to express robust pointcuts which can either relate join points exposed by multiple services or join points within the services' internal implementation. A number of required pointcut constructs have been identified [112], e.g. to bind any desired join point context. One of the most important ones are stateful aspects [67], i.e. pointcuts which incorporate the notion of time in some way, like Arachne's `sequence`-pointcuts do (Section 5.2.2.3). In VMADL, temporal pointcuts enable developers to elegantly specify the complex interactions between services.

In this chapter, we focus on a case study performed on a small VM, CSOM, with a first prototype of a VMADL compiler. CSOM is a VM for a Smalltalk dialect and is used for teaching purposes at the Hasso-Plattner-Institut in Potsdam (Germany). The architecture of CSOM is deliberately simple (37 source and 39 header files, accounting for 4753 SLOC). The standard implementation features a Smalltalk parser and compiler, a corresponding object model for representing Smalltalk entities, a simple bytecode interpreter, and a standard library of more or less two dozen classes. By default, garbage collection and multithreading are not provided. For our case study, CSOM has been extended with four extra services, two for garbage collection (mark-and-sweep and reference counting [124]) and two for multithreading (green and native threading [149]). For each of these services, we have first modified the original CSOM and afterwards we have extracted these modifications as VMADL aspects. We have done this for each service in isolation, i.e. we have not yet combined multiple services at the same time. Advanced topics like resolving interactions or modularising the full VM implementation are considered as future work.

The VMADL compiler transforms high-level VMADL-constructs to Aspicere aspects. This transformation is straightforward, as VMADL pointcuts are converted to Aspicere Prolog predicates and advice is mapped to Aspicere advice. As a consequence, this means that Aspicere needs support for temporal pointcuts. Various optimisations and program history retention strategies have been proposed by researchers to make such a history-based pointcut language feasible, but, except for Arachne, they are mainly targeted at Java-based aspect languages. Because C programmers want explicit control over pointcut behaviour and memory footprint, we have based Aspicere's temporal pointcut language on the HALO [115] pointcut language for Lisp. This one is built around a limited

set of fine-grained temporal pointcut primitives with well-known memory and behaviour. This transparency between primitives and weaver implementation is accomplished via a Rete-based [94] run-time engine. Because of its roots, we have named the resulting extension of Aspicere "cHALO".

VMADL is discussed in detail elsewhere [48, 112]. For this chapter, it suffices to know that each VMADL file is transformed into one Aspicere aspect and Prolog module. Instead, we elaborate on the design of cHALO, as it has important consequences on the build system. After the explanation of cHALO in the next section, we discuss the impact of the CSOM case study on co-evolution of source code and the build system.

9.2 Application of AOP

The rationale behind cHALO is discussed in the next subsections. Section 9.2.1 sketches the general problems of history-based aspect languages for C and how cHALO intends to deal with these. To illustrate the main design choices, this chapter uses a running example which we first implement in two existing history-based pointcut languages (Section 9.2.2), i.e. Arachne [68] and tracematches [12]. Then (Section 9.2.3), the required HALO features for the design of cHALO are presented. Section 9.2.4 rephrases the basic temporal operators of Arachne and tracematches in terms of HALO's primitive predicates to better understand the semantics and memory requirements of Arachne and tracematches. HALO's key ideas are transformed to Aspicere in the form of cHALO (Section 9.2.5), and we briefly discuss the accompanying Aspicere2 implementation. We illustrate the flexibility of cHALO with an example from the CSOM case study. Shortcomings of the resulting system, which represent future work, are discussed in Section 9.2.6. Finally, Section 9.2.7 validates whether or not Aspicere satisfies goal L1.

9.2.1 Problems of History-based Pointcut Languages for C

History-based pointcuts have originally been proposed by Douence et al. [70]. More specifically, temporal pointcuts express patterns over the program execution history. This program trace corresponds to a sequence of AspectJ-like join points. In general, the various pointcut designators which together form a complete temporal pointcut[2] are composed via temporal operators or some other means (e.g. regular expressions) to express certain patterns of events. An alternative interpretation is that the particular join points a pointcut is interested in may change depending on earlier events in the base code ("stateful aspects" [67]). This lends itself naturally for modeling protocol-like concerns [1, 68, 115] or complex behavioural patterns [12].

[2] We will call them "sub-pointcuts" from now on.

Despite its promises, adoption of history-based pointcuts has been slow to catch up [16], primarily because of efficiency issues. There are roughly two areas of concern: memory space and execution time. Intuitively, the idea of matching join points based on program trace elements requires that these traces and available context have to be stored somewhere as long as they can contribute to a pointcut match. Additionally, "partial matches" have to be maintained to keep track of sequences of join points which are matched by the first part of a temporal pointcut, but have not yet given rise to a complete match. Unfortunately, some of this state may need to be retained indefinitely if no future join point completes a partial match. In the meantime, context values associated with past join points may have gone out of scope, which makes partial matches inconsistent and rules out their ability to form a complete match. As for execution time overhead, the process of deciding whether a join point satisfies a given pointcut based on past program events is in general not statically decidable. A run-time component is needed to make the final residual checks for a match, introducing overhead. It is clear that the temporal pointcut language of VMADL needs to take precautions for the memory and run-time problems, otherwise the resulting VM is not usable in practice.

We have looked at solutions and workarounds for both problem areas. For the design of cHALO, we have focused especially on program history retention strategies, i.e. policies which limit the number of needed trace data elements in memory and/or the period during which they are stored. It is our belief that current approaches hide too much complexity from the programmer, without the possibility to get in charge if desired. This clashes with the spirit of C, similar to the way C programmers do not like garbage collection because it takes away too much power/freedom from them. Additionally, existing program history retention techniques almost all exploit garbage collection facilities of the underlying base language. Infrastructure software like VMs, traditionally implemented in C or C++, cannot benefit from this garbage collection. Hence, history-based pointcuts may cause serious memory issues in these systems. Lack of memory control and explicit retention strategies in the absence of garbage collection, severely limits adoption of history-based pointcut languages in C and C++ systems.

Although many dedicated analyses [27, 28] and optimisations [16, 17] have been proposed to reduce execution time overhead, history retention strategies have a large impact on this too. C programmers do not want to use coarse-grained temporal operators. In many cases, these match more join points than needed, and hence occupy more memory than desired. In most cases, there is not even a way to estimate the memory footprint, because this is either undocumented or too complex. Hence, history-based pointcut languages in C need fine-grained temporal operators with clear guarantees on memory behaviour.

To deal with history retention and fine-grained operators within cHALO, we propose to apply ideas introduced by HALO [115], an expressive history-based

```
1 void f(char* s,int d);
2 void g(char* s);
3 BOOL shorter(char* s,int d);
```

Figure 9.1: Procedure declarations of the running example used in this chapter.

pointcut language for Lisp. HALO features a limited set of logic-based pointcut primitives with well-known memory space and cleanup characteristics, and a run-time weaver implemented in terms of a slightly customised Rete-engine [94]. HALO's temporal operators and their history retention strategies are clearly mapped onto join nodes of the Rete-network. This mapping facilitates improvement of the (memory) behaviour of operators or the addition of new operators. We have translated these ideas to Aspicere2 in the form of cHALO.

To summarise, the design philosophy behind cHALO is that:

- History-based pointcut languages for C/C++ need to provide the programmer with explicit control over behaviour and memory footprint of temporal pointcuts.
- The clear mapping between temporal operators and join nodes of the Rete-network, makes HALO an ideal choice for providing the aspect developer with more control on memory and execution overhead.

The next section discusses two existing history-based pointcut languages and how they implement a running example which is used throughout this chapter.

9.2.2 The Design and Implementation of Two Major History-based Pointcut Languages

There are dozens of history-based pointcut languages, which primarily differ by the design of their language and their specific focus. Except for some approaches [45, 239, 185], they all have a similar state machine-based implementation, often without memory retention strategies [185]. For these reasons, we limit the discussion of related work to Arachne and tracematches, two of the most well-known history-based aspect languages. Later on (Section 9.2.3), we consider a third temporal aspect language, i.e. HALO.

To illustrate the claims of Section 9.2.1 about memory and execution overhead in the context of Arachne and tracematches, we use Figure 9.1 as a running example throughout this chapter. This example program consists of two functions, `f` and `g`, and a boolean function which checks whether or not the length of a string (`s`) is shorter than a given integer (`d`). We have to model a history-based pointcut which looks for (possibly non-consecutive) invocations of `f` and `g` which have the

```
1 seq(call(void f(char*,int)) && args(s,d)
2                             && if(shorter(s,d));
3     call(void g(char*)) && args(s2) && if(s==s2) then ...)
```

Figure 9.2: Arachne pointcut with dots instead of concrete `around`*-advice.*

same string value as their first argument. Furthermore, we are only interested in invocations of `f` for which `shorter` returns `TRUE`.

The next two subsections present the design and implementation of Arachne (as an instance of event-based AOP) and tracematches.

9.2.2.1 Event-based AOP and Arachne

Event-based AOP (EAOP) [70, 69] is a history-based aspect technology which is expressed in terms of execution monitors. Join points are reified as events in the execution of a program and pointcuts describe patterns of events which should be intercepted by the monitor. Douence et al. [70] have proposed a formally defined, domain-specific event-based aspect language with operators for join point sequences, parallel execution, filtering of join points, etc. They have developed a Java prototype in which the monitor is synchronously called by the base program. The specific instrumentation points where the monitor is invoked are automatically generated from the pointcut. The distinction between a run-time component (monitor) and the broadcasting of events from within the base code (instrumentation) is widely used in history-based aspect languages. Åberg et al. [1] have proposed another implementation of EAOP. They use rewriting rules, expressed in temporal logic, to select all join point shadows statically, i.e. without any dynamic residues, based on the intra-procedural control flow graph of a program. This static transformation avoids run-time overhead.

The most advanced incarnation of EAOP is Arachne [68], which we have described in Section 5.2.2.3. Arachne retains only the sequencing construct of EAOP, but this forms the backbone of Arachne. Figure 9.2 shows a pointcut that matches a sequence of calls to `f` and `g` which have equal first arguments and pass the `shorter` boolean check. The check for equality of the first arguments has to be specified explicitly (`if(s==s2)`). Arachne allows to associate advice to sub-pointcuts of a sequence, hence the advice represented by "`...`" is actually `around`-advice on the call to `g`, not on the sequence as a whole. This clearly gives aspect developers more control, but it can be misleading in the sense that advice is executed for any partial match, regardless whether it will eventually yield a complete match.

Run-time efficiency of the woven code has been an important concern during the design of Arachne. Aspects are transformed into a dynamic library which can

```
1 tracematch(String s,int d){
2  sym f around: call(void f(..)) && args(s,d)
3                && if(shorter(s,d));
4  sym g around: call(void g(..)) && args(s);
5
6  f g{
7    ...
8  }
9 }
```

Figure 9.3: A `tracematch` definition equivalent to the Arachne advice in Figure 9.2.

direct a weaver daemon to add instrumentation logic to the machine code of a running application, without halting it. The locations where modifications should be performed, can be derived from offline pointcut analysis.

For memory management, Arachne associates a linked list with each sub-pointcut (except the last one) of a `sequence`. Each list node stores the contents of a partial match, with values for all context variables which appear in the whole pointcut. At any moment, multiple partial matches can be associated with a given sub-pointcut. Conceptually, when a new trace element matches a sub-pointcut, every partial match of the previous sub-pointcut is examined to see whether the values of bound context variables are consistent with the context of the new trace element. If they are, the partial match's state can be updated and its node moved to the current sub-pointcut's list. Associated advice (if any) is executed. If the last sub-pointcut has been matched, the partial match's node is freed, i.e. returned to a pool. If a partial match never yields a complete match, the accompanying set of bindings remains indefinitely in one of the linked lists. This is actually inevitable, but we come back to this point in our discussion.

The next section discusses the design and implementation of tracematches.

9.2.2.2 Tracematches

Tracematches [12] consist of a set of events (symbols) which are considered interesting, a regular expression expressed in terms of those symbols and an advice body which is activated once the regular expression pattern is satisfied. Symbols correspond to a combination of an AspectJ advice kind (e.g. `before`) and a primitive pointcut. A tracematch model of the Arachne pointcut of Figure 9.2 is shown in Figure 9.3[3]. Two symbols are declared (`f` and `g`). Prolog-like unification of (and back-tracking over) the string arguments is enforced by reusing the same free variable name (`args(s)`). This forms the biggest difference between

[3]Tracematches are expressed in Java, whereas Arachne and Aspicere2 are based on C. For this example, only the temporal pointcut matters, not the advice.

tracematches and prior history-based pointcut languages, and has had a big impact on the memory behaviour of the tracematch implementation (more on this later). Line 6 contains a simple regular expression in terms of the two symbols. It expresses the same sequence as the Arachne implementation does. Contrary to Arachne, tracematch advice applies only to the last join point matched by the pattern, not to each sub-pointcut.

As introduced by EAOP, the tracematch weaver conceptually distinguishes between base code instrumentation and run-time support for deciding whether there is a match. Tracematches are translated into multiple AspectJ advices. Some of these advise the appropriate instrumentation join points with event signaling logic. Other advice contains specialised code to set up and run the run-time component. Finally, there is still the actual tracematch advice code. Many optimisation techniques have been proposed to optimise either the instrumentation or run-time side of the woven system [16, 17, 27, 28].

Tracematches use a specialised, deterministic finite automaton to keep track of the matching process. The automaton is automatically generated from and specialised to the tracematch. As automata are not designed to hold memory (for partial matches), some advanced concepts have been added on top of them [12]. Every state is labeled by a number of disjunctive constraints. These contain all partial matches for a given state in the automaton. Upon arrival of new events, the constraints are incrementally updated. Conceptually, partial matches are moved to the next state until they disappear from the automaton. It is not clear whether this automaton-based approach easily allows extending it with new temporal operators or history retention strategies.

The biggest problem in the tracematch implementation is the need to avoid memory leaks [12, 17] in the presence of free variables. Apart from HALO [115], tracematches is one of the only approaches which is worried about this. The problem is that bindings inside a partial match may suddenly refer to garbage-collected data without discarding the partial match from memory. To resolve this, the tracematch implementation uses in some cases weak references to hold context variables. As such, if a weak reference is garbage collected, the partial matches referring to it are also cleaned up. Still, getting this memory behaviour correct has proven to be very hard [12, 17]. As is the case with Arachne, leaks are inevitable if non-garbage collected partial matches never contribute to a complete match.

The next section presents the HALO history-based aspect language, which is later used to express the behaviour and memory requirements of Arachne and tracematches, and has been the main inspiration to cHALO.

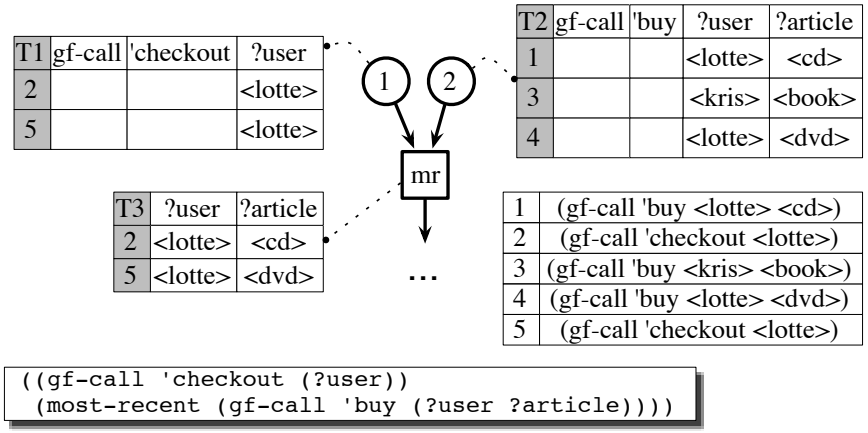


Figure 9.4: HALO's Rete representation for a `most-recent` pointcut (shown at the bottom) and a given program trace (bottom right table) [115].

9.2.3 HALO, a History-based Aspect Language for Lisp

HALO [115] is a history-based pointcut language for Lisp. It provides a limited number of temporal operators: `most-recent`, `cflow`, `since` and `all-past`. All of these relate an outer sub-pointcut to an inner sub-pointcut (two in case of `since`):

`a most-recent b` A new match for outer sub-pointcut `a` only matches with the most recent partial match of inner sub-pointcut `b` with which it can unify bound context variables.

`a cflow b` Similar to `most-recent`, but the join point satisfying `a` should lie within the control flow of a join point satisfying `b` (i.e. AspectJ's `cflow`).

`a since b c` A new partial match for outer sub-pointcut `a` matches with the partial matches of inner sub-pointcut `c` with which bound context variables can be unified and which have occurred later in time than the most recent partial match satisfying inner sub-pointcut `b`.

`a all-past b` A new partial match for outer sub-pointcut `a` matches with all partial matches of inner sub-pointcut `b` with which bound context variables can be unified.

We have listed these operators based on their memory footprint, from low to high. These memory requirements are transparently linked to the implementation of the HALO weaver [115]. This is a dynamic weaver based on a Rete engine [94]

which has been extended with time stamps and extra logic corresponding to the temporal operators. The Rete algorithm is a forward chaining technique[4] specialised in checking whether a sequence of asserted logic facts satisfies a logic formula (in the case of HALO: a logic pointcut matches a join point). The algorithm is based on a network of nodes, node connections and memory tables attached to the nodes, as shown on Figure 9.4. Round nodes are "filter nodes", as they filter newly asserted facts based on specified values for logic variables[5]. The filter nodes correspond to the conditions from the logic formula. In HALO, they correspond to sub-pointcuts. The rectangular node is a "join node", which tries to unify a new partial match which has entered via the node's left or right input node with the other input's partial matches. If there is a match, the node memorises it and sends the new match to its output. Partial matches are stored within memory tables attached to filter and join nodes.

By default, join nodes perform a logical "and" on the partial matches in their input nodes' memory tables. In HALO, each temporal operator is matched onto a custom join node (multiple in the case of `since`). The one on Figure 9.4 e.g. represents the `most-recent` operator in the HALO pointcut at the bottom. Instead of just performing a logical "and", the time stamps which have been added by HALO to memory tables (gray columns) are taken into account. Upon a new partial match in its left input node, the `mr` join node combines this entry with the most recent matching entry in the memory table of the right input node. This means that both entries need to have the same values for common variables (like a normal "and" requires) *and* that the time stamp of the right partial match is lower than the time stamp of the left entry. For this reason, if the `mr` gets a new partial match from its right input instead of from its left input, there will not be a match.

To illustrate HALO's weaver implementation, Figure 9.4 shows how the network has processed the program trace on the lower right. The asserted fact of time stamp 1 is only inserted and memorised by the right filter node, because it is a call to `buy` with two arguments. The join node does nothing, because it cannot match with the empty memory table of its left input. The fact at time 2, however, does trigger a full match, as it matches the left filter node and can be unified with an older fact memorised by the right filter node. The resulting match is stored inside the join node's memory table and is also sent to the output of the network to signal a full match to the weaver. By the time the fifth fact has been asserted, a pure Rete join node would match it with the partial matches of time stamps 1 and 4. However, the `mr` node is only interested in the most recent partial match on the right which matches with its left input, and hence only one full match is made and stored.

[4] Contrary to Prolog, which is a backward chaining programming language.
[5] Variable names start with a '?'.

```
1 (gf-call 'f <aa> <1>)
2 (gf-call 'f <aa> <3>)
3 (gf-call 'f <aa> <4>)
4 (gf-call 'g <bb>)
5 (gf-call 'g <aa>)
6 (gf-call 'g <aa>)
```

Figure 9.5: Example trace of the system in Figure 9.1.

```
1 ((gf-call 'g ?s)
2  (since (most-recent (gf-call 'g ?s))
3         (all-past (gf-call 'f ?s ?d)
4                   (if 'shorter ?s ?d))))
```

Figure 9.6: HALO pointcut corresponding to the Arachne pointcut in Figure 9.2 and the `tracematch` *in Figure 9.3.*

Until now, we have only focused on the direct mapping between temporal pointcuts and Rete network. Contrary to approaches like Alpha [185], the naive memory requirements of Figure 9.4 can be drastically optimised [115]. There are various history retention strategies which are based on time stamps and on the known behaviour of the temporal operators. Because the `mr` node is not connected with any other join node and just sends its partial matches to the output of the network, it does not need to store its partial matches, and hence does not need a memory table. A similar remark holds for the left filter node, as a newly asserted fact is not used anymore once it has been sent to the join node. Finally, the semantics of the `most-recent` operator suggests that duplicate partial matches stored within the filter node's right input can be removed, as only the most recent one is interesting. The interpretation of "duplicate" is more general than usual, as in the context of the `most-recent` operator it boils down to "having identical values for variables which are common between the inner and outer sub-pointcut". Other variables are not used during matching of left and right partial matches. Hence, in the example, fact 1 will become obsolete once fact 4 is stored in the right memory table, even though they have different values for the `?article` variable. This means that at time 5, the network in Figure 9.4 only requires memory space for two facts in the right input's memory table, nothing more.

With these concepts in mind, we now revisit Arachne and tracematches to express their behaviour and memory requirements in terms of HALO's temporal operators. This enables better understanding of problems related to memory footprint.

9.2.4 Modeling Arachne and Tracematches in Terms of HALO

To better understand the semantics and memory requirements of Arachne and tracematches, we rephrase the example Arachne and tracematch implementations of Figure 9.2 and Figure 9.3 in terms of HALO. As a side-effect, this exercise gives an indication on the fine-grainedness of these three pointcut languages with reference to each other, and on the link between temporal operators and memory management. We use the sample program trace in Figure 9.5 for the example system of Figure 9.1. Despite its simplicity, this example suffices to convey the main message.

For our example, the Arachne and tracematch implementations conceptually yield the same program output and memory consumption for partial matches. More precisely, the first event is ignored, as the integer is smaller than two (the string argument's length). The next two invocations do match the first sub-pointcut of the `sequence` and `tracematch`. Both approaches record the bindings `(?s -> <aa> ?d -> <3>)` and `(?s -> <aa> ?d -> <4>)`. The next three events try to extend the partial matches into a complete match. The fact on line 4 does not succeed, since `<aa>` differs from `bb`. The next one does trigger two complete matches, one per partial match. Hence, the advice is executed twice, once for `(?s -> <aa> ?d -> <3>)` and once for `(?s -> <aa> ?d -> <4>)`. Arachne now removes the two partial matches from the linked list associated with the `sequence`'s first sub-pointcut. Tracematches do the same by manipulating the constraints associated with the automaton's first state. This is important, as it means that the last event in the trace (line 6) will not trigger any advice. There are simply no partial matches left anymore.

So far, we have considered the program output and we have also observed that the behaviour is affected by the retention of partial matches. Arachne and tracematches by default retain every matching event as a partial match for a given sub-pointcut until it gives rise to a longer partial match and moves on to the next linked list or state, or is discarded on a complete match. Second, tracematches can also remove partial matches, as negative symbols cause constraints to disappear from a state. Third, analyses and optimisations indirectly influence the bindings. They try to shift as much decision logic as possible to compile-time, or to accelerate access to partial matches [16, 17, 27, 28]. Handling retention of program history is merely a side-effect, not the primary goal of these analyses. As the analyses are optional as well, programmers do not have control over the retention policy. This means that, in our example, all corresponding invocations of `f` need to be retained between two consecutive full matches for a given string argument. Assuming that these invocations are more frequent than calls to `g`, a lot of space is required.

We can express this execution and memory behaviour in terms of HALO concepts. Figure 9.6 shows the corresponding pointcut. The previous full match for a given string `?s` is modeled by `most-recent (gf-call 'g ?s)`, while the

pointcut expression on lines 3–4 selects all past calls to `f` for which the `shorter` test is satisfied. Taken together, lines 2–4 collect all calls to `f` for a given string `?s` since the previous full match of the whole pointcut. This is the set of partial matches with which each new call to `g` can be combined to form a complete match. This pointcut corresponds to the behaviour and memory requirements of the Arachne and tracematch pointcuts. The memory footprint consists of one entry to record the context of the last call to `g` for any given string `?s`, as well as facts for keeping the context of all partial matches of the call to `f` since the last corresponding `g` invocation.

From this, we can make a couple of observations. First, the basic history-based pointcut primitives in approaches like Arachne and tracematches are actually coarser-grained compared to the temporal primitives HALO provides. The reason is that both want to make sure that any event which gives rise to an initial partial match will get a chance to form a complete match. No partial match is considered redundant or a duplicate, except when there are negated sub-pointcuts, a new sub-pointcut is matched or when a binding is garbage-collected. This behaviour is sometimes desired, but in many use cases it suffices to act just once on a series of more or less identical events. Timer alarms, scheduling requests, etc. in a VM could be implemented using the simpler, finer-grained temporal operators of e.g. HALO to reduce unnecessary execution and memory overhead.

A second observation is a consequence of the above coarse-grained language semantics. As illustrated by the extensive coverage of memory leaks by the tracematch team [12, 17], the memory requirements of history-based pointcut implementations are rather opaque and implicit. This conflicts with the spirit of C and C++, as explicit, intuitive control over memory is one of the hallmarks of these languages. It is not immediately clear how to map the memory retention policy of a temporal operator to one specific concept of the run-time components. In HALO, an operator corresponds by design to one specific join node of the Rete network, with guarantees on memory retention of the associated partial matches. This also means that adding a new operator boils down to introducing a new join node type. As a consequence, the aspect developer has more control. This gives systems software developers the possibility to tailor temporal pointcuts to the application at hand, and to make well-founded assessments of memory requirements and execution speed.

Based on these observations, we have designed Aspicere's temporal pointcut extension, cHALO, based on some of HALO's key features. We present the resulting system in the following section.

```
1 ((gf-call 'g ?s)
2  (most-recent (gf-call 'f ?s ?d)
3               (if 'shorter ?s ?d)))
```

Figure 9.7: Sequence pointcut in HALO which is finer-grained than Arachne and tracematch sequences.

```
1 void fg_sequence(char* S,int D) around [around F,around G]:
2  invocation(F,"f") && args(F,[S,D]) && if(shorter,S,D) =>
3  invocation(G,"g") && args(G,[S]){
4    ...
5 }
```

Figure 9.8: Aspicere2 pointcut which corresponds to the HALO pointcut of Figure 9.7.

9.2.5 cHALO, a History-based Extension of Aspicere

This section first presents the design of cHALO (Section 9.2.5.1), followed by a brief overview of the underlying implementation (Section 9.2.5.2) and a demonstration of cHALO's extensibility in the CSOM case study (Section 9.2.5.3).

9.2.5.1 Language Design of cHALO

Figure 9.7 shows a variant of the Arachne and tracematch pointcuts from Section 9.2.2, similar to Figure 9.4. A call to `g` can trigger at most one complete pointcut match, because there can only be one `f` fact in memory with the same values for common bindings (see Section 9.2.3). The others have been identified as duplicates (based on `?s`) and have been removed subsequently. In many cases, this pointcut's behaviour suffices instead of the coarser-grained semantics of Figure 9.2 and Figure 9.3. Furthermore, for the example of Figure 9.7, at most one fact is stored for the calls to `f` in the trace of Figure 9.5, while none of the calls to `g` warrants storage. Note that the most recent partial match which satisfies the inner sub-pointcut is not removed upon a complete match [115], contrary to Arachne [68] and tracematches [12]. Although partial match 1 in Figure 9.4 has contributed to the complete match on time stamp 2, it is only removed from the second filter node's memory table on time stamp 4, i.e. when it has become a duplicate.

Figure 9.8 shows the corresponding pointcut (and advice) in cHALO, i.e. the history-based extension of Aspicere. Line 1 tells us that `fg_sequence` is `around`-advice which matches on a sequence of two symbols, `F` and `G`. It provides two context variables, the string (`char*`) and integer arguments, to the advice. Lines 2 and 3 show that `F` and `G` correspond to invocations (calls) to the `f` and `g` procedures

```
1 ((gf-call 'g ?s)
2  (very-most-recent (gf-call 'f ?s ?d)
3                    (if 'shorter ?s ?d)))
```

Figure 9.9: Sequence pointcut with more limited fact retention policy. This behaviour can be obtained in Aspicere2 if the => is replaced by ~> in Figure 9.8.

mentioned in Figure 9.1. The arguments of these calls are captured (`args`) and the `F` symbols are filtered by the `shorter` boolean function. The => arrow identifies the semantics of the sequence, in this case HALO's `most-recent` operator.

9.2.5.2 Weaver Implementation of cHALO

This section discusses the implementation of cHALO's weaver. HALO has a dynamic weaver which relies on a slightly customised Rete [94] engine, whereas Aspicere2 sports a link-time weaver based on a Prolog engine. These two approaches are not at odds with each other. As we have seen in Section 9.2.2.1, most implementations of temporal aspect languages [70, 16] conceptually combine base code instrumentation with run-time decision logic. The former makes sure that only interesting events are signaled, while the latter decides at run-time whether the recorded join points match a pointcut or not. As identified by Avgustinov et al. [16], both components can be optimised independently to reduce memory footprint and execution time. Using static analysis, one can filter out join point shadows which can never lead to a match, specialise the run-time logic to the pointcuts in use or even eliminate run-time checks altogether. Many optimisations are obvious and do not require whole-program analysis [16], but especially instrumentation can be sped up considerably via whole-program analyses [27].

For cHALO, we apply Aspicere2's Prolog-based link-time weaver to pinpoint suitable instrumentation join points, while a run-time Rete-engine (a modified version of CLIPS[6]) is linked with the woven application to perform the run-time checks. Instrumentation then amounts to asserting facts to the engine, and checking whether or not a rule has been triggered, in which case advice should be executed. Internally, Aspicere2 transforms the advice of Figure 9.8 into normal advice. Static analysis, inter- (whole-program) or intra-procedural, can be added to limit the number of join point shadows where facts need to be asserted. Run-time optimisation can be achieved by garbage collection of facts based on the temporal operators' semantics. Because the CLIPS engine is highly configurable, redundant functionality can easily be excluded from the library to reduce its footprint. Note that Aspicere2's `cflow` implementation is not based on this Rete-engine, but is implemented "natively" via whole-program analysis on the static call graph of the

[6] `http://clipsrules.sourceforge.net/`

```
1 void Shell_start(){
2   ...
3   SEND(current_frame, set_bytecode_index, bytecode_index);
4
5   /*HERE*/
6   SEND(current_frame, push,
7        Universe_new_instance(runClass));
8
9   SEND(current_frame, push, it);
10  ...
11 }
```

*Figure 9.10: CSOM shell initialisation code. The statement on the line marked "/*HERE*/" should be advised by the native threading service.*

```
1 void run_class_pushing(pVMObject RunInstance,pVMObject*
2       Jp1Ret,pVMFrame CurrentFrame,pVMClass* RunClassPtr)
3      after [after_returning Jp1,after Send]:
4  shell_class_instantiation(Jp1,RunClassPtr,RunInstance)
5  && dereference(Jp1Ret,RunInstance)
6  && withincode(Jp1,"Shell_start")
7  ~> current_frame_pushing(Send,CurrentFrame,RunInstance)
8  && if(RunInstance,is_shell_class_instance){
9    // load the system class and create an instance of it
10   pVMObject sys_obj = Universe_new_instance(system_class);
11
12   //push system on stack
13   SEND(CurrentFrame, push, sys_obj);
14
15   //push shell class on stack
16   SEND(CurrentFrame, push, (pOOObject)*RunClassPtr);
17 }
```

Figure 9.11: Aspicere2 advice which performs extra initialisation for the native threading service when the marked statement of Figure 9.10 is executed.

base code [14].

9.2.5.3 Application of cHALO to CSOM

To demonstrate the extensibility of cHALO, we consider an example from the CSOM case study. Figure 9.10 shows part of the implementation of the CSOM shell's initialisation code. The shell is an interactive environment in which code can be entered for execution by the CSOM VM. Our service implementation of

```
1 shell_class_instantiation(Jp,RunClassPtr,RunInstance):-
2   invocation(Jp,"Universe_new_instance"),
3   get_local_variable_ref(Jp,"runClass",RunClassPtr).
4
5 current_frame_pushing(Send,CurrentFrame,RunClassInstance):-
6   execution(Send,"_VMFrame_push"),
7   args(Send,[CurrentFrame,RunClassInstance]).
```

Figure 9.12: Prolog predicates used by Figure 9.11.

native threading[7] needs to advise the statement marked by /*HERE*/ for initialising itself. The problem is that this statement is located in the middle of the `Shell_start` procedure, and that there are multiple expansions of the `SEND` macro with `push` as its second argument. This means that conceptually the `push` instance method is invoked on the `current_frame` object. The only way to distinguish the marked statement from other ones without changing the base code[8], is to take temporal information into account.

A temporal pointcut based on Figure 9.8 seems appropriate to explicitly match the execution of the `push` instance method with the most recent invocation of the `Universe_new_instance` procedure. However, as we have noticed earlier about the `most-recent` operator, the partial match corresponding to the execution of the `push` method is not discarded on a complete match. This is only done when a duplicate entry of this partial match is memorised. In the meantime, the partial match can contribute to other complete matches, which is not what we intend the advice to do. Hence, we want to limit the life time of partial matches to change the semantics of the advice and to reduce the memory footprint. This can be done by extending cHALO.

Extension of cHALO happens by defining new kinds of join nodes, with specific matching and memory cleanup behaviour. For the CSOM case study, we have added a `very-most-recent` operator which extends `most-recent` by removing the partial match of the inner pointcut on a match with a new event of the left filter node. To choose this operator instead of the default one, one just needs to replace $=>$ by $\sim>$. Figure 9.11 shows an advice with a temporal pointcut which uses the `very-most-recent` operator (line 7). The pointcut matches when a `current_frame_pushing` event can be unified with the most recent `shell_class_instantiation`. The former corresponds (Figure 9.12) to the execution of the `_VMFrame_push` procedure (internal name for the `push` instance method), whereas the latter denotes a call to `Universe_new_instance`.

[7] This is a technique which implements multithreading in the VM itself instead of relying on the operating system's threads or processes.

[8] If the base code can be changed, the marked statement can be extracted into a procedure such that its invocation can be advised.

On a match, the partial match for `shell_class_instantiation` is immediately discarded. In practice, before and after the marked statement of Figure 9.10 no memory is taken to store partial matches.

cHALO gives aspect developers the possibility to define more specialised temporal operators which give them more explicit control over semantics and memory footprint if this is needed. Other operators like `all-past` and `since` have not been added yet to cHALO, but in principle the same approach can be followed.

9.2.6 Open Problems of History-based Pointcut Languages for C

Explicit program history retention strategies can only stabilise memory footprint of history-based pointcuts if the latter are well thought out. Suppose that in the example trace of Figure 9.5, invocations to `f` with identical string arguments are very rare. This would mean that the garbage collection optimisation does not kick in, and almost all calls to `f` are memorised. This could get worse when calls to `g` would also be infrequent, or would not contain matching string arguments. In that case, even the `very-most-recent` operator would not be able to reduce the memory footprint. The same problems exist in Arachne and, to a lesser extent (because of weak references), in tracematches.

Apart from memory starvation, there is a second danger. If a very old partial match suddenly leads to a complete pointcut match, chances are high that its context variables are not valid anymore. They could refer to local variables which have gone out of scope already, or to pointers which have become dangling by now. Tracematches can easily deal with this via weak Java references, but in C or C++ no such mechanism exists, simply because these languages do not provide garbage collection. This is the largest obstacle history-based pointcuts are facing in the context of C and C++ systems. Providing more explicit control on memory requirements of temporal pointcuts, is our suggestion to deal with this problem as long as no solution has been found. Such a solution could e.g. advise any function with instrumented shadows in its body to invalidate all partial matches which bind local variables when control reaches the end of the function. Alternatively, some kind of smart pointer could be used to refer to bound context instead of native pointers. This second problem should be dealt with before cHALO, or any other history-based pointcut language, can be safely used in practice.

In our examples, we have used a boolean function to check a condition on captured context. This is trickier than it appears at first sight. Approaches like Alpha [185] check for matches using a backward chaining logic language. As such, the current values of context variables are used to match past facts. HALO's forward chaining weaver works the other way around. Facts are asserted in the past with the then current value of context bindings. This value may differ from

the one in use when the complete pointcut eventually matches, e.g. in the case of pointers where the value pointed at may have changed in the meantime. This concept is called "future variables" in HALO [116, 115].

Conceptually, a partial match's context variables should retain their original value for as long as they are stored in memory, although several gradations of this behaviour could be devised [116]. In all cases, some flavour of deep copying is required to make future variables work in C or C++. This is not easy, as making a deep copy of a generic `void` pointer is not possible. Ideally, the weaver should be able to generate specialised deep copy algorithms for each declared context type, but this is not straightforward either. Currently, cHALO takes shallow copies of context variables, i.e. pointer values (addresses) are copied, but not the memory content they refer to. This means that the original value stored in the referenced memory is not necessarily preserved.

9.2.7 Validation #1: Aspicere Meets Goal L1

cHALO is an expressive, flexible and fine-grained history-based pointcut language, based on HALO. It explicitly deals with retention policies for partial matches and offers a transparent connection between temporal operators and memory footprint. A number of difficult issues remain to be solved before cHALO can be safely used in the VM setting it originally is conceived for. Profiling data is required to determine which challenge has to be dealt with first. In the meantime, extension of cHALO by specialised temporal operators with specific behaviour and memory requirements enables developers to deal with these issues in well-known contexts, such as the application of VMADL in the CSOM case study.

9.3 Impact on the Build System

This section discusses the impact on the build system of the introduction of Aspicere and cHALO in the CSOM VM. We consider the following four build problems:

1. physical integration of Aspicere2 with the build process
2. configuration presents a challenge
3. migration to the re-engineered system
4. increase of build time and incremental weaving

9.3.1 Integration of Aspicere2 with the Build Process

Figure 9.13 shows the build dependency graph of CSOM. It consists of one main executable (left cluster) and a dynamic library of native code (right cluster) to

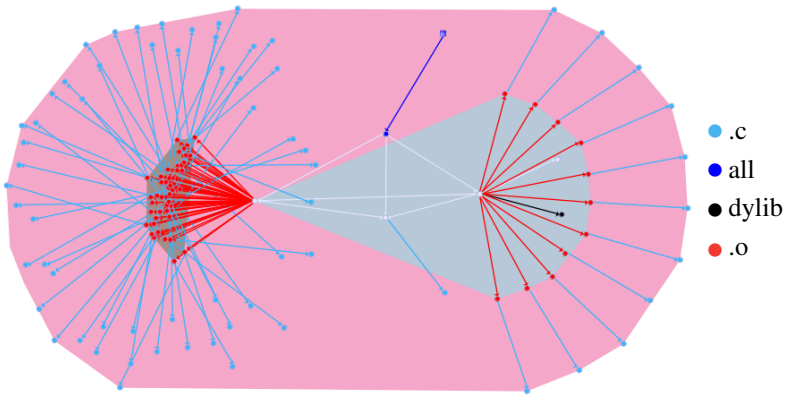


Figure 9.13: Build dependency graph of CSOM. The left cluster is the CSOM executable, while the right one represents the standard library (the header files in the left cluster have been hidden).

```
1  CC = gcc
2  ...
3  CSOM_LIBS = -ldl -lpthread
4  ...
5  INCLUDES = -I$(SRC_DIR)
6  ...
7  CSOM_OBJ = $(MEMORY_OBJ) $(MISC_OBJ) $(VMOBJECTS_OBJ) \
8             $(COMPILER_OBJ) $(INTERPRETER_OBJ) $(VM_OBJ)
9  ...
10 CSOM: $(CSOM_OBJ)
11   @echo Linking CSOM
12   $(CC) $(LDFLAGS) `./ostool.exe l` -o \
13         `./ostool.exe x "$(CSOM_NAME)"` $(CSOM_OBJ) \
14         $(CSOM_LIBS)
15   @echo CSOM done.
```

Figure 9.14: Original makefile of CSOM.

support the CSOM standard library. As a side note, the build rule for the library contains a mistake, as its build target is not the constructed dynamic library, but rather a virtual target named `CORE` (white node in the center of the right cluster). Accidentally, there is a node for the dynamic library which is actually built, but it occurs as an implicit dependee of the `CORE` target (black node on top of the gray hull). In other words, MAKAO's implicit dependency detection algorithm (Section 3.4.3) warns us about this small build mistake.

Aspects have been woven into the executable and the dynamic library. This choice is important, because it means that join points which lead to a partial match

```
CC = llvm-gcc -emit-llvm
...
CSOM_LIBS = -ldl -lpthread
ASPECTS = `paste -s -d\  $(ROOT_DIR)/aspects/aspects.lst`
...
INCLUDES = -I$(SRC_DIR) -I/usr/include
CSOM_OBJ = $(MEMORY_OBJ) $(MISC_OBJ) $(VMOBJECTS_OBJ) \
           $(COMPILER_OBJ) $(INTERPRETER_OBJ) $(VM_OBJ) \
           $(NATIVE_SERV_OBJ)
...
.vmadl.ac:
  vmadl.sh $< -o $@
...
CSOM: CSOM.bc
  @echo Weaving CSOM
  lto.sh $(LDFLAGS) -Wp,-I$(ROOT_DIR)/src `./ostool.exe l`
         -Wl,-Y,1455 -mmacosx-version-min=10.4 -Wl,-x \
         `./ostool.exe x "$(CSOM_NAME)"`.bc -o \
         `./ostool.exe x "$(CSOM_NAME)"` -aspects \
         $(ROOT_DIR)/aspects/aspects.lst -modules \
         $(ROOT_DIR)/aspects/modules.lst -user \
         $(ROOT_DIR)/aspects/user.lst -L/usr/lib \
         $(CSOM_LIBS)
  @echo CSOM done.

CSOM.bc: $(ASPECTS) $(CSOM_OBJ)
  @echo "Removing remains of aspects..."
  find . -name "*.ac.*" | xargs rm -f
  @echo Linking CSOM
  link.sh $(LDFLAGS) -Wp,-I$(ROOT_DIR)/src \
          `./ostool.exe l` \
          -o `./ostool.exe x "$(CSOM_NAME)"` -aspects \
          $(ROOT_DIR)/aspects/aspects.lst -modules \
          $(ROOT_DIR)/aspects/modules.lst -user \
          $(ROOT_DIR)/aspects/user.lst $(CSOM_OBJ) \
          $(CSOM_LIBS)
  @echo "Removing remains of aspects..."
  find . -name "*.ac.*" | xargs rm -f
```

Figure 9.15: Modified makefile of CSOM for native threading.

	number of new...			number of modified...				#added
	.c	.h	lines	.c	.h	lines	loci	lines
reference counting	0	0	0	15	3	22	37	170
mark-sweep	0	0	0	7	2	39	25	332
native threading	3	3	268	5	5	3	13	333
green threading	1	1	148	5	3	2	9	78

Table 9.1: Statistics with the number of new and modified files, and the number of added lines of code for adding the four extensions to CSOM.

of a cHALO pointcut may generate a complete match for join points which occur either during execution of code of the executable or of the library. This has been one of the main reasons for cHALO's history retention strategies.

Because there is only one important makefile (92 SLOC), we again have manually modified the build scripts guided by MAKAO's visualisation and querying. Figure 9.14 shows the relevant makefile snippets for the main executable (`CSOM` on line 10) when the native threading service is added. `CSOM` is compiled from a list of all relevant object files (`CSOM_OBJ`). The modified makefile looks like Figure 9.15. Similar to the Quake 3 case, we have changed a number of variables to integrate LLVM (`CC`) and we have split up the linking activities in two, i.e. one rule for generating the bitcode link module (lines 26–38) and one for the actual weaving (lines 14–24). Similar rules exist for the dynamic library. MAKAO has enabled us to manually integrate Aspicere2 in the CSOM build system.

9.3.2 Configuration of Aspects Presents a Challenge

There are some important differences with the Quake 3 case, especially in the area of configuration. As Table 9.1 shows, the native and green threading implementations not only consist of an Aspicere aspect and Prolog module generated from the VMADL description, they also need additional support code to be compiled with the system. The two garbage collection algorithms on the other hand only require extensive changes inside the base modules (in/decrementing reference count, etc.). Green threading needs one extra module to support the aspect, whereas native threading requires three modules (for scheduling, signals, etc.) and the "pthread" library to be linked with the woven application. As a side-effect, the API (header files) of this library should also be available to the compiler during the build. From these observations, we can deduce that:

- Support code and libraries for aspects should be taken into account when integrating AOP into the build.
- Depending on the aspect configuration, these extra artifacts need to be selected during the build or not, and related variables for e.g. header file di-

rectories should be modified or not.

The first item presents a big problem when the same aspect is applied in various subcomponents. Especially if these components later on interact with each other, e.g. by linking a library and an executable, keeping track of which component should control the support code becomes a hassle. There is no silver bullet to solve this problem, as it depends on the particular build architecture of the software system. The second issue is even worse. The inherent flexibility/pluggability of the aspects at the source code level becomes restricted at the build system level if there is no direct feedback from the aspect configuration to the compiler commands and build script variables. This is similar to the problems de Jonge [56] has encountered in systems without AOP (RC1). In the case of CSOM, we have not yet focused on pluggability or interaction between aspects, hence we have hard-coded the dependency on `pthread` on line 3 of Figure 9.15, have specified the extra header file directory on line 6 and have added the three modules of the native threading aspect on line 9.

A second observation about configuration is the integration of the VMADL preprocessor. Because aspect weaving occurs at link-time, we have added the aspects as a dependency of the link module rule (line 26) and have written a generic build rule (lines 11–12) which generates a .ac file (Aspicere) from the corresponding .vmadl file (VMADL). In bigger systems, where possibly multiple aspect configurations are used at once (e.g. one per library), a more sophisticated solution is needed. Again, a well thought-out strategy is required to keep aspect configuration manageable.

9.3.3 Migration to the Re-engineered System

Similar to the case study of the previous chapter, we have gradually migrated from a tangled implementation of each CSOM service to an aspect-based one (with VMADL). This migration consists of a planning and execution phase. For the planning phase, we have first specified initial VMADL definitions based on knowledge about the tangled service implementations. Then, we have marked the affected join point shadows in the source code with the name of the advice which would advise them, and have also identified the base code which should be eliminated. After this process, we have estimated the impact of the advice on the base code, i.e. the dependencies between advice, and between base code and advice. The outcome of this process is an ordered list which specifies the weaving order of advice and gives indications about which base code fragments have to be commented out. This concludes the planning phase.

The execution phase is a long iterative process in which for every advice of the list, the obsolete base code is commented out, the advice is woven and the test suite of the system is re-executed. In case of problems, we modified the VMADL,

regenerated the aspects and re-executed the current iteration of the migration. Even in an application of the size of CSOM, this gradual process takes a long time. This only reinforces the arguments made for the migration of the ASML system. However, during the migration, we have not needed to change the build scripts. The changes of Figure 9.15 only had to be made once. In larger systems like ASML, this is unlikely.

The test suite of CSOM consists of a number of CSOM programs with known output. Testing only required to run these programs and to automatically compare the outputs. As such, no changes were needed to accommodate aspects in the test infrastructure.

9.3.4 Increase of Build Time and Incremental Weaving

There are a number of factors which slow down weaving. As described in Section 9.2.5, whole-program static analysis is very useful to reduce the execution overhead of temporal pointcuts or `cflow`. This has implications on the choice of a weaver (e.g. compile-time vs. link-time), as discussed multiple times before, but also on the duration of the build process. These analyses are known to be slow [26] and, worse, they usually cannot be performed incrementally. This adds to the link-time weaving overhead reported in the Quake 3 case.

Our gradual migration approach causes extra problems. Each change to a base code file or VMADL declaration causes the bitcode link module to be regenerated, followed by re-weaving of the whole system. This is very costly in developer time. A source-to-source weaver on the other hand does not have these problems, but such a weaver cannot easily perform whole-program analysis. Gluing all source files together right before compilation is not an option, as visibility rules cause conflicts. Some overhead could be prevented by using a check sum tool and a code pretty-printer instead of "make"'s time stamps, as minor changes to comments should not cause re-weaving.

9.4 Validation #2: Roots of Co-evolution Experimentally Confirmed

The CSOM case study has confirmed various build problems (RC1 and RC2) we have encountered in earlier cases, but has also given evidence of new problems, especially regarding RC3 and RC4. Incremental weaving is important for developer productivity (RC3). During the migration from the CSOM VM with a tangled implementation of a service to a VMADL version (based on Aspicere), bug fixes in the base code and aspects have required frequent re-weaving. We have noticed that significant weaving speedups cannot be made through techniques like "trickle-down" recompilation [9]. Instead, the weaver should provide direct support for in-

cremental weaving. This is not straightforward, because the static analyses which are used to optimise the woven code (e.g. to limit the instrumentation for cHALO) mostly cannot incrementally update their results with the changes made in the base code. Even worse, prior weaving results can be invalidated. Optimisations performed on base code modules which have not been changed since, suddenly can be based on (by now) invalid analysis results. Hence, these modules should also be re-woven. This is a serious threat for weaver implementers. The build system cannot readily solve this because it does not understand the fine-grained aspect dependencies.

Second, aspects require better means for configuration than systems without AOSD (RC4). The migration of the original system to the fully re-engineered one has been hampered by dependencies of aspects on accompanying support modules and libraries. These have to be built at the right time if an aspect has been selected during configuration. If the aspect is not selected, the support code is also redundant. This means that configuration information should be linked to build rules, commands and variables in the build scripts. The problem is that during evolution of the aspects, new aspects and support code will be added, whereas others will disappear or be modified. By virtue of co-evolution of source code and the build system, these source code changes should be converted to corresponding changes in the build system. This is not easy, as the invocation of an aspect weaver is highly dependent on the build architecture and configuration system. In the CSOM case, support code was limited, hence we have been able to deal with this manually.

9.5 Validation #3: MAKAO Achieves Goal T2

MAKAO has enabled us to understand the build architecture of the CSOM system. Visualisation and querying have facilitated the physical integration of Aspicere2 with the build system, and the management of support code.

9.6 Conclusion

In this chapter, we have examined the CSOM case study, in which Aspicere has been used as the underlying implementation of the VMADL architectural description language. For this, we have enhanced Aspicere with an expressive, history-based pointcut language for C, cHALO, which is aimed at providing the programmer with sufficient control over memory usage (goal L2). Existing program history retention strategies have been designed explicitly for languages with garbage collection. Second, they make the connection between a temporal operator's behaviour and its memory footprint implicit. This clashes with the philosophy of C/C++ programmers. We have illustrated this via a running example implemented

in Arachne, tracematches and HALO. To remedy these problems, we have proposed to apply key ideas from the HALO pointcut language. HALO contains a limited set of temporal operators, each with a well-known behaviour and memory footprint. A clear connection between operators and the weaver's underlying Rete-engine enables developers to estimate the run-time costs of their pointcuts. We have translated these concepts to Aspicere2, and have illustrated the resulting cHALO system's extensibility by adding a new temporal operator which has been used in the CSOM case.

On the level of the build system, we have identified similar issues like in the previous cases, but also found new evidence for RC3 and RC4. Aspects often require support code which needs to be built along with the base code. However, depending on the aspect configuration the support code should be compiled or not. This decision also depends on the build architecture, which is reified by MAKAO (goal T2). A second new element is the introduction of the VMADL preprocessor, i.e. a new construction tool, the gradual migration to an AOP-based implementation and the necessity of static analysis to optimise the weaving results. Together, these introduce a lot of extra weaving activity. Hence, incremental weaving is a key property to successfully apply AOP in this case. The next chapter presents the fifth and final AOP case study.

[...] preprocessors are pragmatic tools which bridge the gap between PITS [(programming-in-the-small)] and PITL [(programming-in-the-large)]. On the one hand, most preprocessors have been designed by PITS practitioners and are themselves (degenerated) languages. On the other hand, preprocessors try to partially solve a major PITL problem: software variation. Unfortunately, their extensive use leads to severe maintenance problems.

Jean-Marie Favre [83]

10

Case Study 5: Extracting Preprocessor Code into Aspects

IN systems software, the C preprocessor is heavily used to manage variability and to improve efficiency. It is the primary tool to model crosscutting concerns in a very fine-grained way, but leads to extremely tangled and scattered preprocessor code. In this chapter[1], we present a case study in which the Parrot 0.4.14 VM is re-engineered with aspects to decouple the base code from preprocessor-driven conditional compilation. This re-engineering is based on a classification of existing conditional compilation patterns in the Parrot VM. Note that this VM is a new, and not a legacy system, but because it makes heavy usage of the preprocessor this case study suffices to distill important lessons in the context of this dissertation. Integration of the configuration layer with Aspicere's logic fact base (goal L2) controls whether advice should match or not. MAKAO is used to support build system understanding and re-engineering of the build system (goal T2).

We first describe the context of the case study (Section 10.1), followed by an explanation of the aspects which have been used (Section 10.2) and the impact of this on the build system (Section 10.3). Afterwards, Section 10.4, Section 10.5 and Section 10.6 present the validation results for build problems suggested by the four roots of co-evolution, MAKAO (goal T2) and Aspicere (goal L2). Finally, Section 10.7 presents the conclusions of this chapter, which is the final AOP case study of this dissertation.

[1] Parts of this chapter are based on [7].

10.1 Rationale behind the Case Study

Systems software manages and controls the hardware to support the tasks of application software. Over the years, a large body of long-living systems has accumulated, entailing operating systems, device drivers, compilers, virtual machines, etc. Many of these systems have configurable parameters built into the source code to tailor the product to one specific hardware platform or to allow different variants to be configured. Usually, such systems are developed in C or C++, with heavy usage of the C preprocessor to handle this variability.

In Section 2.3.4.4 and Section 2.1.3.1, we have discussed the C preprocessor's three most important constructs (`#define`, `#include` and `#ifdef`) and the many mechanisms through which these constructs, especially conditional compilation, are driven by the build system's configuration layer. As a concrete illustration of the powerful, flexible nature of the resulting coupling between configuration layer and base code, we have observed in Chapter 4 how the Linux kernel build system exploits these mechanisms to manage the kernel's variability.

However, the preprocessor's flexibility and the fact that it treats source code as text files without respecting the C syntax rules, make the C preprocessor an important source of programming errors and of confusion during source code analysis [211]. The programmer actually writes two programs: the C preprocessor program and the C program. The result of the C preprocessor program can be one of a multitude of C programs. To understand the software system, one can either examine the preprocessed code or the unpreprocessed version. The former corresponds to normal C code, but only represents one configuration. Unpreprocessed code is the opposite, as it contains all configuration code, but is not necessarily valid C code. This is not the only problem which hampers program understanding. Complicated macro bodies, side-effects and dynamic scoping, unsafe and inconsistent usage, multiple definitions, etc. are frequent sources of confusion. Nevertheless, every C programmer has to deal with the C preprocessor, because it is the primary mechanism for C to deal with different variants, to handle platform-dependent code, to make features disablable, to define syntactic sugar, etc. Even C++ systems still use the C preprocessor [169].

Singh et al. [204] have proposed to extract conditionally compiled code into aspects, as conditional compilation corresponds to a scattered and tangled representation of crosscutting concerns, i.e. the preprocessor is a fine-grained, but low-level aspect weaver. Conditional compilation is typically used to manage platform-dependent code, to enable or disable features, to guard debugging logic, etc. in a fine-grained way. As a consequence of this fine-grainedness, source code rapidly becomes a maze in which normal program flow is hard to distinguish from (often nested) conditional compilation checks, i.e. the conditional code is tangled with the main logic. Support for a particular platform, or debugging guards are spread

(scattered) throughout the system. These scattered code blocks highly differ from one place to another, and hence correspond to "heterogeneous concerns" [51]. In other words, conditionally compiled code implements crosscutting concerns, but cannot mitigate tangling and scattering phenomena. Hence, extraction of this logic into aspects could disentangle the base code from conditional compilation logic, while the variant code would be modularised.

There are many challenges involved with extracting conditionally compiled code into aspects. To make these challenges explicit, we have made an outline [7] of an approach for exploration and extraction [130, 170] of aspects from preprocessor-driven source code, based on a set of typical conditional compilation patterns. Exploration and extraction tools and techniques have to be able to parse heavily preprocessed code, to derive dependencies between preprocessor flags, to identify free variables in conditional logic, to automatically generate a pointcut, etc. The aspect language on the other hand should provide fine-grained join points and should be able to express highly variable advice [32]. Dealing with these challenges for automatic exploration and extraction is still work-in-progress.

Instead, this chapter investigates existing patterns of conditional compilation usage in a concrete virtual machine implementation, i.e. the Parrot VM. Such characterisations have been made before [80, 221], but here we explicitly focus on the ability of their structure to be extracted into advice. The categorisation corresponds to the one presented in [7], but has been elaborated. Knowledge of this categorisation is necessary to determine which features (join points, pointcut primitives, etc.) aspect languages should have to be able to model the fine-grained crosscutting exposed by conditional compilation. This is a prerequisite before automatic exploration and extraction techniques can be designed.

The next section presents the identified categories, and discusses for each of them the possible aspect implementations. Section 10.2.4 validates whether Aspicere is able to model the extracted conditionally guarded code as advice. Afterwards (Section 10.3), we discuss the impact on the build system.

10.2 Application of AOP in the Source Code

This section discusses the patterns of conditional compilation we have identified in the Parrot VM, and how we have managed to implement some of them as aspects. We have distilled these patterns by manual examination of the source code of the actual VM, i.e. not of the compiler or specific language implementations on top of Parrot. The investigated source code comprises 150 kSLOC of C code (129 files) and 15 kSLOC of PMC ("Parrot Magic Cookie") code (88 files). The latter is C code which is enhanced with syntactic sugar for defining classes which represent user-defined data types used inside the VM. A preprocessor tool expands PMC files into C code (100 kSLOC). In 79 C and 14 PMC files, conditional compilation

```
1 #ifdef NDEBUG
2 #  define TRACE_FM(i, c, m, sub)
3 #else
4 static void
5 debug_trace_find_meth(PARROT_INTERP, NOTNULL(PMC *_class),
6                   NOTNULL(STRING *name), NULLOK(PMC *sub)){
7   ...
8 }
9 #  define TRACE_FM(i, c, m, sub) \
10     debug_trace_find_meth(i, c, m, sub)
11 #endif
```

Figure 10.1: Conditional Definition of the `debug_trace_find_meth` procedure in src/objects.c.

occurs.

We have distilled three classes of conditional compilation: "Conditional Definitions" (Section 10.2.1), "Fine-grained Conditional Compilation" (Section 10.2.2) and "Coarse-grained Conditional Compilation" (Section 10.2.3). The latter two can be further divided in respectively five and two subcategories. This categorisation is presented in the following subsections. For each of them, a representative example is given and we discuss whether an aspect implementation is feasible and if so, how this would look like. Afterwards, we discuss our findings (Section 10.2.4).

10.2.1 Class 1: Conditional Definitions

Conditional Definitions are the cleanest application of conditional compilation, as the conditional guards are used to decide whether a complete C construct like a type or procedure definition should be compiled or not. This pattern is also used to conditionally include header files and to guard header files against problems of multiple inclusion (e.g. duplicate definitions of types). As a special case, we have found several cases where conditional compilation is used to adapt the signature of procedures to ANSI C or K&R C, based on the capabilities of the compiler.

Figure 10.1 shows an example of a Conditional Definition, in which the definition of the `debug_trace_find_meth` procedure (lines 4–8) is only selected in debugging mode (`NDEBUG` is not defined), in which case it is used for the implementation of a trace macro (lines 9–10).

There are various ways in which we can decouple the base code from conditional compilation logic. The most straightforward approach is to extract the code on line 2 and on lines 5–10 into separate base code modules, as two different implementations of debugging, and to make the configuration layer select the right

module based on the chosen mode (debugging or not). This conceptual simplicity has a number of drawbacks, however. First, control over the selection of definitions is moved completely to the build system, which means that developers need to know how this configuration is connected to the source code. They now have to develop code in different modules instead of using conditional compilation, and they need to be able to control how these modules interact with each other. In the case of Figure 10.1, there is only one conditional check. Second, conditional guards can in general be nested such that many different combinations of conditions are possible instead of only two. This not only means that developers should be able to understand these different cases [144, 204], but also that the number of modules may explode. Hence, extraction into separate modules is not always viable.

A second technique is to use AOP's ITD techniques to introduce definitions in the base code. The largest difference with the separate modules, is that pointcuts are used to select whether or not the weaver should perform the ITD, instead of making this decision dependent on selection of modules. Pointcuts should be enhanced with access to build configuration information, such that the weaver can directly exploit the knowledge of the current configuration to steer ITD. As explained in Section 2.4.5 on page 67, such a mechanism fosters the decoupling of the base code from configuration logic (RC4). As Aspicere does not provide ITD, we have not been able to implement this technique.

Note that the example of Figure 10.1 involves more than just ITD, as there are two alternatives for the tracing macro. One way to deal with this is to put the empty macro definition in the base code and let ITD override the macro definition. Alternatively, the macro could be replaced by a procedure such that `around`-advice can redefine the tracing implementation. Third, similar to the extracted modules, two aspects could be used to introduce the tracing, but this would suffer from the same state explosion issues as the module extraction technique. A more drastic approach is to make the base code completely oblivious to tracing and to let aspects decide how they implement tracing and where it should apply. Later, we will see that this is not easy because of the large variability in tracing invocation [32]. This explanation shows that even in the clean case of Conditional Definitions extraction, decoupling the base code from configuration logic is not trivial.

10.2.2 Class 2: Fine-grained Conditional Compilation

The second category of conditional compilation patterns we have identified considers fine-grained selection of behaviour or state implementations. With "fine-grained", we refer to intra-procedure and intra-type definition patterns. These patterns exploit the preprocessor's capabilities to its fullest, although almost all cases we have observed in the Parrot VM more or less obey to the C syntax rules. This

```
1  static INTVAL
2  ret_int(Interp *interp, const char *p, int type){
3    switch (type) {
4      case enum_type_INTVAL:
5       return *(const INTVAL*) p;
6  #if INT_SIZE == 4
7      case enum_type_int32:
8      case enum_type_uint32:
9  #endif
10 #if INT_SIZE == 8
11     case enum_type_int64:
12     case enum_type_uint64:
13 #endif
14 ...
15     case enum_type_uchar:
16      return *p;
17     default:
18      real_exception(interp, NULL, 1,
19                  "returning unhandled int type in struct");
20   }
21   return -1;
22 }
```

Figure 10.2: General Fine-grained Conditional Compilation in the implementation of the `ret_int` *procedure in src/pmc/unmanagedstruct.pmc.*

is in line with the findings of Ernst et al. [80], who have measured that on average two thirds of preprocessor usage in general is sufficiently disciplined.

For our purposes, i.e. determining whether the structure of the patterns is amenable to an aspect implementation, we have distilled five subcategories of Fine-grained Conditional Compilation:

1. General Case
2. Scattered Conditional Compilation
3. Simple Conditional Compilation
4. Simple Conditional Compilation with Dependencies
5. Simple Conditional Compilation with Declarations

We consider each of these subcategories in turn in the following subsections.

10.2.2.1 The General Case

In general, the preprocessor facilitates fine-grained (de)selection of code lines, statements or even parts of tokens based on whether certain preprocessor constants

```
1 static void compact_pool(PARROT_INTERP,NOTNULL(Memory_Pool
2                                                 *pool)){
3   INTVAL        j;
4   UINTVAL       object_size;
5   UINTVAL       total_size;
6   ...
7   /* ever increasing allocations but fewer collect runs */
8 #if WE_WANT_EVER_GROWING_ALLOCATIONS
9   total_size += pool->minimum_block_size;
10 #endif
11
12  /* Snag a block big enough for everything */
13  new_block = (Memory_Block *)alloc_new_block(interp,
14                total_size, pool, "inside compact");
15  ...
16 }
```

Figure 10.3: General Fine-grained Conditional Compilation in the implementation of the `compact_pool` *procedure in src/malloc.c.*

have been defined and/or the specific value they have. Figure 10.2 gives a common example of such Fine-grained Conditional Compilation. On lines 6–9 and 10–13, the inclusion of specific `case`-entries of a conditional `switch` structure depends on the specific value of the `INT_SIZE` preprocessor constant. A module-based approach is too coarse-grained to deal with this[2], whereas there are no sufficiently fine-grained join points and pointcuts to use with conventional ITD or advice.

Figure 10.3 gives a second example of Fine-grained Conditional Compilation. On lines 8–10 an assignation is conditionally guarded to let developers control the strategy used for managing the pool of objects in memory. This case seems easier to deal with than Figure 10.2, but still there is no straightforward way to extract this single conditional block into advice. If assignments would be valid join points, succinct selection of a particular assignment join point would heavily depend on implementation details and hence would not be maintainable. Extracting the assignment into a procedure and managing the variability of this procedure is a feasible technique. However, in most cases multiple conditional blocks like the one on Figure 10.3 (often with a different condition) occur in a given procedure body. Hence, this could lead to a proliferation of short procedures with a dedicated, i.e. not reusable, advice. Just as for Conditional Definitions, there is no unique way to decouple the base code from the General Case of Fine-grained Conditional Compilation. Later subsections consider special cases of Fine-grained Conditional Compilation which offer better opportunities for extraction.

[2]Unless one tries to maintain multiple versions of the `compact_pool` in parallel.

```
1  void Parrot_STM_waitlist_wait(Parrot_Interp interp) {
2    struct waitlist_thread_data *thr;
3    thr = get_thread(interp);
4    LOCK(thr->signal_mutex);
5  #if WAITLIST_DEBUG
6    fprintf(stderr, "%p: got lock, waiting...\n", interp);
7  #endif
8    while (!thr->signaled_p) {
9      pt_thread_wait_with(interp, &thr->signal_mutex);
10 #if WAITLIST_DEBUG
11     fprintf(stderr, "%p: woke up\n", interp);
12 #endif
13   }
14   UNLOCK(thr->signal_mutex);
15 #if WAITLIST_DEBUG
16   fprintf(stderr, "%p: done waiting.\n", interp);
17 #endif
18 }
```

Figure 10.4: Scattered Conditional Compilation in the implementation of the `compact_Parrot_STM_waitlist_wait` *procedure in src/stm/waitlist.c.*

As a side note, there are many conditional blocks with a constant condition, e.g. `#if 0` or `#if 1`. This typically corresponds to temporary experimental or buggy code, and can be easily understood because of the simple condition. Hence, extraction into a different module is overkill in this case.

10.2.2.2 Scattered Conditional Compilation

Scattered Conditional Compilation is a first special case of Fine-grained Conditional Compilation. One procedure contains multiple conditional blocks, each of which uses the same condition and guards highly similar code. In Figure 10.4, the blocks on lines 5–7, 10–12 and 15–17 all depend on the definedness of `WAIT_LIST_DEBUG` and print out a debugging message to the error stream. These blocks are not only tangled with the main logic, they are also scattered across multiple places in the `compact_Parrot_STM_waitlist_wait` and elsewhere. Hence, conventional aspect technology should be able to deal with this.

Figure 10.5 gives a possible aspect implementation (lines 18–22). To deal with the variable advice bodies, annotations are used in the base code (lines 6, 9 and 13) to specify the appropriate debugging messages, similar to the approach in Chapter 8. In general, not all Scattered Conditional Compilation instances guard similar uniform code as Figure 10.4 depicts. Second, the required context of the conditional block (in this case `interp`) cannot be reused anywhere `WAIT_LIST`

```
1 /*base code*/
2 void Parrot_STM_waitlist_wait(Parrot_Interp interp) {
3   struct waitlist_thread_data *thr;
4   thr = get_thread(interp);
5
6   /*@waitlist_debug("%p: got lock, waiting...\n",interp)*/
7   LOCK(thr->signal_mutex);
8   while (!thr->signaled_p) {
9     /*@waitlist_debug("%p: woke up\n",interp)*/
10    pt_thread_wait_with(interp, &thr->signal_mutex);
11  }
12
13  /*@waitlist_debug("%p: done waiting.\n",interp)*/
14  UNLOCK(thr->signal_mutex);
15 }
16
17 /*aspect*/
18 void debug(char* String,Parrot_Interp Interp) after Jp:
19  invocation(Jp,_)
20  && annotation(Jp,waitlist_debug,[String,Interp]){
21   fprintf(stderr, String, Interp);
22 }
```

Figure 10.5: Aspect implementation of the Scattered Conditional Compilation example of Figure 10.4.

conditional blocks occur. Hence, the variability of conditional blocks is an important factor to take into account when deciding on a particular aspect language to extract Scattered Conditional Compilation into an aspect.

Third, we have assumed that `LOCK` (line 4 of Figure 10.4) and `UNLOCK` (line 14) are procedure calls. However, these actually correspond to macro expansions. Accidentally, this might align with a procedure call, but in general they can expand into data accesses, multiple statements, etc. Hence, the aspect language should support these constructs as possible join points, unless macros are extracted into procedures (as discussed in Section 10.2.1).

Finally, we should consider the efficiency of the advice in comparison with that of the conditional code it replaces. As a preprocessor generates preprocessor-less C code, this means that all configuration choices have been hardcoded in the resulting source code file. Hence, no run-time checks are needed anymore to deal with this, nor should the compiler know about the configuration decisions. In the case of the advice on Figure 10.5, the efficiency benefits seem to have vanished, as conceptually the advice is implicitly called at the advised join points. However, as discussed before (in Section 5.4.2 and Section 8.2.5.2), static weavers like Aspicere2 incorporate various optimisations like inlining or constant propagation

```
1 PMC* pt_transfer_sub(Parrot_Interp d, Parrot_Interp s,
2                       PMC *sub){
3 #if THREAD_DEBUG
4   PIO_eprintf(s, "copying over subroutine [%Ss]\n",
5               Parrot_full_sub_name(s, sub));
6 #endif
7   return make_local_copy(d, s, sub);
8 }
```

Figure 10.6: Simple Conditional Compilation inside src/thread.c for the implementation of the `pt_transfer_sub` procedure.

```
1 static void add_entry(STM_waitlist *waitlist,
2                        struct waitlist_entry *entry) {
3   int successp = -1;
4   assert(entry->next == NULL);
5   do {
6     PARROT_ATOMIC_PTR_GET(entry->next, waitlist->first);
7     assert(successp != -1 || entry->next != entry);
8     assert(entry->next != entry);
9     PARROT_ATOMIC_PTR_CAS(successp, waitlist->first,
10                          entry->next, entry);
11  } while (!successp);
12 #if WAITLIST_DEBUG
13  fprintf(stderr, "added %p(%p) to waitlist %p\n",
14          entry, entry->thread->interp, waitlist);
15 #endif
16 }
```

Figure 10.7: Simple Conditional Compilation inside src/stm/waitlist.c for the implementation of the `add_entry` procedure.

to remove redundant dynamic checks. In the case of Figure 10.5, this means that the resulting woven code is equivalent to the preprocessor approach, as the annotations statically determine the relevant join point shadows and inlining can be applied. No additional function calls or checks are needed. Hence, if advice is able to implement the conditionally guarded logic, the consequences on run-time efficiency are not necessarily worse than for preprocessed code. However, the use of a weaver incurs a higher compilation cost (more on this later).

10.2.2.3 Simple Conditional Compilation

This section considers a second special case of Fine-grained Conditional Composition, i.e. the appearance of one conditional block at the beginning or end of a

```
1 /*base code*/
2 PMC* pt_transfer_sub(Parrot_Interp d, Parrot_Interp s,
3                       PMC *sub){
4   return make_local_copy(d, s, sub);
5 }
6
7 /*aspect*/
8 void debug_transfer(Parrot_Interp S, PMC* Sub) before Jp:
9  execution(Jp,``pt_transfer_sub'')
10  && args(Jp,[_,S,Sub])
11  && thread_debug(_){
12   PIO_eprintf(S, "copying over subroutine [%Ss]\n",
13               Parrot_full_sub_name(S, Sub));
14 }
```

Figure 10.8: Aspect implementation of the Simple Conditional Compilation example of Figure 10.6.

procedure. Figure 10.6 and Figure 10.7 give examples of these two cases respectively. Note that the latter shows another manifestation of the scattered conditional block of the previous subsection.

Simple Conditional Compilation can be implemented via normal `before`- or `after`-advice on the base code procedures, in which the conditional code has been completely removed. Figure 10.8 shows such an aspect for the example of Figure 10.6. A similar approach can be used for Figure 10.7. The main observation from these aspects, is that the pointcut is expressed in terms of build configuration information (`thread_debug` on line 11). The configuration layer should communicate this information to the weaver's database of meta data (in Aspicere: the logic fact base) before join point matching can start. Hence, the build system is coupled to the aspects instead of to the base code. Again, optimisations by the weaver can eliminate redundant dynamic checks.

An alternative approach is the use of annotations, similar to the Scattered Conditional Compilation (Figure 10.5). The disadvantage of this is that the base code still contains configuration-like constructs, but on the upside base code developers do not need to write aspects and can more easily integrate the desired logic with the base code. Hence, a trade-off should be made between code readability and development responsibilities.

10.2.2.4 Simple Conditional Compilation with Dependencies

A special case of Simple Conditional Compilation is depicted on Figure 10.9. Here, there is still one conditional block at the end of a procedure, but it nests multiple levels of conditional compilation and contains the procedure's return state-

```
1  static opcode_t fetch_op_be_4(NOTNULL(unsigned char *b)){
2    union {
3      unsigned char buf[4];
4      opcode_t o;
5    } u;
6    fetch_buf_be_4(u.buf, b);
7
8  #if PARROT_BIGENDIAN
9  #  if OPCODE_T_SIZE == 8
10   return u.o >> 32;
11 #  else
12   return u.o;
13 #  endif
14 #else
15 #  if OPCODE_T_SIZE == 8
16   return (opcode_t)(fetch_iv_le((INTVAL)u.o) & 0xffffffff);
17 #  else
18   return (opcode_t) fetch_iv_le((INTVAL)u.o);
19 #  endif
20 #endif
21 }
```

Figure 10.9: Simple Conditional Compilation with Dependencies in the implementation of the `fetch_op_be_4` procedure in src/packfile/pf_items.c.

ment. Based on the byte order (line 8) and the size of opcodes (lines 9 and 15), the return value of the `fetch_op_be_4` differs. There are four potential cases, but in general nesting can lead to many more combinations.

Figure 10.10 shows a possible way to deal with this. One particular configuration is chosen as the base configuration (`PARROT_BIGENDIAN` defined and `OPCODE_T_SIZE` different from 8) in the body of `fetch_op_be_4`. The two advices on lines 13–30 encode the other configurations as C conditions and checks based on the chosen configuration (lines 16–17 and 24–25). Aspicere2 implements advice on `execution` join points as a call to a procedure which contains the advice body. Because the value for `Size` is constant, inlining and further optimisation techniques eliminate the call and the conditional checks. Eventually, the woven code is equivalent to the code generated by the preprocessor. Normal C development environments help developers in understanding the different control flows, i.e. the possible configurations, in the extracted advice bodies. Annotations are less desirable, because multiple ones are needed to control the choice of the right return value.

```
1  /*base code*/
2  static opcode_t fetch_op_be_4(NOTNULL(unsigned char *b)){
3    union {
4      unsigned char buf[4];
5      opcode_t o;
6    } u;
7    fetch_buf_be_4(u.buf, b);
8
9    return u.o;
10 }
11
12 /*aspect*/
13 opcode_t big_endian(int Size) after Jp
14                               returning (opcode_t* Op):
15  execution(Jp,``fetch_op_be_4'')
16  && parrot_bigendian(_)
17  && opcode_t_size(Size){
18    if(Size==8) *Op=*Op >> 32;
19 }
20
21 opcode_t little_endian(int Size) after Jp
22                                  returning (opcode_t* Op):
23  execution(Jp,``fetch_op_be_4'')
24  && !!parrot_bigendian(_)
25  && opcode_t_size(Size){
26    if(Size==8){
27      *Op=(opcode_t)(fetch_iv_le((INTVAL)(*Op))
28                     & 0xffffffff);
29    }else *Op=(opcode_t)fetch_iv_le((INTVAL)(*Op));
30 }
```

Figure 10.10: Aspect implementation of the Simple Conditional Compilation with Dependencies example of Figure 10.9.

10.2.2.5 Simple Conditional Compilation with Declarations

A second special case of Simple Conditional Compilation, and the last subcategory of Fine-grained Conditional Compilation, is Simple Conditional Compilation with Declarations. Figure 10.11 illustrates this. Basically, this corresponds to Simple Conditional Compilation at the end of a procedure with dependencies on local variables which have been declared inside a conditional block at the beginning of the procedure. This can actually be implemented in a similar way as Figure 10.10, because the declared variable is only used in the advice. Figure 10.12 illustrates this. Note that the check on line 19 incurs overhead compared to the original code in Figure 10.11, as the return value of the procedures has to be dynamically checked

```
1 void* parrot_pic_opcode(PARROT_INTERP, INTVAL op){
2   const int core = interp->run_core;
3 #ifdef HAVE_COMPUTED_GOTO
4   op_lib_t *cg_lib;
5 #endif
6
7   if(core == PARROT_SWITCH_CORE ||
8      core == PARROT_SWITCH_JIT_CORE)
9       return (void*) op;
10 #ifdef HAVE_COMPUTED_GOTO
11  cg_lib = PARROT_CORE_CGP_OPLIB_INIT(1);
12  return ((void**)cg_lib->op_func_table)[op];
13 #else
14  return NULL;
15 #endif
16 }
```

Figure 10.11: Simple Conditional Compilation with Declarations in the implementation of the `parrot_pic_opcode` procedure in src/pic.c.

for a null-pointer. This is the only way to distinguish between the return statements on lines 7 and 9 of Figure 10.12, provided that the incoming `Op` argument of `parrot_pic_opcode` is not `NULL`.

10.2.3 Class 3: Coarse-grained Conditional Compilation

Next to Conditional Definitions and Fine-grained Conditional Compilation, the third main category of conditional compilation in the Parrot VM is Coarse-grained Conditional Compilation. In essence, this is similar to Fine-grained Conditional Compilation, but instead of controlling parts of a procedure definition, the whole body (not the definition) is conditional. We consider two subcategories, i.e. Partitioned Conditional Compilation and Semi-partitioned Conditional Compilation.

10.2.3.1 Partitioned Conditional Compilation

Partitioned Conditional Compilation encloses the whole procedure body. Typically (Figure 10.13), there are two major implementations for a procedure body (line 5 and lines 7–22). In this case, there is nesting of conditional compilation, as the second branch of the conditional block has two possible implementations as well (lines 8–9 and 11–21).

Because the conditional logic fills the whole procedure body, extracting the different configurations into separate modules and deciding at build-time which module to use, seems a feasible approach. As mentioned for Conditional Defini-

```
1 /*base code*/
2 void* parrot_pic_opcode(PARROT_INTERP, INTVAL op){
3   const int core = interp->run_core;
4
5   if(core == PARROT_SWITCH_CORE ||
6      core == PARROT_SWITCH_JIT_CORE)
7       return (void*) op;
8
9   return NULL;
10 }
11
12 /*aspect*/
13 void* computed_goto(INTVAL Op) after Jp
14                                returning (void** Tmp):
15  execution(Jp,``parrot_pic_opcode'')
16  && args(Jp,[_,Op])
17  && have_computed_goto(_){
18    op_lib_t* cg_lib;
19    if(!(*Tmp)){
20      cg_lib = PARROT_CORE_CGP_OPLIB_INIT(1);
21      *Tmp= ((void**)cg_lib->op_func_table)[Op];
22    }
23 }
```

Figure 10.12: Aspect implementation of the Simple Conditional Compilation with Declarations example of Figure 10.11.

tions, the number of modules may explode, however. Alternatively, one implementation can be chosen and the others can be implemented as `around`-advice which is triggered by configuration information, as we have seen earlier in this chapter. No run-time overhead is to be expected, as any configuration is uniquely determined by the value of build-time configuration parameters.

Note that `fetch_iv_le` has a dual form, i.e. `fetch_iv_be`, in which the conditional logic is negated by omitting the `!`-sign on line 4. Reuse of modules or advice depends on the ability to assign different configuration conditions to them in the configuration layer or the pointcut.

10.2.3.2 Semi-partitioned Conditional Compilation

The second incarnation of Coarse-grained Conditional Compilation, i.e. Semi-partitioned Conditional Compilation, is similar to Partitioned Conditional Compilation, except that the procedure's return statement is not conditionally guarded. This is illustrated on Figure 10.14 (line 22). In principle, Semi-partitioned Conditional Compilation does not require a drastically different approach from Par-

```
1  PARROT_WARN_UNUSED_RESULT
2  PARROT_CONST_FUNCTION
3  INTVAL fetch_iv_le(INTVAL w){
4  #if !PARROT_BIGENDIAN
5    return w;
6  #else
7  #  if INTVAL_SIZE == 4
8    return (w << 24) | ((w & 0xff00) << 8) |
9           ((w & 0xff0000) >> 8) | (w >> 24);
10 #  else
11   INTVAL r;
12
13   r = w << 56;
14   r |= (w & 0xff00) << 40;
15   r |= (w & 0xff0000) << 24;
16   r |= (w & 0xff000000) << 8;
17   r |= (w & 0xff00000000) >> 8;
18   r |= (w & 0xff0000000000) >> 24;
19   r |= (w & 0xff000000000000) >> 40;
20   r |= (w & 0xff00000000000000) >> 56;
21   return r;
22 #  endif
23 #endif
24 }
```

Figure 10.13: Partitioned Conditional Compilation of the implementation of the `fetch_iv_le` procedure in src/byteorder.c.

titioned Conditional Compilation. The return statement can be distributed over the conditional blocks to enable the same solutions as for Partitioned Conditional Compilation.

This concludes the categorisation of conditional compilation patterns in the Parrot VM. The next section discusses our findings.

10.2.4 Validation #1: Aspicere Meets Goal L1

The classification of conditional compilation patterns has taught us a number of things. First, separation of different variants into modules can in some cases avoid the use of aspects. However, it requires tight control over the configuration layer, and often results in explosion of modules because of nested conditional compilation.

Second, aspects prevent this proliferation of modules by encoding configuration logic inside the pointcut (Simple Conditional Compilation) or inside the advice body (Simple Conditional Compilation with Dependencies/Declarations). In

```
1  PARROT_API
2  opcode_t *
3  Parrot_sleep_on_event(PARROT_INTERP, FLOATVAL t,
4                        opcode_t *next)
5  {
6  #if PARROT_HAS_THREADS
7    if (interp->sleeping)
8        fprintf(stderr, "nested sleep might not work\n");
9    /*
10    * place the opcode_t* next arg in the event data, so
11    * that we can identify this event in wakeup
12    */
13   Parrot_new_timer_event(interp, (PMC *) next, t,
14           0, 0, NULL, EVENT_TYPE_SLEEP);
15   next = wait_for_wakeup(interp, next);
16 #else
17   /*
18    * TODO check for nanosleep or such
19    */
20   Parrot_sleep((UINTVAL) ceil(t));
21 #endif
22   return next;
23 }
```

Figure 10.14: Semi-partitioned Conditional Compilation of the implementation of the `Parrot_sleep_on_event` procedure in src/events.c.

the latter case, redundant dynamic checks can mostly be eliminated by the weaver. However, the General Case of Fine-grained Conditional Composition often cannot be modeled as robust advice, because it needs very fine-grained join points like assignments and macro expansions, which potentially couple the advice too closely with implementation details. This is an important problem to solve. A second important issue is advice variability, as mentioned for Scattered Conditional Compilation. To avoid multiple copies of the same advice with only minor differences [33], advice should be robust to variability and accessible context.

Third, except for the Scattered Conditional Compilation example given in Figure 10.4, the advices we have shown exploit build configuration information in their pointcut to decide whether the advice should match or not. This communication channel between the build system and the weaver is crucial to decouple the base code from configuration logic. This concept has been suggested in the context of RC4 in Section 2.4.5 on page 67, and it actually is an extension of the use of build system information in pointcuts as used in the Quake 3 and ASML case studies. In the case of preprocessor-driven systems, it is the key technique to couple configuration decisions to advice instead of to the base code.

To summarise, Aspicere is able to extract some, but not all conditionally guarded base code into advice. It needs finer-grained join points to be able to implement more cases as advice, but in that case the advice could become too dependent on and coupled with the base code. Hence, this should be investigated in more depth. The next section considers the impact on the build system of the application of the presented aspects in the Parrot VM.

10.3 Impact on the Build System

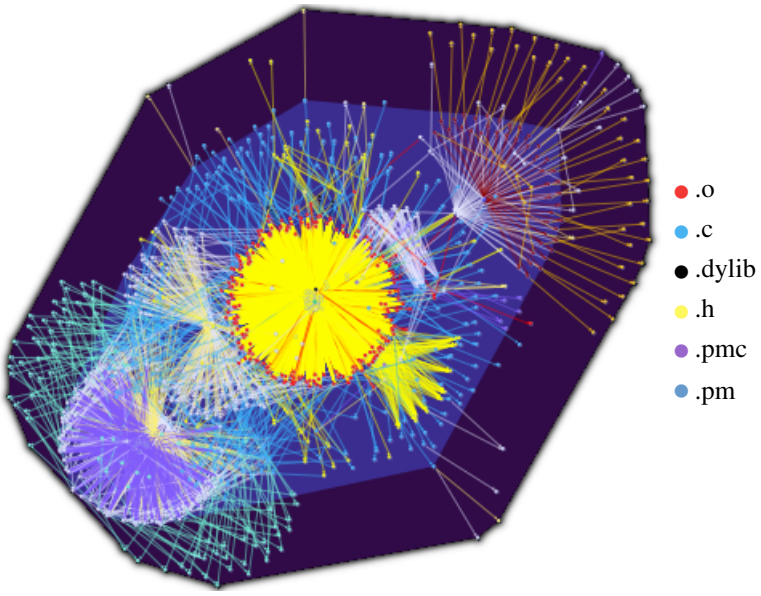
This section discusses the impact on the build system of the introduction of aspects in the Parrot VM. We consider the following four build problems:

1. physical integration of Aspicere2 with the build process
2. migration to the re-engineered system
3. build time increase and incremental weaving
4. communication between Aspicere2 and the build system

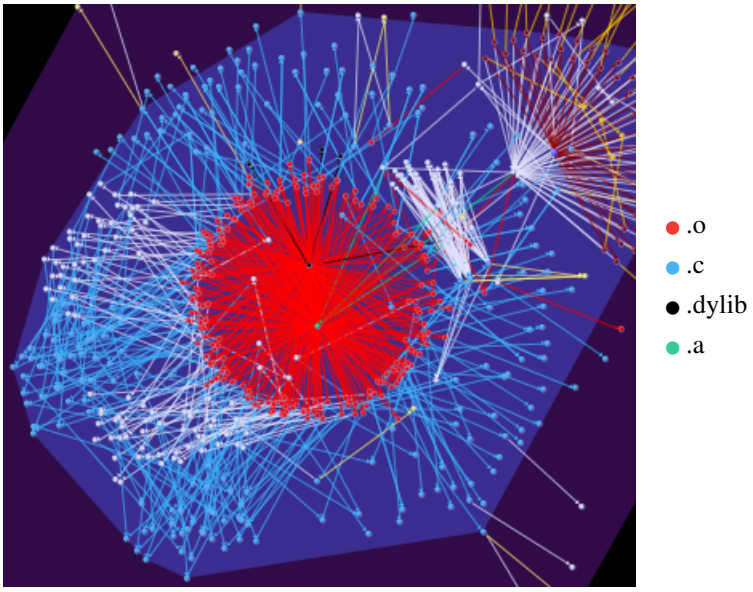
10.3.1 Integration of Aspicere2 with the Build Process

Just as in the Quake 3 and CSOM cases, we have first generated the build dependency graph of a complete build of the Parrot VM. Figure 10.15a shows the resulting build DAG. There is a yellow cluster in the middle (header files), with a red border (object files) and blue satellite nodes (C files). In the lower left half of the graph, .pmc and .pm files occur. As .pmc files are transformed into .c files (as mentioned earlier), many .c files depend on .pmc files. On the upper right, runtime support files of the Parrot VM are constructed. To summarise, we only need to take into account the central cluster.

Figure 10.15b shows a filtered version of Figure 10.15a, in which the lower left targets have been hidden and all header files as well. What is left is a red cluster in the middle with .c files around it. The object files are actually linked into two libraries, i.e. a static one (green node) and a dynamic one (black node). The static one is always built, whereas the dynamic library is only constructed if the build engineer has configured the build like this. Hence, the build architecture of the Parrot VM is very simple. The actual VM implementation is built into a library, and this is linked with a main file and a Perl library into the VM executable ("parrot"). There is also a "miniparrot" executable, which is a dependee of "parrot". This is used for bootstrapping the build of "parrot" (cf. compilers), and also depends on the static or dynamic library. To summarise, we can consider the libraries as "whole program" for the aspects.



(a)



(b)

Figure 10.15: Build DAG of Parrot 0.4.14, (a) in full and (b) after hiding .pm, .pmc, .dump, .pl and header files. A dynamic (black) and static (green) library are built from all object files at once.

```
1 CC          = @cc@
2 CFLAGS      = $(CC_INC) @ccflags@ @cc_debug@ @ccwarn@ \
3               @cc_hasjit@ @cg_flag@ @gc_flag@ $(CC_SHARED)
4 ...
5 LD          = @ld@
6 ...
7 LINK        = @ld@
8 ...
9 $(LIBPARROT_SHARED): $(O_FILES)
10    $(MKPATH) @blib_dir@
11    $(LD) $(LD_SHARE_FLAGS) $(LDFLAGS) \
12          @ld_out@$@ @libparrot_soname@ \
13          $(O_FILES) $(C_LIBS) $(ICU_SHARED)
14 #CONDITIONED_LINE(libparrot_shared_alias): \
15          ( cd @blib_dir@ ; ln -sf \
16            @libparrot_shared@ @libparrot_shared_alias@ )
```

Figure 10.16: Original build commands and rules in the Parrot build script template (config/gen/makefiles/root.in).

We have integrated Aspicere2 with the construction of the dynamic library. The approach for doing this is similar to previous cases, except that Parrot uses a custom build layer generation and configuration layer. Figure 10.16 shows the relevant parts of the "root.in" build script template. Similar to Quake 3, build variables are consistently used for abstracting the name of the C compiler (line 1) and linker (lines 5 and 7). As shown in Figure 10.17, these variables could easily be overridden by the name of the LLVM front end, and extra variables were added on lines 3–7 for the Aspicere2 weaver scripts. The library construction rule of lines 9–16 on Figure 10.16 is split in two on lines 9–21 of Figure 10.17. The extra compilation flags for the LLVM front end (`-fno-builtin -g -c -emit-llvm`) had to be provided during configuration of the build layer, together with the decision to build a dynamic library.

To conclude, visualisation of the build architecture, and the querying and filtering features of MAKAO have been able to support us in integrating Aspicere2 in the Parrot VM build.

10.3.2 Migration to the Re-engineered System

Decoupling the base code from conditional logic entails removing the conditional preprocessor constructs from the source code. Depending on the specific category of conditional compilation (Section 10.2), conditional blocks should be implemented in a particular way. Each technique has different consequences for the conditional logic: extraction to advice body, extraction of different variants to dif-

```
1 CC      = llvm-gcc
2 ...
3 FLAGS   = @ccflags@ @cc_hasjit@ @cg_flag@ @gc_flag@
4 LDD     = lto.sh
5 LDDD    = link.sh -aspects /path/to/aspects/aspects.lst \
6                   -modules /path/to/aspects/modules.lst
7 ASPECTS =`paste -s -d\  /path/to/aspects/aspects.lst`
8 ...
9 $(LIBPARROT_SHARED) : $(LIBPARROT_SHARED).bc
10    $(LDD) $(LD_SHARE_FLAGS) $(LDFLAGS) \
11           @ld_out@$@ @libparrot_soname@ \
12           $(LIBPARROT_SHARED).bc $(C_LIBS) $(ICU_SHARED)
13 #CONDITIONED_LINE(libparrot_shared_alias): \
14          ( cd @blib_dir@ ; ln -sf \
15            @libparrot_shared@ @libparrot_shared_alias@ )
16 $(ASPECTS): ;
17 $(LIBPARROT_SHARED).bc : $(O_FILES) $(ASPECTS)
18    $(MKPATH) blib/lib
19    $(LDDD) $(FLAGS) $(LD_SHARE_FLAGS) $(LDFLAGS) \
20            @ld_out@$@ @libparrot_soname@ \
21            $(O_FILES) $(C_LIBS) $(ICU_SHARED)
```

Figure 10.17: Modified build commands and rules in the Parrot build script template (config/gen/makefiles/root.in).

ferent modules, etc. Automatic extraction is not straightforward because of the different trade-offs between modularity and manageability, speed and modularity, etc. Hence, a one-time migration from a tangled preprocessor-driven implementation to an aspect-based one is not feasible.

In the ASML case, stepwise migration from the initial system to an aspect-based implementation required support in the build system to support the co-existence of old and new components. However, in the Parrot case this is not really a problem. If aspects are applied to the whole dynamic library at once and compilation flags are still passed to the compiler, preprocessor-based conditional compilation can co-exist with a (partial) aspect implementation. We have been able to conclude this from knowledge of the build architecture.

10.3.3 Build Time Increase and Incremental Weaving

Because all source code files end up in the same dynamic library, the linking takes relatively long. We have no exact measurements of the weaving time, but as observed in earlier case studies, incremental weaving is a necessity to not reduce the developers' productivity. Splitting up the build rules and adding the aspects as dependee (lines 9–21 of Figure 10.17) is only a partial solution. The Aspicere2

link-time weaver lacks internal incremental weaving support to succinctly resolve this problem completely.

10.3.4 Communication between Aspicere2 and the Build System

Two mechanisms are needed to support extraction of conditional compilation logic. The first one deals with extraction of conditional logic into source code modules. We have not experimented with this, but (de)selection of modules requires conditional build logic in the build script template. By either setting environment variables or by using variables defined during configuration, the right collection of modules can be selected.

The second mechanism is the exchange of build configuration decisions to the weaver. By querying the build dependency graph in Figure 10.15b, we have observed that preprocessor constants are defined in the `CFLAGS` variable on lines 2–3 of Figure 10.16. Only a couple of the configuration variables used in the definition of `CFLAGS` contain preprocessor flags. Hence, we have defined a new variable (`FLAGS` on line 3 of Figure 10.17) with only these variables. By passing this variable to the build rule which links all base modules together in a link module (line 19 of Figure 10.17), the "link.sh" script can deduce the values of active preprocessor flags and assert these flags as facts during compilation of the Prolog fact base. Existing modules which still contain preprocessor constructs are not broken (the flags are still passed to the compiler), whereas the aspects have direct access to the chosen configuration options.

Note that preprocessor constants are not only passed via compiler flags, but also (similar to autoconf in Section 2.1.3.1) via macro definitions in header files, i.e. parrot/config.h and parrot/feature.h. To deal with this, we have used an ad hoc Perl script for extracting such definitions during the execution of the "link.sh" script. The results of this script are asserted afterwards in the Prolog fact base. We have not taken into account other header or implementation files which could define macro constants.

10.4 Validation #2: Roots of Co-evolution Experimentally Confirmed

The case study in this chapter provides a fifth and last experimental validation of the roots of co-evolution in a (legacy[3]) system in which AOP has been introduced. We especially have encountered problems regarding RC3 and RC4.

[3] As mentioned before, the Parrot VM is not a legacy system, but its preprocessor usage is representative for legacy systems.

RC1 and RC2 have not caused actual trouble because the Parrot VM has a relatively simple build system and because MAKAO has enabled us to understand its architecture. Incremental weaving (RC3) is, just as in earlier cases, a problem which we have only been able to deal with partially. Internal support is needed in the weaver. RC4 has required us to communicate configuration information to the aspect weaver, or alternatively to select the right base modules if conditional logic is not extracted into advice. Knowledge of the build internals, especially the values of build variables, has enabled us to integrate Aspicere2 with the build system.

10.5 Validation #3: MAKAO Achieves Goal T2

The knowledge obtained via MAKAO's visualisation, querying and filtering features have enabled us to integrate Aspicere2 in the build system via manual modification of the build script template. Visualisation has taught us how the Parrot VM build architecture works, assisted by filtering to reduce detail. Querying has allowed us to query for the values of build variables.

10.6 Validation #4: Aspicere Meets Goal L2

Aspicere's logic-based pointcut language has shown to be a perfect mechanism to integrate build system data, in this case configuration choices, to the advices. Combined with generic advice, it is able to replace conditional compilation by aspects in various cases. However, as discussed in Section 10.2.4, Aspicere has some limitations to fully be able to implement conditional logic as advice.

10.7 Conclusion

In this chapter, we have explored how aspects can be used to extract conditional compilation from the base code. This decouples the base code from the build system. We have categorised the different patterns of conditional compilation usage in the Parrot VM to identify aspect language challenges, and have described which build problems did occur by introducing AOP into Parrot. Especially RC3 and RC4 have been considered. MAKAO has been used to understand the build architecture (goal T2), whereas Aspicere's integration with the build system has enabled configuration of advice by the build system (goal L2).

This chapter has described the last case study of this dissertation. A summary of the validation results of each case study is given in the next chapter, which presents the conclusions of this dissertation.

The End is the Beginning is the End...
The Smashing Pumpkins

11 Conclusions and Future Work

THIS chapter presents the conclusions of this dissertation and directions for future work. First (Section 11.1), we recapitulate the six research questions we have addressed in the introduction. A detailed account on conceptual contributions (Section 11.2.1) summarises how we have addressed these research questions. Afterwards, technical contributions (Section 11.2.2) are discussed as well as various directions for future work (Section 11.3). Section 11.4 presents our final conclusions.

11.1 Problem Statement

Legacy systems are old, mission-critical software systems which are still plagued by constantly changing requirements [21, 58]. To cope with these requirements, legacy systems have to be re-engineered, because re-implementation from scratch is economically not feasible. Unfortunately, re-engineering is hampered by a lack of knowledge of the system's internals and the fear of changing critical functionality. AOSD [134] has been proposed as a means to overcome these two problems [51, 103, 142, 202, 170]. We have conjectured that the introduction of AOP technology in a legacy system is hampered by the phenomenon of co-evolution of source code and the build system. This forms the core topic of this dissertation.

Co-evolution is not a new phenomenon in software development, as various researchers have shown how different software models and different levels of describing and thinking about software can be causally connected to one another [243,

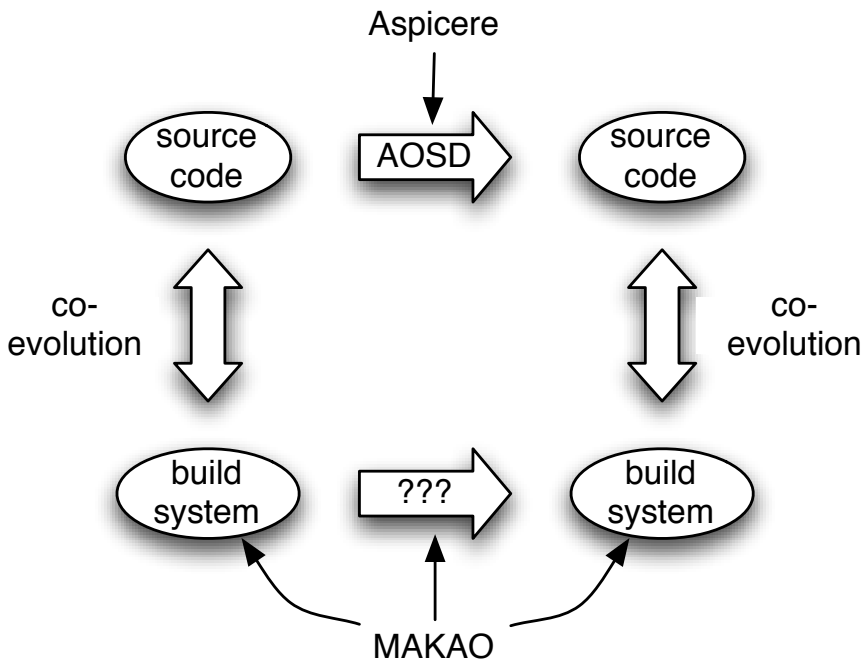


Figure 11.1: High-level overview of co-evolution of source code and the build system.

232, 85, 128, 172]. Co-evolution has been generalised by Favre [85] into a potential phenomenon between any two (or more) software artifacts. If one end of the co-evolution changes, the other side has to follow to enforce consistency between the two artifacts.

This dissertation has focused on co-evolution of source code and the build system, for which initial indications and early experimental evidence has been described in the literature [57, 58]. The primary focus has been to determine what kind of tools can manage the problems caused by co-evolution of source code and the build system. The design and scope of such tools depends on a crisp notion of what co-evolution of source code and the build system actually means. However, these fundamental reasons for co-evolution had not been investigated before. Based on a conceptual explanation of co-evolution of source code and the build system, tool support has been designed. This has been used to collect empirical evidence in legacy systems to validate the conceptual explanation of co-evolution of source code and the build system, and the ability of tool support to assist in understanding and managing the co-evolution.

The consequences of co-evolution of source code and the build system become particularly visible in the context of the introduction of AOP technology in legacy

systems, either when aspects are applied to reverse-engineer the base system or to re-engineer it. AOSD brings with it a major paradigm shift in the source code, i.e. one end of the co-evolution relation undergoes a drastic change. Because of co-evolution of source code and the build problem, major changes are necessary in the build system to keep both ends of the co-evolution consistent with each other. This is depicted in Figure 11.1 by means of the question marks.

Unfortunately, legacy systems have a legacy build system, of which the behaviour is not well-understood anymore. Because this lack of understanding precludes making changes to the build system, the build system is bound to become unsynchronised w.r.t. the source code. The uncertainty about the build system's behaviour and structure leads to compromises in the integration of AOP technology with the build system, e.g. because the notion of "whole program" is not clearly defined or the configuration of aspects cannot be sufficiently controlled. These compromises put restrictions on the potential of AOSD technology in the source code.

This dissertation has examined the conceptual explanation of the problems related to integration of AOP technology in the build system, by linking these problems to the conventional co-evolution of source code and the build system. This conceptual explanation has sparked ideas for adaptation of AOSD technology for dealing with co-evolution of source code and the build system. Experimental evidence has been collected to validate the existence of problems caused by co-evolution of source code and the build system when AOP technology is introduced in the source code. This evidence has also evaluated the ability of AOSD technology to deal with co-evolution.

To summarise, these are the six research questions addressed in this dissertation:

1. What are the fundamental reasons for co-evolution of source code and the build system?

2. What kind of tools do we need to understand and manage the co-evolution phenomena of source code and the build system in legacy systems?

3. Can we use our tools to confirm the conceptual reasons experimentally by applying them to third-party legacy systems?

4. How does the introduction of AOSD add to the fundamental reasons for co-evolution of source code and the build system?

5. How do we design AOSD technology which adequately deals with the co-evolution when applied to legacy systems?

6. Can we validate this technology by using it to manage co-evolution phenomena in existing legacy systems?

The next section considers how the contributions of this dissertation have addressed these six research questions.

11.2 Contributions

This section summarises the conceptual (Section 11.2.1) and technical (Section 11.2.2) contributions made in this dissertation. The conceptual contributions each address one research (sub-)question.

11.2.1 Conceptual Contributions

We cover the aforementioned research questions one by one in order to discuss the contributions made in this dissertation to address these questions.

11.2.1.1 What is Co-evolution of Source Code and the Build System?

Applying Favre's [85] taxonomy, co-evolution of source code and the build system corresponds to "asynchronous evolution of source code and the build system during which changes on one artifact have a vertical impact on the other one and vice versa". Corrective actions are required to enforce consistency between the two artifacts.

To be able to understand, observe and act upon co-evolution of source code and the build system, four roots of co-evolution have been postulated:

RC1 Modular reasoning in programming languages and compilation units in build systems are strongly related.

RC2 Build dependencies are used to reify the architectural structure of the source code.

RC3 Architectural interfaces in the source code are not always respected by incremental build processes.

RC4 Build system configuration layers are used as a poor man's support for product line-styled variability of source code.

Each of these roots has been distilled from research on build systems and programming languages, and represents a partial explanation for co-evolution. The four roots of co-evolution form the basis for understanding the co-evolution of source code and the build system, and for tool and aspect language support to deal with it.

	RC1	RC2	RC3	RC4
Linux kernel	V_iQF	V_iQF	V_iQF	
Kava	V_iQFRV_e	V_iQFV_e		
Quake 3	V_iQI	V_iQ		
ASML	V_iQFV_eR	V_iQFV_eR	V_iQFV_eR	I
CSOM	V_iQ	V_iQ		V_iQ
Parrot	V_iQ	V_iQ		V_iQFI

Table 11.1: Evaluation of the proposed tool (Visualisation, Querying, Filtering, Verification and Re-engineering) and language support (Integration of the build system with the aspect language) for understanding of and dealing with co-evolution problems which are explained by each root of co-evolution. Colored cells highlight build problems related to incremental weaving or the understanding of configuration layer which have not been solved by our tool and aspect language support. Empty cells indicate that a particular root of co-evolution is not applicable to a given case study.

11.2.1.2 Tool Support to Understand and Manage Co-evolution Phenomena?

Based on the conceptual explanation of co-evolution of source code and the build system (the four roots of co-evolution), we have proposed tool support to help with the understanding and management of build system problems in general (goal T1) and of the co-evolution of source code and the build system (goal T2). MAKAO is a re(verse)-engineering framework for build systems which is able to visualise, query, filter, verify and re-engineer build systems. The build system model on which MAKAO is based is the dependency graph of a concrete build run, enhanced with dynamic values of variables, static information of build scripts and derived knowledge about implicit dependencies.

We have validated MAKAO's ability to identify symptoms of co-evolution of source code and the build system on six case studies, as shown in Table 11.1. For a given case study, this table specifies for each root of co-evolution the tool and aspect language support which has helped in understanding and managing co-evolution of source code and the build system.

A first observation is that visualisation and querying of build systems have been useful for every case study. This is not surprising, as visualisation provides an intuitive overview of a build system, whereas querying allows direct access to static and dynamic build data. However, neither of these two tool features conveys explicit data about the build configuration. As indicated for the Linux case, MAKAO has not helped us to understand the custom configuration layer of the Linux kernel build system. This is an explicit design choice of MAKAO, but sys-

tems like the Linux kernel which are extremely configurable require configuration layer tool support for understanding and managing symptoms of co-evolution of source code and the build system.

Second, filtering of the build system has only been needed for large build systems with complex build recursion. Similarly, re-engineering of the build system has only been used (or expected to be used) for invasive, large build script changes. In the other case studies, the information obtained via visualisation and querying has sufficed to steer manual build changes. Verification is expected to be crucial for checking the validity of build changes when MAKAO's re-engineering is used, but in the Kava case this has not been necessary.

In general, the proposed tool support has proven to be capable of improving understanding of co-evolution of source code and the build system in most cases.

11.2.1.3 Experimental Evidence of the Roots of Co-evolution in Legacy Systems?

To validate the four roots of co-evolution, we have investigated the evolution of the Linux kernel build system from its inception until now. Co-evolution of source code and the build system forms a common thread throughout the evolution of the Linux kernel build system. There has been a continuous struggle to improve the "recursive make" build to facilitate the integration of new source code components (RC1) and the synchronisation of the build system dependencies with the source code architecture (RC2). Explicit evidence has been found about the perpetual trade-off between omitting source code dependencies in the build process to improve build speed and safeguarding build correctness (RC3). The configuration layer lives under extreme pressure to manage the abundant potential of source code configurability, which has clearly determined its evolution (RC4).

Hence, we have found concrete experimental evidence of co-evolution of source code and the build system, and this evidence can be attributed to the four roots of co-evolution. This validates the four roots of co-evolution.

11.2.1.4 What is the Relation between the Introduction of AOSD and Co-evolution?

By relating the introduction of AOP technology in a legacy system (horizontal arrow on top of Figure 11.1) to the four roots of co-evolution, we can infer that the build system also has to change to retain consistency between source code and the build system (horizontal arrow at the bottom of Figure 11.1).

Aspects require whole-program reasoning at the source code level [219, 162, 135] instead of traditional modular reasoning [187] (RC1). Inversion of dependencies [182, 160, 162], combined with fine-grained composition of aspects and the intensional selection of join points via pointcuts hamper synchronisation of source

code and build system dependencies (RC2). Consistent composition of aspects induces new challenges for incremental weaving because of whole-program reasoning [135], fine-grained composition and aspect interaction [137, 113, 114, 74] (RC3). The higher potential of source code modularity [219, 162], the influence of weaving order [163, 137, 113, 114, 74] and the existence of dependencies between aspects [75] lead to higher demands on the configuration system (RC4). However, aspects also have the ability to decouple the base code from configuration parameters if the build system can exchange the build configuration and structure with the aspects, i.e. if the build system is integrated with the aspect language.

Given these influences of the introduction of AOP on the source code, the four roots of co-evolution suggest that these non-trivial changes require corresponding changes on the build system. Otherwise, the source code and the build system become inconsistent.

11.2.1.5 AOSD Technology to Deal with Co-evolution?

Apart from tool support, the four roots of co-evolution have also contributed to aspect language support for managing co-evolution of source code and the build system. RC1, RC2 and RC4 suggest that the exchange of build system information between the build system and the aspects can improve the robustness of aspects to the build architecture and configuration (goal L2). The build architecture e.g. contains knowledge about the main components in the source code architecture [84], while the configuration choices can be used to extract configuration logic into aspects. The case studies have provided evidence of this.

We have presented the design and implementation of Aspicere, i.e. an aspect language for C which is based on the principles of logic meta-programming (LMP) [237, 30]. Aspicere fulfils goals L1 and L2. To deal with legacy systems (goal L1), Aspicere only introduces two new constructs to C, has an expressive pointcut language, is capable of specifying generic advice and provides access to a variety of join point context and weave-time meta data. The pointcut language is based on a logic programming language (with a temporal extension) to compose robust pointcuts in terms of Prolog queries and facts. The logic facts form the key entry point for the integration of the build system with Aspicere (goal L2), i.e. to pass build system structure and configuration information to Aspicere's logic fact base. This enables access to configuration decisions and build dependencies in pointcuts and advice, which is more powerful than the traditional integration of preprocessor constructs with the base code.

11.2.1.6 Validation of AOSD Technology to Deal with Co-evolution?

We have performed five case studies in which AOP technology is introduced in a legacy C system (Table 11.1 below the Linux kernel). These case studies have col-

lected additional experimental evidence of the existence of co-evolution of source code and the build system (attributed to RC1, RC2, RC3 and RC4), and of the ability of Aspicere to deal with legacy systems (goal L1) and to manage co-evolution of source code and the build system (goal L2). At the same time, these cases have validated MAKAO's ability to help in understanding and managing symptoms of co-evolution of source code and the build system in the presence of AOSD technology (see Section 11.2.1.2).

The main co-evolution problem attributed to RC1 and RC2 is the understanding and definition of the notion of "whole program", i.e. the scope of aspects. Theoretically, aspects apply across the whole base code, but the boundaries of libraries and executables, and complex interactions between these build components blur this notion. In fact, the build system has explicit control over the scope of aspects and hence the semantics of the composed AOSD system.

Current aspect weavers require more time to build a system than base code compilers. Because of the fine-grained aspect composition, the build system is not able to externally accelerate the weaving process without making overly simplistic assumptions which might cause inconsistent builds (RC3). Either weavers should provide internal support for incremental weaving, or build systems should become aware of AOP composition. As witnessed by the colored cells in Table 11.1, a lot of work is still needed to solve the co-evolution problems caused by RC3.

The increased potential for configurability in an AOSD-based system requires tight control. Apart from determining the high-level scope of aspects ("whole program" of RC1), configuration entails selection of a consistent set of aspects to apply on a system and the association of aspects to specific sets of base code modules. This is a challenging task, especially when the configurations heavily fluctuate, e.g. during migration of the system (cf. the ASML case study).

Aspicere's support for integration of build structure and configuration information with the logic fact base (goal L2) has been useful in three cases. The Quake 3 case uses build component information, the aspect implementation of ASML's return-code idiom exploits knowledge of the current build configuration and the Parrot case study applies the currently active configuration options. No application of build dependency information has been presented, but we believe that exchange of build system information with the aspect language in general is an effective means to deal with problems caused by co-evolution of source code and the build system. In general, Aspicere has shown to be capable of dealing with legacy systems (goal L1).

11.2.2 Technical Contributions

Apart from conceptual contributions directly aimed at addressing the six research questions, various technical contributions have been made. This section lists these

contributions.

Investigation of GBS The GNU Build System has been presented as an illustration of the build system model used throughout this dissertation, i.e. the combination of a build and configuration layer, and of the extensive range of channels for configuration between source code and the build system.

Survey of Build Problems We have analysed the research and practice of build systems and have come up with a list of build problems which hamper build systems in legacy systems. Solving the majority of these build problems corresponds to goal T1.

MAKAO Prototype We have built a prototype of MAKAO[1] based on Perl scripts, the GUESS graph manipulation framework[2] and the SWI Prolog engine[3]. This prototype has been used throughout all case studies.

Application of MAKAO on Build Problems Apart from support for co-evolution of source code and the build system, MAKAO has been designed to deal with most of the identified build problems (goal T1). We have validated this on typical build problems in the context of the Kava system, Quake 3 and Linux 2.6.16.18 build systems. The five core features of MAKAO have been capable of tackling the majority of build problems.

Survey of Aspect Languages for C We have made a survey of aspect languages for C in order to compare their language and weaver characteristics. This information has been used to evaluate the existing aspect languages w.r.t. the requirements resulting from goals L1 and L2.

Aspicere Aspicere is our aspect language for C which is based on LMP [237, 30] and has been inspired by Cobble [142, 202], an aspect language for Cobol. Apart from conventional join points like `call` and `execution`, Aspicere provides delimited continuation join points to enable control over the resumption of a procedure. The Prolog-based pointcut language facilitates robust pointcuts which can make use of build structure and configuration meta data stored as logic facts (goal L2). Advice is generic to make it robust to variability in join point context, and Aspicere does only introduce a limited number of new constructs to C (goal L1). Aspicere has been applied in the five AOP case studies.

[1] `http://users.ugent.be/~badams/makao/`
[2] `http://graphexploration.cond.org/`
[3] `http://www.swi-prolog.org/`

Aspicere1 Aspicere1[4] is a source-to-source weaver for Aspicere which has been used in the Kava case. It is implemented in Java, and is based on XML processing [142, 202].

Aspicere2 Aspicere2[5] is a link-time weaver for Aspicere built on top of the LLVM framework[6] and the SWI Prolog engine[7]. This foundation enables powerful static analysis and optimisation of the woven code, and fosters extensibility of the weaver. Aspicere2 has been used in all case studies except for the Kava case.

cHALO cHALO is a history-based extension of Aspicere which is based on the HALO [115] aspect language for Lisp. At run-time, a modified Rete-engine (based on CLIPS[8]) manages program history and determines whether or not join points match. The transparent connection between pointcut primitives and the history retention strategy enables tight control of memory and run-time requirements.

11.3 Future Work

This section presents opportunities for future work. We have made a distinction between minor (Section 11.3.1) and major (Section 11.3.2) areas of future work. The former correspond to more technical problems, whereas the latter represent fundamental open problems.

11.3.1 Minor Future Work

This section considers five areas of more technical future work.

Incremental Weaving in Aspicere2 As identified on various occasions in this dissertation and in Table 11.1, Aspicere2 does not provide incremental weaving. Aspicere1 did not either, but this weaver could benefit from the natural ability for incremental weaving source-to-source weavers have. The main issues to build incremental weaving into Aspicere2 are to keep track of which link modules correspond to which source files, to find out which link modules have been woven into and which ones not, to selectively (un)weave certain parts of the link module and to enable incremental execution of static analyses. The latter is the most complex problem to deal with.

[4] http://users.ugent.be/~badams/aspicere1/
[5] http://users.ugent.be/~badams/aspicere2/
[6] http://llvm.org/
[7] http://www.swi-prolog.org/
[8] http://clipsrules.sourceforge.net/

Support for the Configuration Layer in MAKAO MAKAO has only focused on the build layer enhanced with build-time data. In the case of the Linux kernel, support for analysis of configuration specifications could have given us information about constraints between source code components, explicit mappings between source code and the build system, etc. A simplified model of configuration specifications should be derived from product line research and should be integrated into MAKAO. The enhanced MAKAO could be used to revisit the Linux kernel case study.

Online Filtering Support in MAKAO The current MAKAO prototype only allows offline filtering with logic rules. Integration into GUESS requires online synchronisation of the Gython graph model and the Prolog fact base.

Optimising Speed of Weaving and Woven Code Aspicere2 has not yet been thoroughly profiled. The woven programs it generates have not been instrumented either to measure their performance. Refactoring of Aspicere2 should be steered by identification of areas for speed improvement.

Generative Advice Cobble [202] and Mirjam [179] both have provisions for generative advice, which is a more advanced form of generic advice. We are interested into an extension of Aspicere2 with generative advice to improve the robustness of advice to join point variability.

11.3.2 Major Future Work

This section discusses seven major areas of future work.

An AOP-aware Build System We have argued how the four roots of co-evolution suggest major changes in the build system for keeping the build system consistent with the source code changes introduced by AOSD. Because the re-implementation of a legacy system's build system is not feasible because of investments and the lack of internal knowledge, this dissertation has focused on external tool support to find workarounds and hacks for resolving the symptoms of co-evolution of source code and the build system.

However, an important area of future work would be to investigate whether a build system which has explicit knowledge of aspects and fine-grained composition is better capable of understanding and dealing with co-evolution in the presence of AOSD. This corresponds to the integration of the aspect language with the build system (instead of the inverse), as described in Section 5.1.2 on page 148. A new build model should be distilled, which should be more general than "make"'s directed acyclic graph model. Similarly, a fine-grained configuration model should

be derived. Unfortunately, applying the AOP-aware build system to existing systems is hampered by the need to re-implement the old build system's functionality first.

An even more ambitious project would be to design a build system which is capable of modeling any kind of composition, i.e. to decouple the build system from a particular technology or paradigm.

More Experimental Validation of the Roots of Co-evolution More experimental evidence should be gathered about co-evolution of source code and the build system to further validate the roots of co-evolution. New roots of co-evolution could be identified, or other means for tool support. We should broaden the scope of the experiments by looking at other real-world systems, both open and closed source. This is needed in order to check whether or not our observations hold for a larger class of software. Other categories of systems like compilers, word processors, etc. should be investigated as well. Not only C systems, but also legacy Java or Fortran systems should be examined.

A concrete case study we are thinking of is the Mozilla suite[9]. The Mozilla suite is the open source continuation of the Netscape browser and consists of a web browser, email client, etc. Although the source code now lives on under a different name, the original Mozilla has been abandoned in favour of the separate Firefox browser [10] and Thunderbird email client [11]. A comparison between the build system and source code organisation of the Mozilla Suite and those of Firefox and Thunderbird could teach us a lot about RC1 and RC4.

Analysis of Synchronised Changes of Source Code and Build System When defining the scope of tool support, we explicitly have chosen not to develop support for source code understanding and manipulation. In our case studies, we have used available documentation to fill this void, and in the Linux case in particular, we explicitly have focused on relatively coarse-grained changes of the build system. As an interesting area of future work, a detailed analysis of dependencies between *individual* changes in the source code and the build system in a given software system could shed light on the low-level consequences of co-evolution of source code and the build system. A reviewer of one of our papers [4] has suggested to also take e.g. information of the bug tracking system into account. These measurements could identify finer-grained roots of co-evolution.

Reify the Whole Build DAG as Weaving Meta Data Aspicere provides access to build structure and configuration data to the weaver. However, currently only a

[9] `http://www.mozilla.org/`
[10] `http://www.mozilla-europe.org/nl/products/firefox/`
[11] `http://www.mozilla-europe.org/nl/products/thunderbird/`

limited amount of information is passed to the aspects, i.e. the names of selected modules, active preprocessor flags and the names of libraries and executables. Conceptually, the whole build dependency graph and configuration layer should be reified as logic facts. Unfortunately, this can only be achieved by modifying existing build tools. It is also not clear whether the complete build dependency graph should be passed or only the processed parts. Likewise, separate "make" processes which break the build dependency graph should be dealt with somehow. The impact of this reification in practice should be investigated as well, especially to evaluate whether this improves management of co-evolution of source code and the build system (goal L2).

History Retention in cHALO A detailed account of problems with history retention for history-based pointcut languages in C has been given. The choice for specific pointcut primitives with limited retention of facts is not a silver bullet. Hence, the issues identified in avoiding dangling pointers and unbounded growth of memory should be dealt with. If this is impossible, run-time support, possibly from the operating system, should be pursued.

Aspect Mining in Preprocessed Code We have proposed an outline for an aspect mining approach to (semi-)automatically extract C preprocessor usage into aspects [7], based on a combination of existing techniques and open issues. We have discussed a characterisation of conditional compilation usage in the fifth case study, but the automatic detection and extraction steps and their application in practice is still important future work. These techniques have a direct consequence on RC4.

11.4 Conclusion

This dissertation has investigated the phenomenon of co-evolution of source code and the build system. We have distilled conceptual and gathered experimental evidence of the existence and nature of this co-evolution, and we have shown how the co-evolution causes problems for legacy systems to deal with source code changes introduced by AOSD. To understand and manage co-evolution of source code and the build system, we have distilled requirements for tool and aspect language support. We have validated this support on six case studies. These case studies have shown that co-evolution of source code and the build system is indeed a real problem, but that tool and aspect language support are able to assist in understanding and dealing with it.

A Example GBS Application

This appendix contains an example GBS build system. It is based on a sample system implemented by Alexandre Duret-Lutz for his GBS tutorial[1], which was released to the public domain. It is a simple "Hello World"-program written in C with the following directory layout:

- Makefile.am (Figure A.5)
- configure.ac (Figure A.4)
- lib:
  - Makefile.am (Figure A.6)
  - say.c.in (Figure A.3)
  - say.h (Figure A.2)
- src:
  - Makefile.am (Figure A.7)
  - gettext.h
  - main.c (Figure A.1)

Header file "gettext.h" has been copied from the "gettext" internationalisation tool[2] for distribution with the sample system.

[1] `http://www.lrde.epita.fr/~adl/autotools.html`
[2] `http://www.gnu.org/software/gettext/`

After issuing `autoreconf -install`, the following important files have been added:

- Makefile.in
- aclocal.m4
- autom4te.cache
- config.h.in (Figure A.8)
- configure
- lib:
  - Makefile.in
- po:
  - Makefile.in.in
- src:
  - Makefile.in

The "*.in" files are makefile templates generated from an automake specification with some remaining platform-dependent parameters. The most crucial file is "configure", a shell script which is used by anyone wishing to compile the system on his or her machine. "aclocal.m4" contains all third party m4 macros in use, while "autom4te.cache" is the autotools cache. To pass configuration choices to the source code, the "config.h.in" template can be used. The "po" directory contains localisation code for the source code, i.e. translations.

After distributing the source code archive (generated by "make dist") to a user and unpacking it, the user first has to "./configure –prefix /path/to/install". In the build directory, the following important files are generated (amongst others):

- Makefile
- config.status
- lib:
  - Makefile
- po:
  - Makefile.in
- src:

  - Makefile

The “config.status” is the shell script which is able to fill in all the remaining platform-dependent values in the parametrised source code and build scripts. Now, all what is left to do is compiling the application (“make”) and installing it in the indicated installation directory (“make install”). By fiddling with the `$LANG` environment variable, the specific user localisation can be manipulated.

The installation directory looks like this:

- bin:
  - hello
- lib:
  - libhello.la
  - libhello.a
  - ...
- share

The “share” directory contains the localisation resources.

```
1  #include <config.h>
2  #include <locale.h>
3  #include "gettext.h"
4  #include "say.h"
5
6  int main (void){
7    setlocale(LC_ALL,"");
8    bindtextdomain(PACKAGE,LOCALEDIR);
9    textdomain(PACKAGE);
10
11   say_hello();
12   return 0;
13 }
```

Figure A.1: src/main.c

```
#ifndef AMHELLO_SAY_H
#define AMHELLO_SAY_H
  void say_hello (void);
#endif
```

Figure A.2: lib/say.h

```
#include <config.h>
#include <stdio.h>

#ifdef HAVE_GETTEXT
#include "../src/gettext.h"
#define _(string) gettext (string)
#else
#define _(string) (string)
#endif

void say_hello (void){
  puts (_("Hello World (with%s gettext)!"),
        ("@USE_NLS@"=="yes"?_(""):_("out")));
  printf (_("This is %s.\n"), PACKAGE_STRING);
}
```

Figure A.3: lib/say.c.in

```
AC_INIT([amhello], [2.0], [bug-report@address])
AC_CONFIG_AUX_DIR([build-aux])
AM_INIT_AUTOMAKE([foreign])
AM_GNU_GETTEXT_VERSION([0.14.5])
AM_GNU_GETTEXT([external])
AM_CONDITIONAL(GETTEXT_INSTALLED, [test x$USE_NLS = xyes])
AC_SUBST([USE_NLS])
AC_PROG_LIBTOOL
AC_PROG_CC
AM_CONFIG_HEADER([config.h])
AC_CONFIG_FILES([Makefile lib/Makefile src/Makefile \
                 po/Makefile.in lib/say.c m4/Makefile])
AC_OUTPUT
```

Figure A.4: configure.ac

```
SUBDIRS = po lib src

ACLOCAL_AMFLAGS = -I m4

EXTRA_DIST = build-aux/config.rpath
```

Figure A.5: Makefile.am

```
AM_CPPFLAGS = -DLOCALEDIR=\"$(localedir)\"
lib_LTLIBRARIES = libhello.la
libhello_la_SOURCES = say.c say.h
libhello_la_LDFLAGS = -version-info 0:0:0

localedir = $(datadir)/locale
DEFS = -DLOCALEDIR=\"$(localedir)\" @DEFS@
```

Figure A.6: lib/Makefile.am

```
AM_CPPFLAGS = -I$(srcdir)/../lib \
              -DLOCALEDIR=\"$(localedir)\"
bin_PROGRAMS = hello
hello_SOURCES = main.c gettext.h

if GETTEXT_INSTALLED
LDADD = ../lib/libhello.la $(LIBINTL)
else
LDADD = ../lib/libhello.la
endif

localedir = $(datadir)/locale
DEFS = -DLOCALEDIR=\"$(localedir)\" @DEFS@
```

Figure A.7: src/Makefile.am

```
1  /* config.h.in.  Generated from configure.ac by autoheader.
   */
2
3  /* Define to 1 if translation of program messages to the
4     user's native language is requested. */
5  #undef ENABLE_NLS
6
7  ...
8
9  /* Define if the GNU gettext() function is present or
10    preinstalled. */
11 #undef HAVE_GETTEXT
12
13 ...
14
15 /* Define to 1 if you have the <stdint.h> header file. */
16 #undef HAVE_STDINT_H
17
18 /* Define to 1 if you have the <stdlib.h> header file. */
19 #undef HAVE_STDLIB_H
20
21 ...
22
23 /* Name of package */
24 #undef PACKAGE
25
26 /* Define to the address where bug reports for this
27    package should be sent. */
28 #undef PACKAGE_BUGREPORT
29
30 ...
31
32 /* Define to 1 if you have the ANSI C header files. */
33 #undef STDC_HEADERS
34
35 /* Version number of package */
36 #undef VERSION
```

Figure A.8: config.h.in

B

Rules for Filtering the Linux 2.6.x Build Process

This appendix lists all Prolog rules which have been used during the investigation of the Linux kernel build system (Chapter 4) to filter the Linux 2.6.x build dependency graph. The "FORCE" idiom filtering rule of Section B.4 is explained in Section 3.5.3 on page 96. This explanation suffices to understand the other rules. The semantics of the filtered idioms are discussed in Section 4.5.3.2 on page 127.

B.1 Auxiliary predicates

```
1  %Calculate full path of Target
2  unix_path(Target, PathToTarget) :-
3    target(Target, Name),
4    path_to_target(Target, Path),
5    append(Path, [Name], PathToTarget),
6    simplify_path([], PathToTarget, SimplifiedPathToTarget),
7    concat_atom(SimplifiedPathToTarget, '/', PathToTarget).
8
9  %Auxiliary predicates
10 simplify_path(Path, [], Path).
11
12 simplify_path(Path, [End], Simplified) :-
13   \+ End == '.',
14   \+ End == '..',
```

```
15   append(Path, [End], Simplified).
16
17 simplify_path(Prev, ['.' | Rest], Simplified) :-
18   simplify_path(Prev, Rest, Simplified).
19
20 simplify_path(Prev, [Up | ['..' | Rest]], Simplified) :-
21   \+ Up == '.',
22   append(Prev, Rest, Todo),
23   simplify_path([], Todo, Simplified).
24
25 simplify_path(Prev, [Up | [Here | Rest]], Simplified) :-
26   \+ Up == '.',
27   \+ Here == '..',
28   append(Prev, [Up], Current),
29   simplify_path(Current, [Here | Rest], Simplified).
30
31 % - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
32
33 %Find all directory names occurring
34 folder_target(Folder, Name) :-
35   target(Folder, Name),
36   once((path_to_target(_, P),
37         last(P, Name))).
```

B.2 Eliminate meta-edges

```
1  is_base_dependency(Key):-
2    dependency_type(Key,0).
3
4  is_base_target(Target):-
5    once(
6     ((rdependency(Target,_,Key);dependency(Target,_,Key)),
7      is_base_dependency(Key))
8    ).
9
10 % - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
11
12 meta_cached:-
13   forall(dependency(Src,Dst,Key),
14          assert(rdependency(Dst,Src,Key))).
15
16 meta_target(Target, Name):-
17   target(Target, Name),
18   is_base_target(Target).
19
20 meta_dependency(Target, Dependency, Key):-
21   dependency(Target, Dependency, Key),
22   is_base_dependency(Key).
```

B.3 Initial cleanup

```
autoconf_target(Target) :-
  target(Target,'autoconf.h').

custom_target(Target):-
  (autoconf_target(Target);
   (meta_rdependency_a(Target,Au,_),autoconf_target(Au))).

custom_dependency(Target,Dependency,_):-
  (autoconf_target(Target);autoconf_target(Dependency)).

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

is_tmp_o_target(Target):-
  target_concern(Target,'o'),
  meta_rdependency_a(Target,OTarget,_),
  target_concern(OTarget,'o'),
  meta_target_a(Target,TName),
  concat_atom(['',OName],'.tmp_',TName),
  meta_target_a(OTarget,OName).

is_gcc_dep_target(Target):-
  (target_concern(Target, 'd');
   target_concern(Target, 'tmp');
   target_concern(Target, 'cmd');
   target_concern(Target, 'ver');
   is_tmp_o_target(Target)),
  \+ meta_dependency_a(Target,_,_).

is_gcc_dep_dependency(_,Dependency):-
  is_gcc_dep_target(Dependency).

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

looping_dependency(X, X, Key):-
  meta_dependency_a(X, X, Key).

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

clean_cached:-
  forall(meta_target(Target,Name),
         assert(meta_target_a(Target,Name))),
  forall(meta_dependency(Src,Dst,Key),
         assert(meta_dependency_a(Src,Dst,Key))),
  forall(meta_dependency_a(Src,Dst,Key),
         assert(meta_rdependency_a(Dst,Src,Key))).

```

```
clean_target(Target, Name):-
  meta_target_a(Target, Name),
  \+ is_gcc_dep_target(Target),
  \+ custom_target(Target).

clean_dependency(Target, Dependency, Key):-
  meta_dependency_a(Target, Dependency, Key),
  \+ looping_dependency(Target, Dependency, Key),
  \+ is_gcc_dep_dependency(Target,Dependency),
  \+ custom_dependency(Target,Dependency,Key).
```

B.4 FORCE idiom

```
force_target(Force, 'FORCE') :-
  clean_target_a(Force, 'FORCE').

force_dependency(Target, Force, Key) :-
  force_target(Force, _),
  clean_rdependency_a(Force, Target, Key).

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

forceless_cached:-
  forall(clean_target(Target,Name),
         assert(clean_target_a(Target,Name))),
  forall(clean_dependency(Src,Dst,Key),
          (assert(clean_dependency_a(Src,Dst,Key)),
           assert(clean_rdependency_a(Dst,Src,Key)))).

forceless_target(Target, Name):-
  clean_target_a(Target, Name),
  \+ force_target(Target,_).

forceless_dependency(Target, Dependency, Key):-
  clean_dependency(Target, Dependency, Key),
  \+ force_dependency(Target,Dependency,Key).
```

B.5 Shipped targets

```
is_shipped_target(Target):-
  target_concern(Target,Concern),
  concat_atom(ConcernPieces,'.',Concern),
  last(ConcernPieces,'c_shipped').

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

shipless_cached:-
  forall(forceless_target(Target, Name),
```

```
10          assert(forceless_target_a(Target, Name))),
11   forall(forceless_dependency(Target, Dep,Key),
12          assert(forceless_dependency_a(Target, Dep,Key))).
13
14 shipless_target(Target,Name):-
15   forceless_target_a(Target,Name),
16   \+is_shipped_target(Target).
17
18 shipless_dependency(Target,Dep,Key):-
19   forceless_dependency_a(Target,Dep,Key),
20   \+is_shipped_target(Dep).
```

B.6 Source-level abstraction

```
1 is_leaf_c_target(Target):-
2   target_concern(Target,'c'),
3   \+shipless_dependency_a(Target,_,_).
4
5 % - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
6
7 abs_cached:-
8   forall(shipless_target(Target, Name),
9          assert(shipless_target_a(Target, Name))),
10  forall(shipless_dependency(Target, Dep,Key),
11         assert(shipless_dependency_a(Target, Dep,Key))).
12
13 abs_target(Target,Name):-
14   shipless_target_a(Target,Name),
15   \+is_leaf_c_target(Target).
16
17 abs_dependency(Target,Dep,Key):-
18   shipless_dependency_a(Target,Dep,Key),
19   \+is_leaf_c_target(Dep).
```

B.7 Composite object abstraction

```
1 is_simple_o_target(Target):-
2   target_concern(Target,'o'),
3   \+ abs_dependency_a(Target,Dep,_),
4   forall(abs_rdependency_a(Target,Dep,_),
5          target_concern(Dep,'o')).
6
7 % - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
8
9 com_cached:-
10  forall(abs_target(Target, Name),
11         assert(abs_target_a(Target, Name))),
12  forall(abs_dependency(Target, Dep,Key),
```

```
          (assert(abs_dependency_a(Target,Dep,Key)),
           assert(abs_rdependency_a(Dep,Target,Key)))).

com_target(Target,Name):-
  abs_target_a(Target,Name),
  \+ is_simple_o_target(Target),
  assert(com_target_a(Target,Name)).

com_dependency(Target,Dep,Key):-
  abs_dependency_a(Target,Dep,Key),
  com_target_a(Dep,_).
```

B.8 Circular dependency chain

```
linux___build_target(Build, '__build') :-
  com_target_a(Build, '__build').

recursive_make_idiom(Build, Folder, Name) :-
  folder_target_a(Folder, Name),
  linux___build_target(Build, _),
  com_dependency_a(Build, Folder, _),
  com_dependency_a(Folder, Build, _),
  once((com_rdependency_a(Folder,SimpleObject, _),
        target_concern(SimpleObject, 'o'),
        com_rdependency_a(SimpleObject,CompositeObject,_),
        target_concern(CompositeObject, 'o'))),
  com_dependency_a(Build, CompositeObject, _).

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

circ_cached:-
  forall(folder_target(Folder, FolderName),
         assert(folder_target_a(Folder, FolderName))),
  forall(com_dependency(Target,Dep,Key),
         (assert(com_dependency_a(Target,Dep,Key)),
          assert(com_rdependency_a(Dep,Target,Key)))).

circ_target(Target, Name) :-
  com_target_a(Target, Name).

circ_dependency(Target, Dependency, Key) :-
  com_dependency_a(Target, Dependency, Key),
  \+recursive_make_idiom(Target,Dependency,_).
```

References

[1] R.A. Åberg, J.L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th Conference on Automated Software Engineering (ASE)*, pages 196–204, 6-10 Oct. 2003.

[2] Bram Adams. AOP on the C-side. In *Proceedings of the 2nd Linking Aspect Technology and Evolution Workshop (LATEr), AOSD*, Bonn, Germany, 2006.

[3] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In Ladan Tahvildari and Gerardo Canfora, editors, *Proceedings of the 23rd International Conference on Software Maintenance (ICSM)*, pages 114–123, Paris, France, October 2007. IEEE Computer Society.

[4] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the Linux build system. In *Informal Proceedings of the 3rd International ERCIM Workshop on Software Evolution at ICSM*, pages 93–102, Paris, France, October 2007.

[5] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the Linux build system. *Electronic Communications of the ECEASST*, 8, February 2008.

[6] Bram Adams, Charlotte Herzeel, and Kris Gybels. cHALO, stateful aspects in C. In *Proceedings of the 7th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, Brussels, Belgium, 2008.

[7] Bram Adams, Bart Van Rompaey, Celina Gibbs, Yvonne Coady, and Herman Tromp. Aspect mining in the presence of the C preprocessor. In *Proceedings of the 4th Linking Aspect Technology and Evolution Workshop (LATE), AOSD*, Brussels, Belgium, 2008.

[8] Bram Adams and Kris De Schutter. An aspect for idiom-based exception handling (using local continuation join points, join point properties,

annotations and type parameters). In *Proceedings of the 5th Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD*, Vancouver, Canada, March 2007.

[9] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Trans. Softw. Eng. Methodol.*, 3(1):3–28, 1994.

[10] Eytan Adar. GUESS: a language and interface for graph exploration. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pages 791–800, Montréal, Québec, Canada, April 2006.

[11] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586, pages 144–168, Glasgow, Scotland, 2005. Springer.

[12] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 345–364, San Diego, CA, USA, 2005. ACM.

[13] Glenn Ammons. Grexmk: speeding up scripted builds. In *Proceedings of the international workshop on Dynamic systems analysis (WODA)*, pages 81–87, Shanghai, China, 2006. ACM.

[14] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 117–128, Chicago, IL, USA, 2005. ACM Press.

[15] Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc* : An Extensible AspectJ Compiler. *Transactions on Aspect-Oriented Software Development I*, 3880:293–334, 2006.

[16] Pavel Avgustinov, Julian Tibble, Eric Bodden, Laurie Hendren, Ondrej Lhotak, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. In *Companion to the 21st ACM SIGPLAN conference on*

Object-oriented programming systems, languages, and applications (OOPSLA), pages 685–686, Portland, OR, USA, 2006. ACM.

[17] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. *SIGPLAN Not.*, 42(10):589–608, 2007.

[18] John Backus. The history of Fortran I, II, and III. *SIGPLAN Not.*, pages 25–74, 1981.

[19] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity*. MIT Press, Cambridge, MA, USA, 1999.

[20] M. Baxter, I.D.; Mehlich. Preprocessor conditional removal by simple partial evaluation. *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, pages 281–290, 2001.

[21] Keith Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[22] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[23] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Trans. Program. Lang. Syst.*, 21(4):813–847, 1999.

[24] Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. Envelope-based weaving for faster aspect compilers. In *Proceedings of the 6th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a NetworkedWorld (Net.ObjectDays)*, pages 3–18, Erfurt, Germany, 2005.

[25] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD)*, pages 83–92, Lancaster, UK, 2004. ACM.

[26] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 125–138, Portland, OR, USA, 2006. ACM.

[27] Eric Bodden, Laurie Hendren, and Ondřej LhotÁk. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21th European Conference on Object-Oriented Programming (ECOOP)*, pages 525–549, Berlin, Germany, 2007.

[28] Eric Bodden, Patrick Lam, and Laurie Hendren. Flow-sensitive static optimizations for runtime monitors. Technical Report abc-2007-3, Sable Research Group, McGill University, July 2007.

[29] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 555–563, Los Angeles, CA, USA, 1999. IEEE Computer Society Press.

[30] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE)*, pages 110–127, Pittsburgh, PA, USA, 2002. Springer-Verlag.

[31] Michael Brodie and Michael Stonebreaker. *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann, 1995.

[32] Magiel Bruntink, Arie van Deursen, Maja D'Hondt, and Tom Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 199–211, Vancouver, BC, Canada, March 2007.

[33] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. An initial experiment in reverse engineering aspects. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pages 306–307, Delft, The Netherlands, November 2004. IEEE Computer Society.

[34] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 37–46, Budapest, Hungary, 2005. IEEE Computer Society.

[35] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In *Proceeding of the 28th international conference on Software engineering (ICSE)*, pages 242–251, Shanghai, China, 2006. ACM Press.

[36] Magiel Bruntink, Arie van Deursen, Tom Tourwé, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, pages 200–209, Chicago, IL, USA, September 2004.

[37] J. Buffenbarger and K. Gruell. A language for software subsystem composition. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS)*, page 9072, Maui, HI, USA, 2001. IEEE Computer Society.

[38] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–28, Boston, MA, USA, 2004.

[39] Andrea Capiluppi, Maurizio Morisio, and Juan F. Ramil. The evolution of source folder structure in actively evolved open source systems. In *Proceedings of the 10th International Symposium on Software Metrics (METRICS)*, pages 2–13, Chicago, IL, USA, 2004. IEEE Computer Society.

[40] Andrea Capiluppi, Maurizio Morisio, and Juan F. Ramil. Structural evolution of an open source system: A case study. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC)*, page 172, Bari, Italy, 2004. IEEE Computer Society.

[41] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 266–277, Paris, France, 1997. ACM.

[42] P. M. Cashin, M. L. Joliat, R. F. Kamel, and D. M. Lasker. Experience with a modular typed language: PROTEL. In *Proceedings of the 5th international conference on Software engineering (ICSE)*, pages 136–143, San Diego, CA, USA, 1981. IEEE Press.

[43] Craig Chambers, Jeffrey Dean, and David Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *Proceedings of the 17th international conference on Software engineering (ICSE)*, pages 221–230, Seattle, WA, USA, 1995. ACM.

[44] John Champaign, Andrew Malton, and Xinyi Dong. Stability and volatility in the Linux kernel. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE)*, page 95, Helsinki, Finland, 2003. IEEE Computer Society.

[45] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. *SIGPLAN Not.*, 42(10):569–588, 2007.

[46] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[47] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Trans. Program. Lang. Syst.*, 12(1):1–25, 1990.

[48] Yvonne Coady, Celina Gibbs, Michael Haupt, Jan Vitek, and Hiroshi Yamauchi. Towards a domain specific language for virtual machines. In *Proceedings of the 1st Domain-Specific Aspect Languages Workshop (DSAL), AOSD*, Portland, OR, USA, October 2006.

[49] Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 50–59, Boston, MA, USA, 2003.

[50] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, 2001.

[51] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD)*, pages 56–65, Lancaster, UK, 2004. ACM.

[52] Keith D. Cooper, Ken Kennedy, and Linda Lorczon. Interprocedural optimization: eliminating unnecessary recompilation. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction (SC)*, pages 58–67, Palo Alto, CA, USA, 1986. ACM.

[53] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Joinpoint inference from behavioral specification to implementation. In *Proceedings of the 21th European Conference on Object-Oriented Programming (ECOOP)*, pages 476–500, Berlin, Germany, 2007.

[54] Daniel S. Dantas and David Walker. Harmless advice. *ACM SIGPLAN Notices*, 41(1):383–396, January 2006.

[55] Merijn de Jonge. The Linux kernel as flexible product-line architecture. Technical Report SEN-R0205, CWI, 2002.

[56] Merijn de Jonge. *To reuse or to be reused: Techniques for component composition and construction*. PhD thesis, University of Amsterdam, January 2003.

[57] Merijn de Jonge. Build-level components. *IEEE Trans. Softw. Eng.*, 31(7):588–600, 2005.

[58] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[59] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. *Practical Aspects of Declarative Languages*, 1990:184–198, 2001.

[60] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, Los Angeles, CA, USA, 1975. ACM.

[61] Mikhail Dmitriev. Language-specific make technology for the Java programming language. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 373–385, Seattle, WA, USA, 2002. ACM.

[62] Eelco Dolstra. Integrating software construction and software deployment. In *Proceedings of the 11th International Workshop on Software Configuration Management (SCM)*, volume 2649 of *Lecture Notes in Computer Science*, pages 102–117, Portland, OR, USA, May 2003.

[63] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *Proceedings of the 18th USENIX conference on System administration (LISA)*, pages 79–92, Atlanta, GA, USA, 2004. USENIX Association.

[64] Eelco Dolstra, Gert Florijn, Merijn de Jonge, and Eelco Visser. Capturing timeline variability with transparent configuration environments. In *Proceedings of the International Workshop on Software Variability Management*, Portland, OR, USA, May 2003.

[65] Eelco Dolstra and Armijn Hemel. Purely functional system configuration management. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS), USENIX*, Portland, OR, USA, May 2007.

[66] Eelco Dolstra, Eelco Visser, and Merijn de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 583–592, Edinburgh, UK, 2004. IEEE Computer Society.

[67] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE)*, pages 173–188, Pittsburgh, PA, USAA, 2002. Springer-Verlag.

[68] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language

for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 27–38, Chicago, Illinois, March 2005. ACM Press.

[69] Rémi Douence and Olivier Motelet. Sophisticated crosscuts for e-commerce. In *Proceedings of the workshop on Advanced Separation of Concerns*, Budapest, Hungary, 2001.

[70] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION)*, pages 170–186, Kyoto, Japan, 2001. Springer-Verlag.

[71] Paul F. Dubois, Thomas Epperly, and Gary Kumfert. Why Johnny can't build. *Computing in Science and Engg.*, 5(5):83–88, 2003.

[72] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of 12th International Smalltalk Conference (ISC)*, pages 231–257, Prague, Czech Republic, September 2004.

[73] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 150–169, Vancouver, BC, Canada, 2004. ACM Press.

[74] Pascal Dürr, Lodewijk Bergmans, and Mehmet Akşit. *Ideals: evolvability of software-intensive high-tech systems*, chapter Detecting behavioral conflicts among crosscutting concerns, pages 55–67. Embedded Systems Institute, TU/e Campus, Eindhoven, The Netherlands, December 2007.

[75] Pascal Durr, Gurcan Gulesir, Lodewijk Bergmans, Mehmet Aksit, and Remco van Engelen. Applying AOP in an industrial context. In *Proceedings of the Workshop on Best Practices in Applying Aspect-Oriented Software Development (BPAOSD), AOSD*, Bonn, Germany, March 2006.

[76] Robert Dyer and Hridesh Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD)*, Brussels, Belgium, April 2008. ACM Press. To appear.

[77] Yusuke Endoh, Hidehiko Masuhara, and Akinori Yonezawa. Continuation join points. In *Proceedings of the Workshop on the Foundations of Aspect-Oriented Languages (FOAL), AOSD*, pages 1–10, Bonn, Germany, March 2006.

[78] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 51–62, Chicago, IL, USA, 2005. ACM Press.

[79] Michael Engel and Bernd Freisleben. Using a LowLevel Virtual Machine to improve dynamic aspect support in operating system kernels. In *Proceedings of the 4th workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, Chicago, IL, USA, 2005.

[80] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, 2002.

[81] Jacky Estublier. Software configuration management: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE)*, pages 279–289, Limerick, Ireland, 2000. ACM.

[82] Homayoun D. Fard, Yijun Yu, John Mylopoulos, and Periklis Andritsos. Improving the build architecture of legacy C/C++ software systems. In *Proceedings of Fundamental Approaches in Software Engineering (FASE)*, pages 96–110, Edinburgh, Scotland, 2005.

[83] J.-M. Favre. Preprocessors from an abstract point of view. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE)*, page 287, Monterey, CA, USA, 1996. IEEE Computer Society.

[84] J.-M. Favre. Understanding-in-the-large. *Proceedings of the 5th International Workshop on Program Comprehension (IWPC)*, pages 29–38, March 1997.

[85] Jean-Marie Favre. Meta-model and model co-evolution within the 3D software space. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications, ICSM*, Amsterdam, The Netherlands, September 2003.

[86] Jean-Marie Favre. Languages evolve too! changing the software time scale. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE)*, pages 33–44, Lisbon, Portugal, 2005. IEEE Computer Society.

[87] P. Feiler, S. Dart, and G. Downey. Evaluation of the Rational environment. Technical Report CMU/SEI-88-TR-15, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PE, USA, July 1988.

[88] Peter H. Feiler and Raul Medina-Mora. An incremental programming environment. In *Proceedings of the 5th international conference on Software engineering (ICSE)*, pages 44–53, San Diego, CA, USA, 1981. IEEE Press.

[89] Stuart I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, 1979.

[90] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Marrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, 1987.

[91] Juan Fernandez-Ramil, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi. Empirical studies of open source evolution. In Tom Mens and Serge Demeyer, editors, *Software evolution*, chapter 11, pages 263–288. Springer Verlag, 1st edition edition, February 2008.

[92] Patrick Finnigan, Richard C. Holt, Ivan Kallas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kerny Wong. The software bookshelf. *Advances in software engineering*, pages 295–339, 2002.

[93] Marc Fiuczynksi, Robert Grimm, Yvonne Coady, and David Walker. patch (1) Considered Harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, USA, 2005.

[94] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.

[95] G. Fowler. A case for make. *Softw. Pract. Exper.*, 20(S1):35–46, 1990.

[96] Andreas Gal and Olaf Spinczyk. Build management for AspectC++. In *Proceedings of the Workshop on Tools for Aspect-Oriented Software Development (TAOSD), OOPSLA*, Seattle, WA, USA, November 2002.

[97] Alejandra Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, Univ. of Illinois at Urbana-Champaign, August 2005.

[98] Wasif Gilani and Olaf Spinczyk. Dynamic aspect weaver family for family-based adaptable systems. In Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, and Mathias Weske, editors, *Proceedings of the 6th Annual*

International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a NetworkedWorld (Net.ObjectDays), volume 69 of *LNI*, pages 94–109, Erfurt, Germany, 2005. GI.

[99] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 16th International Conference on Software Maintenance (ICSM)*, page 131, San Jose, CA, 2000. IEEE Computer Society.

[100] Ryan Golbeck, Sam Davis, Immad Naseer, Igor Ostrovsky, and Gregor Kiczales. Lightweight virtual machine support for AspectJ. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD)*, Brussels, Belgium, April 2008. ACM Press. To appear.

[101] Adele Goldberg. *SMALLTALK-80: the interactive programming environment*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[102] Weigang Michael Gong and Hans-Arno Jacobsen. *AspeCt-oriented C Specification*. Middleware Systems Research Group, 0.8 edition, January 2008.

[103] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 36–45, Lancaster, UK, 2004. ACM Press.

[104] Robert Grimm. Systems need languages need systems! In *Proceedings of the 2nd Workshop on Programming Languages and Operating Systems (PLOS), ECOOP*, Glasgow, UK, 2005.

[105] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[106] J. Gutknecht. Separate compilation in Modula-2: An approach to efficient symbol files. *Software, IEEE*, 3(6):29–38, 1986.

[107] Jr Guy Lewis Steele. *RABBIT: A Compiler for SCHEME*. PhD thesis, AI Lab MIT, May 1978.

[108] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 60–69, Boston, MA, USA, March 2003. ACM Press.

[109] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with Datalog. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Nantes, France, 2006. Springer.

[110] Maarit Harsu. Translation of conditional compilation. *Nordic J. of Computing*, 6(1):93–109, 1999.

[111] Ahmed E. Hassan, Zhen Ming Jiang, and Richard C. Holt. Source versus object code extraction for recovering software architecture. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE)*, pages 67–76, Pittsburgh, PA, USA, 2005. IEEE Computer Society.

[112] Michael Haupt, Celina Gibbs, and Yvonne Coady. Disentangling virtual machine architecture. In *Proceedings of the 4th Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT)*, Berlin, Germany, July 2007.

[113] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proceedings of the 5th international conference on Aspect-oriented software development (AOSD)*, pages 214–225, Bonn, Germany, 2006. ACM.

[114] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD)*, pages 85–95, Vancouver, BC, Canada, 2007. ACM.

[115] C. Herzeel, K. Gybels, P. Costanza, C. De Roover, and T. D'Hondt. Forward chaining in HALO: An implementation strategy for history-based logic pointcuts. In *Proceedings of the International Conference on Dynamic Languages (ESUG)*, Lugano, Switzerland, August 2007. Springer LNCS.

[116] Charlotte Herzeel, Kris Gybels, and Pascal Costanza. Escaping with future variables in HALO. *Runtime Verification*, pages 51–62, 2007.

[117] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management. Technical Report 168, Compaq Systems Research Center, 1999.

[118] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD)*, pages 26–35, Lancaster, UK, 2004. ACM.

[119] Richard C. Holt, Michael W. Godfrey, and Andrew J. Malton. The build/-comprehend pipelines. In *Proceedings of the 2nd workshop on software architecture (ASERC)*, Banff, AB, Canada, February 2003.

[120] Robert Hood, Ken Kennedy, and Hausi A Müller. Efficient recompilation of module interfaces in a software development environment. In *Proceedings of the second ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (SDE)*, pages 180–189, Palo Alto, CA, USA, 1987. ACM.

[121] Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Lagüe. C/C++ conditional compilation analysis using symbolic execution. In *Proceedings of the 16th International Conference on Software Maintenance (ICSM)*, page 196, Bad Gastein, Austria, 2000. IEEE Computer Society.

[122] Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings of the 21th European Conference on Object-Oriented Programming (ECOOP)*, pages 399–424, Berlin, Germany, 2007.

[123] Gregory F. Johnson. GI: a denotational testbed with continuations and partial continuations. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 165–176, Saint-Paul, MN, USA, June 1987.

[124] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[125] Niels Jörgensen. Safeness of make-based incremental recompilation. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right (FME)*, pages 126–145, London, UK, 2002. Springer-Verlag.

[126] Michael Karasick. The architecture of montana: an open and extensible programming environment with an incremental C++ compiler. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, pages 131–142, Lake Buena Vista, FL, USA, 1998. ACM.

[127] Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engg.*, 6(2):107–138, 1999.

[128] Andy Kellens. *Maintaining causality between design regularities and source code*. PhD thesis, Vrije Univeristeit Brussel, Pleinlaan 2, Etterbeek, 2007.

[129] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, pages 501–525, Nantes, France, 2006.

[130] Andy Kellens, Kim Mens, and Paolo Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, IV(LNCS 4640):143–162, 2007.

[131] Al Kelley and Ira Pohl. *A book on C (Fourth Edition)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1998.

[132] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[133] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.

[134] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[135] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 49–58, St. Louis, MO, USA, 2005. ACM.

[136] Oleg Kiselyov and Chung-chieh Shan. Delimited continuations in operating systems. *Modeling and Using Context*, pages 291–302, 2007.

[137] Günter Kniesel. Detection and resolution of weaving interactions. *Transactions on Aspect-Oriented Software Development, Special issue on 'Dependencies and Interactions with Aspects'*, 2007. To appear.

[138] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th international conference on Software engineering (ICSE)*, pages 49–57, Sorrento, Italy, 1994. IEEE Computer Society Press.

[139] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[140] Bernt Kullbach and Volker Riediger. Folding: An approach to enable program understanding of preprocessed languages. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, page 3, Stuttgart, Germany, 2001. IEEE Computer Society.

[141] David Alex Lamb. Relations in software manufacture. Technical Report ISSN-0836-0227-90-292a, Department of Computing and Information Science, Queen's University, Kingston, ON K7L 3N6, March 1991.

[142] Ralf Lämmel and Kris De Schutter. What does Aspect Oriented Programming mean to Cobol? In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 99–110, Chicago, IL, USA, 2005. ACM Press.

[143] Butler W. Lampson and Eric E. Schmidt. Practical use of a polymorphic applicative language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*, pages 237–255, Austin, TX, USA, 1983. ACM.

[144] Mario Latendresse. Fast symbolic evaluation of C/C++ preprocessing using conditional values. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, page 170, Benevento, Italy, 2003. IEEE Computer Society.

[145] Mario Latendresse. Rewrite systems for symbolic evaluation of C-like preprocessing. In *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR)*, page 165, Tampere, Finland, 2004. IEEE Computer Society.

[146] Hugh C. Lauer and Edwin H. Satterthwaite. The impact of Mesa on system design. In *Proceedings of the 4th international conference on Software engineering (ICSE)*, pages 174–182, Munich, Germany, 1979. IEEE Press.

[147] David B. Leblang and Jr. Robert P. Chase. Computer-aided software engineering in a distributed workstation environment. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (SDE)*, pages 104–112, Pittsburgh, PE, USA, 1984. ACM.

[148] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[149] B. Lewis and D. J. Berg. *Threads Primer. A Guide to Multithreaded Programming*. Prentice Hall, 1996.

[150] C. H. Lindsey. A history of ALGOL 68. In *Proceedings of the 2nd ACM SIGPLAN conference on History of programming languages (HOPL)*, pages 97–132, Cambridge, MA, USA, 1993. ACM.

[151] C. H. Lindsey and H. J. Boom. A modules and separate compilation facility for ALGOL 68. *ALGOL Bull.*, (43):19–53, 1978.

[152] *Linux kernel build documentation*, linux 2.4.0 edition.

[153] *Linux kernel build documentation*, linux 2.6.0 edition.

[154] *Linux kernel build documentation*, linux 2.6.16.18 edition.

[155] Linux-kbuild mailing list. http://www.torque.net/kbuild/archive/.

[156] Kbuild 2.5 history. http://kbuild.sourceforge.net/.

[157] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering (ICSE)*, pages 418–427, Limerick, Ireland, 2000. ACM Press.

[158] D.T. Livadas, P.E.; Small. Understanding code containing preprocessor constructs. *Program Comprehension, 1994. Proceedings., IEEE Third Workshop on*, pages 89–97, 14-15 Nov 1994.

[159] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In Gabor Karsai and Eelco Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3286 of *svlncs*, pages 55–74, Vancouver, BC, Canada, October 2004. Springer.

[160] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. *Lean and Efficient System Software Product Lines: Where Aspects Beat Objects*, volume II of *Lecture Notes in Computer Science*, pages 227–255. Springer Verlag, 2006.

[161] Cristina Videira Lopes. *D: A language framework for distributed programming*. PhD thesis, College of Computer Science, Northeastern University, Boston, 1997.

[162] Cristina Videira Lopes and Sushil Krishna Bajracharya. An analysis of modularity in aspect oriented design. In *Proceedings of the 4th international conference on Aspect-oriented software development (AOSD)*, pages 15–26, Chicago, IL, USA, 2005. ACM.

[163] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 68–77, Charleston, SC, USA, 2006. ACM.

[164] Nicolas Loriant, Marc Ségura-Devillechaise, Thomas Fritz, and Jean-Marc Menaud. A reflexive extension to Arachne's aspect language. In *Proceedings of the workshop on Open and Dynamic Aspect Languages (ODAL'O6), AOSD*, Bonn, Germany, 2006.

[165] Axel Mahler and Andreas Lampen. An integrated toolset for engineering software configurations. In *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (SDE)*, pages 191–200, Boston, MA, USA, 1988. ACM.

[166] Bill McCloskey and Eric Brewer. ASTEC: a new approach to refactoring C. *SIGSOFT Softw. Eng. Notes*, 30(5):21–30, 2005.

[167] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly Media, Inc., 3rd edition edition, 2004.

[168] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[169] Christopher A. Mennie and Charles L.A. Clarke. Giving meaning to macros. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC)*, pages 79–85, Bari, Italy, 2004. IEEE Computer Society.

[170] Kim Mens and Tom Tourwé. Evolution issues in aspect-oriented programming. In Tom Mens and Serge Demeyer, editors, *Software evolution*, chapter 9, pages 203–232. Springer Verlag, 1st edition edition, February 2008.

[171] Peter Miller. Recursive make considered harmful. *AUUGN Journal of AUUG, Inc.*, 19(1):14–25, 1998.

[172] Leon Moonen, Arie Deursen, Andy Zaidman, and Magiel Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In Tom Mens and Serge Demeyer, editors, *Software evolution*, chapter 8, pages 173–202. Springer Verlag, 1st edition edition, February 2008.

[173] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th international conference on Software engineering (ICSE)*, pages 80–86, Singapore, 1988. IEEE Computer Society Press.

[174] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE)*, pages 47–60, Limerick, Ireland, 2000. ACM Press.

[175] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 217–226, Toronto, ON, Canada, 1993. IBM Press.

[176] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[177] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering (FSE)*, pages 18–28, Washington, D.C., USA, 1995. ACM Press.

[178] Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. In Andreas Polze Robert Hirschfeld, Ryszard Kowalczyk and Mathias Weske, editors, *Proceedings of the 6th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a NetworkedWorld (Net.ObjectDays)*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[179] Istvan Nágy, Remco van Engelen, and Durk van der Ploeg. *Ideals: evolvability of software-intensive high-tech systems*, chapter An overview of Mirjam and WeaveC, pages 69–86. Embedded Systems Institute, TU/e Campus, Eindhoven, The Netherlands, December 2007.

[180] Srinivas Neginhal and Suraj Kothari. Event views and graph reductions for understanding system level C code. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 279–288, Philadelphia, PE, USA, 2006. IEEE Computer Society.

[181] Glenn Niemeyer and Jeremy Poteet. *Extreme Programming with Ant: Building and Deploying Java Applications with JSP, EJB, XSLT, XDoclet, and JUnit*. Sams, first edition edition, May 2003. ISBN-0672325624.

[182] M.E. Nordberg III. Aspect-Oriented Dependency Inversion. In *Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA*, Tampa Bay, FL, USA, 2001.

[183] Esko Nuutila, Vesa Hirvisalo, Jari Arkko, Juha Kuusela, and Markku Tamminen. Smart recompilation in the XE compiler (extended abstract), 1986. Unpublished.

[184] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. *SIGPLAN Not.*, pages 439–480, 1981.

[185] Klaus Ostermann, Mira Mezini, and Christophe Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, Glasgow, Scotland, 2005.

[186] Keith Owens. If you want Kbuild 2.5, tell Linus (email). http://lwn.net/Articles/1500/, June 2002.

[187] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[188] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo. A language-independent software renovation framework. *J. Syst. Softw.*, 77(3):225–240, 2005.

[189] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD)*, pages 100–109, Boston, MA, USA, 2003. ACM.

[190] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development (AOSD)*, pages 141–147, Enschede, The Netherlands, 2002. ACM.

[191] Quake3. http://www.idsoftware.com/games/quake/quake3-arena/.

[192] Russell W. Quong and Mark A. Linton. Linking programs incrementally. *ACM Trans. Program. Lang. Syst.*, 13(1):1–20, 1991.

[193] George Radin. The early history and characteristics of PL/I. *SIGPLAN Not.*, 13(8):227–241, 1978.

[194] The Ideals research team. *Ideals: evolvability of software-intensive high-tech systems*, chapter Industrial impact, lessons learned and conclusions, pages 143–160. Embedded Systems Institute, TU/e Campus, Eindhoven, The Netherlands, December 2007.

[195] Gregorio Robles. *Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, February 2006.

[196] Marc J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[197] O. Rohlik, A. Pasetti, V. Cechticky, and I. Birrer. Implementing Adaptability in Embedded Software through Aspect Oriented Programming. In *Proceedings of the IEEE Conference on Mechatronics & Robotics*, pages 85–90, Aachen, Germany, September 2004.

[198] Ed Roman, Scott W. Ambler, and Tyler Jewell. *Mastering Enterprise Javabeans*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[199] Graham Ross. Integral-c -âĂŤ- a practical environment for C programming. In *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (SDE)*, pages 42–48, Palo Alto, CA, USA, 1987. ACM.

[200] Daniel Sabbah. Aspects: from promise to reality. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 1–2, Lancaster, UK, 2004. ACM Press.

[201] Patrick Sansom. Smart Recompilation in Glasgow Haskell. In Phil Trinder, editor, *Proceedings of the Glasgow Functional Programming Workshop (FP)*, Ceilidh Place, Ullapool, Scotland, july 1996.

[202] Kris De Schutter. *Aspect oriented revitalisation of legacy software through logic meta-programming*. PhD thesis, Ghent University, Ghent, Belgium, May 2006.

[203] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 110–119, Boston, MA, USA, 2003. ACM.

[204] Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-CLR: A tool for navigating highly configurable system software. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, Vancouver, BC, Canada, 2007.

[205] P. Singleton and P. Brereton. A Case for Declarative Programming-in-the-Large. In *Proceedings of the 5th IEEE Conf. on Software Engineering and Knowledge Engineering (SEKE)*, San Francisco, CA, USA, 1993.

[206] Paul Singleton. *Applications of Meta-Programming to the Construction of Software Products from Generic Configurations*. PhD thesis, Keele University, Keele, Newcastle, UK, 1992.

[207] Dag I. K. Sjøberg, Ray Welland, Malcolm P. Atkinson, Paul Philbrow, and Cathy Waite. Exploiting persistence in build management. *Softw. Pract. Exper.*, 27(4):447–480, 1997.

[208] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[209] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(2):146–189, 1996.

[210] S. S. Somé and T. C. Lethbridge. Parsing minimization when extracting information from code in the presence of conditional compilation. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC)*, page 118, Ischia, Italy, 1998. IEEE Computer Society.

[211] Henry Spencer and Geoff Collyer. #ifdef considered harmful or portability experience with C News. In Rick Adams, editor, *Proceedings of the USENIX Conference*, pages 185âĂŞ–198, Baltimore, MD, USA, June 1992. USENIX Association.

[212] Dennis Spenkelink. Incremental compilation in Compose*. Master's thesis, University of Twente, 2006.

[213] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the 40th International Conference on Tools Pacific*, pages 53–60, Sydney, Australia, 2002. Australian Computer Society, Inc.

[214] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Know.-Based Syst.*, 20(7):636–651, 2007.

[215] Diomidis Spinellis. Checking C declarations at link time. *The Journal of C language translation*, 4(3):238–249, March 1993.

[216] Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Trans. Softw. Eng.*, 29(11):1019–1030, 2003.

[217] Amitabh Srivastava and Alan Eustace. ATOM — A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, Orlando, FL, USA, 1994.

[218] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make manual*. Free Software Foundation, April 2006. http://www.gnu.org/software/make/manual/make.html.

[219] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/SIGSOFT FSE)*, pages 166–175, Lisbon, Portugal, 2005. ACM.

[220] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *Proceedings of the 8th European Software Engineering Conference, Held Jointly with the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, pages 99–108, Vienna, Austria, 2001. ACM.

[221] Andrew Sutton and Jonathan Maletic. How we manage portability and configuration with the C preprocessor. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM)*, Paris, France, October 2007.

[222] Swag Kit. http://www.swag.uwaterloo.ca/swagkit/instructions.html.

[223] SWI Prolog. http://www.swi-prolog.org/.

[224] Clemens Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[225] Peri Tarr, William Chung, William Harrison, Vincent Kruskal, Harold Ossher, Jr. Stanley M. Sutton, Andrew Clement, Matthew Chapman, Helen Hawkins, and Sian January. The concern manipulation environment. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 29–30, Vancouver, BC, Canada, 2004. ACM Press.

[226] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 107–119, Los Angeles, CA, USA, 1999. IEEE Computer Society Press.

[227] Walter F. Tichy. Software development control based on module interconnection. In *Proceedings of the 4th international conference on Software engineering (ICSE)*, pages 29–41, Munich, Germany, 1979. IEEE Press.

[228] Qiang Tu and Michael W. Godfrey. The build-time software architecture view. In *Proceedings of the 17th International Conference on Software Maintenance (ICSM)*, pages 398–407, Florence, Italy, 2001.

[229] Tijs van der Storm. Variability and component composition. *Software Reuse: Methods, Techniques and Tools*, pages 157–166, 2004.

[230] Tijs van der Storm. Continuous release and upgrade of component-based software. In *Proceedings of the 12th international workshop on Software configuration management (SCM)*, pages 43–57, Lisbon, Portugal, 2005. ACM.

[231] Tijs van der Storm. Lightweight incremental application upgrade. Technical Report SEN-R0604, SEN, CWI, April 2006.

[232] Arie van Deursen and Merijn de Jonge. Product line evolution using source packages. Unpublished?

[233] Arie van Deursen, Merijn de Jonge, and Tobias Kuipers. Feature-based product line instantiation using source-level packages. In *Proceedings of the Second International Conference on Software Product Lines (SPLC)*, pages 217–234, London, UK, 2002. Springer-Verlag.

[234] Gary V. Vaughan, , Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake and Libtool*. New Riders Publishing, Thousand Oaks, CA, USA, 2000.

[235] László Vidács, Árpád Beszédes, and Rudolf Ference. Columbus schema for C/C++ preprocessing. In *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR)*, page 75, Tampere, Finland, 2004. IEEE Computer Society.

[236] Marian Vittek. Refactoring browser with preprocessor. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, page 101, Benevento, Italy, 2003. IEEE Computer Society.

[237] Kris De Volder and Theo D'Hondt. Aspect-Oriented Logic Meta Programming. In *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (REFLECTION)*, pages 250–272, London, UK, 1999. Springer-Verlag.

[238] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 1–14, Albuquerque, NM, USA, 1992. ACM.

[239] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering (FSE)*, pages 159–169, Newport Beach, CA, USA, 2004. ACM.

[240] Jean-Paul Van Waveren. Quake III Arena bot. Master's thesis, TU Delft, Delft, The Netherlands, June 2001.

[241] David A. Wheeler. sloccount. http://www.dwheeler.com/sloccount/.

[242] Niklaus Wirth. Modula-2 and Oberon. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL)*, pages 3–1–3–10, San Diego, CA, USA, 2007. ACM.

[243] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, Pleinlaan 2, Etterbeek, 2001.

[244] Yoshisato Yanagisawa, Kenichi Kourai, and Shigeru Chiba. A dynamic aspect-oriented system for OS kernels. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 69–78, Portland, OR, USA, October 2006. ACM.

[245] Marco Yuen, Marc Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. Making extensibility of system software practical with the C4 toolkit. In *Proceedings of the 4th Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD*, Bonn, Germany, March 2006.

[246] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 134–142, Manchester, UK, 2005. IEEE.

[247] Andy Zaidman and Serge Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–338, Tampere, Finland, 2004.

[248] Andy Zaidman, Serge Demeyer, Bram Adams, Kris De Schutter, Ghislain Hoffman, and Bernard De Ruyck. Regaining lost knowledge through dynamic analysis and aspect orientation. In *Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR)*, pages 91–102, Bari, Italy, March 2006. IEEE Computer Society.

[249] Charles Zhang and Hans-Arno Jacobsen. TinyC^2:towards building a dynamic weaving aspect language for C. In *Proceedings of the Workshop on Foundations Of Aspect-Oriented Languages (FOAL)*, Boston, MA, USA, 2003.