UNIVERSITEIT
GENT

Faculteit Wetenschappen

Multi-Formalism Modelling and Simulation

Hans Vangheluwe

Proefschrift voorgedragen tot het behalen
van het doctoraat in de wetenschappen

Promotoren: Prof. Dr. A. Hoogewijs
Prof. Dr. ir. G.C. Vansteenkiste

Academiejaar 2000 – 2001

— *The road goes ever on and on*
Down from the door where it began.
Now far ahead the road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then ? I cannot say.

Bilbo Baggins in “The Lord of The Rings”

Acknowledgements

In the never ending quest for knowledge and insight which research is (in particular the production of this thesis), I have been fortunate to have learned from and been helped, guided, and stimulated by many people.

First of all, my parents have made all this possible. Without their endless support, I would never have reached this point. The last sprint to the end of this thesis was only possible thanks to the devoted help of Indrani, my wife.

I have of course been moulded by all my teachers. Prof. Grosjean, my physics thesis promotor, in particular has shown me the beauty of scientific research. The promotors of this work, Profs. Vansteenkiste and Hoogewijs have shown infinite trust in my capabilities. Thanks to Prof. Vansteenkiste, I had the opportunity to develop the multi-formalism direction of my research. Though not officially so, Eugène Kerckhoffs has been my supervisor for many years. I am grateful for so many intense technical discussions. The fruitful collaboration with the TU Delft is certainly also due to Prof. Henk Koppelaar.

At BIOMATH, the collaborators and students over the years are too numerous to list. Annie Gevaert's secretarial support has extended beyond the call of duty (including taxi service). It has been an instructive pleasure to attend the SCS conferences organized by Philippe Geril. WEST++ has had a long history. It has been a pleasure to work on its various incarnations with Bart Derauw, Jonathan Levine, Pedro Lopez, Bas Kops and Indrani A.V. The two pillars on which WEST++ is built are of course Filip Claeys and Bhama Sridharan. They have kept up with my wild ideas and I have enjoyed every minute of working with them. The waste water knowledge for WEST++ was initially provided by Bart Vanderhaegen and Peter Vanrolleghem. Later, Henk Vanhooren's help was invaluable during the design of the final model base structure. WEST++ would not exist without the sponsorship by Dirk Van der Stede (Hemmis). I enjoyed working with Dirk Stevens of Hemmis on WEST, the pre-cursor of WEST++. I am certain my nonchalance with respect to deliverables often drove him to despair.

Not only WEST++, but my whole research career has been generously sponsored by Ghent University, the Belgian Prime Minister's Office – Science Policy Programming, the IWT, FWO Vlaanderen, the ESPRIT programme of the European Union, Hemmis, Aquafin, and the Society for Computer Simulation Europe.

Zhiguo Yuan was an endless source of control engineering knowledge. All the members of the Simulation in Europe working group provided a stimulating medium for intense discussions on modelling and simulation. Françis Lorenz and Jacques Lefèvre in particular were wonderful intellectual sparring partners. In the Modelica group, Europe's experts in modelling language design were brought together and I drank up their wisdom. The discussions with Thilo Ernst, Jan Broenink, and Pieter Mosterman were (and are still) particularly invigourating. The guest researchers at BIOMATH, Mohamed Khamis, Mohamed Masmoudi, Richard Fryer, John Zenor and François Cellier taught me about the world as well as about science.

I am grateful to Prof. Renato de Mori for inviting me to the Centre de Recherche Informatique de Montréal (CRIM) and McGill University in 1991. Prof. Laurie Hendren's compiler project was a challenge I enjoyed tremendously. My times in Montréal were (and are) scientifically and otherwise agreeable thanks to friends Michael Galler, Chandrika Mukerji, Larry Baer, Jonathan Levine and Justin Bur. Prof. Ramana Reddy generously invited me to the Concurrent Engineering Research Center, Morgantown, West Virginia, twice. I lapped up knowledge, in particular during discussions with Srinivas Kankanahalli. During my 1996 stay at CERC, discussions as well as "too much fun" with fellow researchers Andrea d'Ambrogio, Carlos Vila, Kouji Okada, Igor Lapshin and Roy Lawson were unforgettable. Director Pichot of the Management Unit of Mathemathical Modelling of the North Sea and the Scheldt Estuary (MUMM) was kind enough to draft me from the Navy. Thanks to Commander Grandfils for letting MUMM "borrow" me. In Ostend, the operational head of MUMM and coordinator of the Belgica research vessel campaigns, André Pollentier gave me the opportunity to work on muSTORM and participate in Belgica campaigns. At the Brussels headquarters the almost daily discussions with Jean-Pierre Malisse and Kevin Ruddick broadened my scientific mind. Prof. N. Balakrishnan gave me the opportunity to work and lecture at the Supercomputing Education Research Centre (SERC) of that wonderful research heaven, the Indian Institute of Science in Bangalore, India. Prof.

R. Govindarajan was my ever-helpful coach. Hank Dorris, who I met in Washington in ’97 believed in and supported my Generic Modelling and Simulation Architecture (GMSA) ideas with vigour and vision.

The modelling and simulation community would not be what it is today without the brilliant ideas of Bernie Zeigler and François Cellier and the vision of John McLeod, founder of the Society for Computer Simulation.

Hans L.M. Vangheluwe

Gent, December 19, 2000.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

This thesis develops a *multi-formalism* modelling and simulation methodology for complex systems. Its primary contribution is the *unification* of different modelling and simulation concepts, methods and techniques. To complement the theoretical concerns, computer implementation issues are dealt with, and the study of activated sludge Waste Water Treatment Plants (WWTPs) by means of modelling and simulation is presented as a concrete application. A number of model (inter-formalism) *compilers* as well as the full-fledged *distributed, interactive modelling and simulation environment WEST++* have been developed. In this connection, a crucial part of WEST++ is the *declarative modelling language MSL-USER*.

In the following section, these contributions are placed in the appropriate *context*.

The Modelling and Simulation context

It is in human nature to want to understand dynamic systems, control them, and above all predict their future behaviour.

During the last century, this desire has lead to inter-disciplinary research into modelling and simulation, bringing together results from mathematics, computer science, cognitive sciences, and a variety of application-domain-specific research. Modelling covers the understanding and represention of structure and behaviour at an abstract level, whereas simulation produces behaviour as a function of time based on an abstract model and initial conditions.

From the 1950s, the focus of research was on *efficient* and *accurate* numerical simulations (and hardly on modelling). This originated from results in numerical analysis (for differential equation models), and in random number generation and statistical analysis (for discrete event models). This research led, at the end of the 1960s to the establishment of simulation-oriented standards such as the Continuous System Simulation Language (CSSL) [SAF$^+$67] and the discrete event world views, both of which are still in common use today and will be described and elaborated upon in this thesis.

From the 1980s, evolutions in cognitive science and Artificial Intelligence (AI) amplified the idea of an abstract model (described in some modelling formalism) as a form of knowledge representation. AI techniques, most notably expert systems for choosing optimal models and simulators for a given problem, were used in modelling and simulation. Conversely, AI called upon explicit numerical simulation to obtain "deep" knowledge in the form of detailed behaviour trajectories [VVVW$^+$91, VKVWV92]. This, as opposed to qualitative simulation of Kuipers [Kui86, Kui88, Kui94], de Kleer [dKB84], and Forbus [For84, For90], which tried to chart classes of behaviours, and thus predict multiple possible futures, by ignoring quantitative details.

With the advent of the object-oriented methodology for software design at the end of the 1980s, object-oriented sofware found its way into modelling and simulation. This is not surprising, as the alleged root of object-oriented programming languages is Simula, a simulation language [Weg90]. As will be discussed in this text, object-oriented modelling and object-oriented simulation (implementation) are two distinct concepts.

Since the 1990s, AI research has again called upon modelling and (object-oriented) simulation using individual-based simulations to investigate the emergent behaviour of communities of interacting "agents" [WJ95, Ode98, Mae94, vdCKV96].

Today, modelling and simulation are increasingly recognized as a separate inter-disciplinary research area distinct from Computer Science, Mathematics, AI, *etc.* with a vast range of applications. As the constraints

on product production costs as well as demands for predictable (often, higher) quality increase, the need for "doing it right the first time" is felt. If iterative development and improvement is not feasible in the physical world, repeated *virtual experiments* by means of simulation is a valid approach. One area where modelling and simulation is getting a strong foothold is the design of software. In particular, software engineering needs modelling and simulation techniques to tackle hardware/software co-design, real-time (due to the presence of time constraints), and software process problems.

In this thesis, some of the needs currently felt in the modelling and simulation community are addressed. These needs were identified by ESPRIT's SiE-WG, of which the author was a co-founder and co-ordinator. The European Strategic Programme in Information Technology (ESPRIT) Basic Research Working Group 8467 [VKV96, VV96d, KVVG94, KVVG95] on "Simulation for the Future: new concepts, tools and applications", with the acronym SiE-WG (Simulation in Europe - Working Group), started its work on December 1, 1993. SiE-WG was an initiative of the SiE Special Interest Group (SiE-SIG), currently consisting of some 200 industrial and academic members. The SiE-SIG acts as a platform and validating forum for SiE-WG results. The –now concluded– SiE-WG activities will be resumed in 2001. Needs of particular relevance to European industry and to the end-user were identified. "Simulation Research Policy Guidelines" were formulated by SiE-WG:

1. Improve the modelling and simulation process:
   (a) Modelling:
   Redefine "modelling" in a broader perspective than currently used and exploit this as a basis for new modelling and simulation methodologies (*i.e., multi-formalism modelling*, named multi-paradigm modelling, at the time of the report writing).
   Focus on generic, object-oriented, component modelling and supporting representations to enhance re-usability and portability of existing and new simulation models.
   (b) Techniques:
   Adapt Software Engineering and Artificial Intelligence methods and tools (*e.g.*, formal verification, re-use, version management and decision support) to modelling and simulation problems. Merge results in integrated methods and tools (e.g., *multi-language software systems*).
   (c) Life-cycle:
   Pay attention to the full Modelling/Simulation Experimentation/Validation life-cycle. Use explicit descriptions and prescriptions for this (possibly concurrent) life-cycle to improve quality of the end-products (software and/or hardware).
2. Open new application areas:
   (a) Include new peripheral devices and novel algorithms into simulators. Enter new application areas (for example, the medical sector).
   (b) Exploit *highly parallel* hardware architectures to simulate *multi-component* systems by directly mapping model structure onto hardware structure.
3. Provide user-simulator interfaces:
   (a) Provide a common basis for independent development of simulators and user-interfaces by means of Open Systems.
   (b) Intelligent user-simulator *interfaces* should present multiple interaction scenarios for simulation information (*e.g.*, education by simulation, assisted statistical *interpretation*). "User Centred" interfaces necessitate integration of both engineering and human science models.
4. Enhance awareness (through knowledge dissemination):
   (a) Provide *education* in Modelling and Simulation to remedy the skill shortage in this field. The education (both in universities and on-site) must be tailored to the end-user needs.

(b) Disseminate information about simulation as well as standardized tools to current and potential simulation beneficiaries.

5. Prepare standards and standardization procedures. Support flexibility in design and re-usability of models by developing general formats for the information base of models in different application areas.

The gaps in current modelling and simulation practice are believed to be due to:

- the use of simulation tools without formal underpinning. As many tools are application-specific, concepts such as non-causal modelling are re-discovered independently in different application domains and often solved in a far from optimal fashion;
- the lack of a clear distinction between modelling and simulation. There is no common understanding of these concepts;
- the legacy of techniques and tools such as causal modelling in Simulink [SIM97] and ACSL [ACS95] dating back to the 1970s. Though these are excellent simulation tools, the expressiveness of non-causal modelling allows for better model re-use. Non-causal modelling does not preclude the use of these tools as symbolic manipulation can convert non-causal models into causal ones;
- the lack of integration of different formalisms. Either a modeller is forced to cast his model into a single formalism for which a tool is available or multi-formalism models are simulated by connecting formalism-specific simulators for each of the components. The latter is inefficient and a vast amount of semantic information is lost which could be used for optimization, insight, validation, automatic parallellization*, etc.*

Contributions

This thesis addresses items 1, 3, and 5 of the aforementioned needs of the modelling and simulation community. It *unifies* different approaches in a *multi-formalism* modelling and simulation methodology. The work generalizes Zeigler's general Theory of Modelling and Simulation [Zei84b, ZPK00] by introducing non-causal modelling and multi-formalism modelling. The developments are made possible through the application of state-of-the-art computer science techniques:

- some results from Artificial Intelligence for automating model selection;
- object-oriented software design for the construction of a generic modelling and simulation architecture;
- graph theory to transform continuous models based on manipulation of a dependency graph of variables occurring in algebraic equations;
- computer algebra to re-write algebraic expressions;
- numerical analysis for the implementation of efficient solvers for ordinary differential equations;
- compiler compilers to generate model compilers from high-level specifications.

We now present a brief overview of the thesis, pointing out original contributions.

In the first chapter, general modelling and simulation concepts such as verification and validation are introduced. In particular, a new, unified representation of modelling errors is introduced. The presentation brings structure to the multitude of different existing views, and introduces original *classifications* of modelling formalisms. As not only the formalism a model is represented in is important, but also the process of manipulating this model, models of the modelling and simulation process are proposed. Such models may be used not only to describe the process, but also to prescribe the process, and as such form the basis for automated modelling and simulation environments. The main contribution here is the Virtual Product Life-cycle (VPL) concept and a recursive process model for modelling and simulation, supporting bottom-up as well as top-down construction and use of models.

In the second chapter, a rigourous presentation is given of diverse formalisms with a description of possible transformations between these formalisms. These transformations are at the basis of the implementation of model compilers. The transformations include

- transformation of event scheduling discrete event models to DEVS,
- transformation of cellular automata to DEVS,
- symbolic transformation of continuous formalisms based on graph algorithms,
- transformation between differential equations and transfer functions,
- transformation of Forrester's System Dynamics models to ordinary differential equations,
- transformation of a commonly used class of partial differential equations to ordinary differential equations based on orthogonal collocation over finite elements.

As may be apparent from the above, the described formalisms are: I/O data trajectories, discrete event formalisms in the form of four different "world views", the DEVS formalism, Cellular Automata, differential and algebraic equation formalisms. Also introduced are non-causal models, how they enable model re-use and how to process them efficiently, the transfer function formalism, Forrester's System Dynamics formalism and a class of partial differential equations.

This chapter concludes by presenting an algorithm for "flattening" coupled multi-formalism models (modular networks). This is achieved by transforming the components to a common formalism. Knowledge about which transformations are feasible is retained in a Formalism Transformation Graph (FTG). The introduction of multi-formalism modelling, the FTG and the flattening algorithm forms a major contribution to the meaningful modelling of complex systems.

Based on the discussion in the first two chapters, the third chapter contains the design of a declarative Model Specification Language (MSL-USER) as well as a full, interactive modelling and simulation environment (WEST++) to create, modify, and simulate MSL-USER models. The design requirements for MSL-USER were genericity, support for re-use and exchange, as well as for the representation of multiple formalisms. MSL-USER concepts are now used in the standardization effort Modelica [EBB+99]. MSL-USER is a first step towards meta-modelling, also described in this chapter.

In the fourth and final chapter, it is shown how the modelling and simulation concepts developed in this thesis can be applied in the domain of bio-activated sludge waste water treatment. In particular, a methodology for constructing model bases for physical systems is presented and illustrated for Waste Water Treatment Plants (WWTPs). This leads to a generic model base for WWTP modelling. Finally, the application of WWTP models in the WEST++ tool is illustrated.

1

Modelling and Simulation Concepts

At a first glance, it is not easy to characterize modelling and simulation. Certainly, a variety of application domains such as fluid dynamics, energy systems, and logistics management make use of it in one form or another. Depending on the context, modelling and simulation is often seen as a sub-set of Systems Theory, Control Theory, Numerical Analysis, Computer Science, Artificial Intelligence, or Operations Research. Increasingly, modelling and simulation *integrates* all of the above disciplines. Recently, modelling and simulation has been slated to become the computing *paradigm* of the future. As a paradigm, it is a way of representing problems and thinking about them, as much as a solution method. The problems span the analysis and design of complex dynamical systems. In analysis, abstract models are built inductively from observations of a real system. In design, models deductively derived from a priori knowledge are used to build a system, satisfying certain design goals. Often, an iterative combination of analysis and design is needed to solve real problems. Though the focus of modelling and simulation is on the behaviour of dynamical (*i.e.*, time-varying) systems, static systems (such as entity-relationship models, described in the Unified Modelling Language UML [RJB99]) are a limit-case. Both physical (obeying conservation and constraint laws) and non-physical (informational, such as software) systems and their interactions are studied by means of modelling and simulation.

1.1 Basic concepts

In the following, an introduction to the basic concepts of modelling and simulation is given.

Figure 1.1 presents modelling and simulation concepts as introduced by Zeigler [Zei84b, ZPK00].

Object is some entity in the Real World. Such an object can exhibit widely varying behaviour depending on the context in which it is studied, as well as the aspects of its behaviour which are under study.

Base Model is a hypothetical, abstract representation of the object's properties, in particular, its behaviour, which is valid in *all* possible contexts, and describes all the object's facets. A base model is hypothetical as we will never —in practice— be able to construct/represent such a "total" model. The question whether a base model exists at all is a philosophical one (akin to the hidden variable problem in physics).

System is a well defined object in the Real World under specific conditions, only considering specific aspects of its structure and behaviour.

Experimental Frame When one studies a system in the real world, the experimental frame (EF) describes experimental conditions (context), aspects, ... within which that system and corresponding models

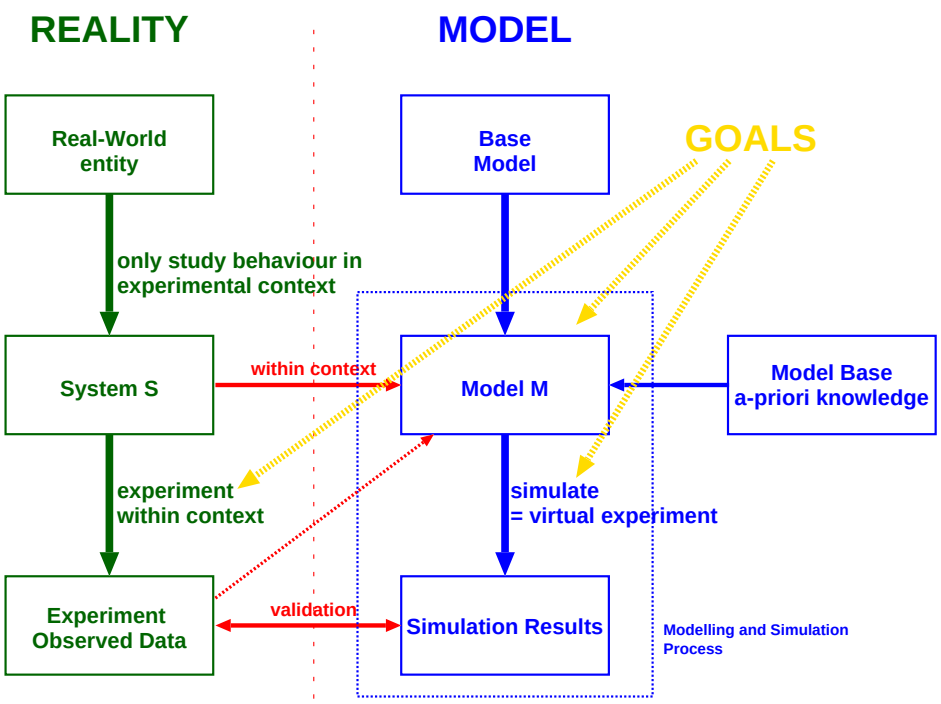

Figure 1.1: Modelling and Simulation

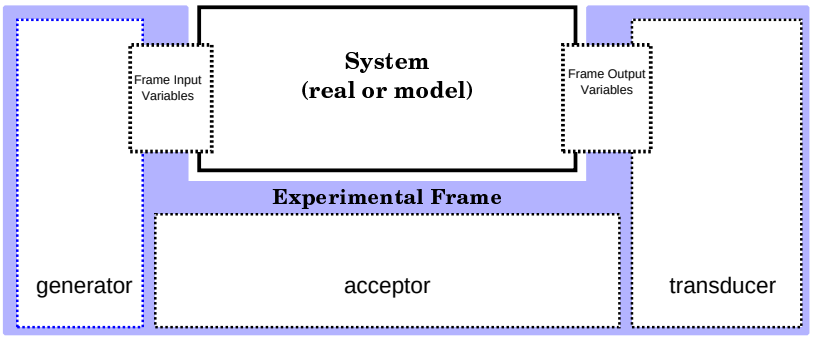

Figure 1.2: System versus Experimental Frame

will be used. As such, the Experimental Frame reflects the *objectives* of the experimenter who performs experiments on a real system or, through simulation, on a model. In its most basic form (see Figure 1.2), an Experimental Frame consists of two sets of variables, the Frame Input Variables and the Frame Output Variables, which match the system or model terminals. On the input variable side, a *generator* describes the inputs or stimuli applied to the system or model during an experiment. A generator may for example specify a unit step stimulus. On the output variable side, a *transducer* describes the transformations to be applied to the system (experiment) or model (simulation) outputs for meaningful interpretation. A transducer may for example specify the calculation of the extremal values of some of the output variables. In the above, *output* refers to physical system output as well as to the synthetic outputs in the form of internal model states measured by an observer. In case of a model, outputs may *observe* internal information such as state variables or parameters. Apart from input/output variables, a generator and a transducer, an Experimental Frame may also comprise an *acceptor* which compares features of the generator inputs with features of the transduced output, and determines whether the system (real or model) “fits” this Experimental Frame, and hence, the experimenter’s objectives.

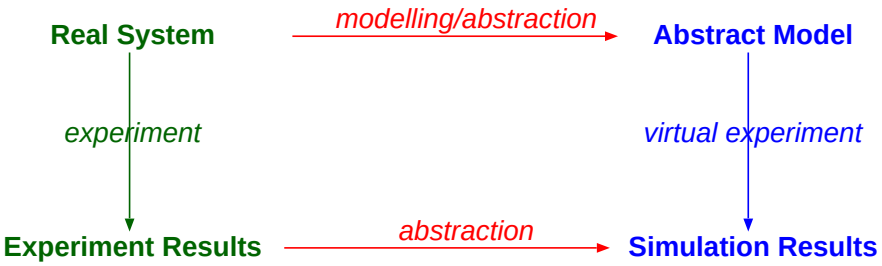

Figure 1.3: Modelling – Simulation Morphism

(Lumped) Model gives an accurate description of a system within the context of a given Experimental Frame. The term "accurate description" needs to be defined precisely. Usually, certain properties of the system's structure and/or behaviour must be reflected by the model within a certain range of accuracy. Note: a lumped model is not necessarily a lumped parameter model [Cel91]. Due to the diverse applications of modelling and simulation, terminology overlap is very common.

Experimentation is the physical act of carrying out an experiment. An experiment may interfere with system operation (influence its input and parameters) or it may not. As such, the experimentation environment may be seen as a system in its own right (which may in turn be modelled by a lumped model). Also, experimentation involves observation. Observation yields *measurements.*

Simulation of a lumped model described in a certain formalism (such as Petri Net, Differential Algebraic Equations (DAE) or Bond Graph) produces *simulation results*: the dynamic input/output behaviour. Simulation may use *symbolic* as well as *numerical* techniques. Simulation, which mimics the real-world experiment, can be seen as *virtual experimentation,* allowing one to *answer questions* about (the behaviour of) a system. As such, the particular technique used does not matter. Whereas the goal of modelling is to *meaningfully describe* a system presenting information in an understandable, re-usable way, the aim of simulation is to be *fast and accurate*. Symbolic techniques are often favoured over numerical ones as they allow the generation of classes of solutions rather than just a single one. For example, $sin(x)$ as a solution to the harmonic equation is preferred over one single approximate trajectory solution. Furthermore, symbolic optimizations have a much larger impact than numerical ones thanks to their global nature. Crucial to the System–Experiment/Model–Virtual Experiment scheme is that there is a *homomorphic* relation between model and system: building a model of a real system and subsequently simulating its behaviour should yield the same results as performing a real experiment followed by observation and codifying the experimental results (see Figure 1.3). A simulation model is a tool for achieving a goal (design, analysis, control, optimisation, ...) [BO96]. A fundamental prerequisite is therefore some assurance that inferences drawn from modelling and simulation (tools) can be accepted with *confidence.* The establishment of this confidence is associated with two distinct activities; namely, verification and validation.

Verification is the process of checking the consistency of a simulation program with respect to the lumped model it is derived from. More explicitly, verification is concerned with the correctness of the transformation from some intermediate abstract representation (the conceptual model) to the program code (the simulation model) ensuring that the program code faithfully reflects the behaviour that is implicit in the specification of the conceptual model.

Validation is the process of comparing experiment *measurements* with *simulation results* within the context of a certain Experimental Frame [Bal97a]. When comparison shows differences, the formal model built may not correspond to the real system. A large number of matching *measurements* and *simulation results*, though increasing confidence, does *not prove* validity of the model however. For this reason, Popper has introduced the concept of *falsification* [Mag85], the enterprise of trying to falsify or disprove a model. Various kinds of validation can be identified; *e.g.*, conceptual model validation,

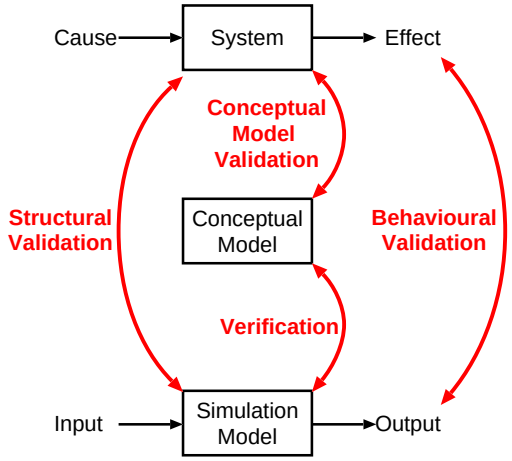

Figure 1.4: Verification and validation activities

structural validation, and behavioural validation. *Conceptual validation* is the evaluation of a conceptual model with respect to the system, where the objective is primarily to evaluate the realism of the conceptual model with respect to the goals of the study. *Structural validation* is the evaluation of the structure of a simulation model with respect to perceived structure of the system. *Behavioural validation* is the evaluation of the simulation model behaviour. An overview of verification and validation activities is shown in Figure 1.4. It is noted that the correspondence in generated behaviour between a system and a model will only hold within the limited *context* of the Experimental Frame. Consequently, when using models to exchange information, a model must always be matched with an Experimental Frame before use. Conversely, a model should never be developed without simultaneously developing its Experimental Frame. This requirement has its repercussions on the design of a model representation language.

1.2 The modelling and simulation process

To understand any enterprise, it is necessary to analyze the *process*: which activities are preformed, what entities are operated on, and what the causal relationships (determining activity order and concurrency) are. A *described* process gives insight, a *prescribed* process can be the basis for automation and implementation of a software tool [Hum89, HK89]. Note how a prescribed process is not necessarily deterministic as it may still leave a large number of decisions to the user. The importance of studying processes is exemplified by the SEI Capability Maturity Model (`http://www.sei.cmu.edu/cmm/cmms/cmms.html`) which assesses the quality of software companies by the level of knowledge, re-use, and optimization of their processes.
The simulation activity is part of the larger model-based systems analysis enterprise. A rudimentary process model for these activities is depicted in Figure 1.5. By means of a simple mass-spring experiment example (see Figure 1.6), the process will be explained. In this example, a mass sliding without friction over a horizontal surface is connected to a wall via a spring. The mass is pulled away from the rest position and let go.

A number of *Information Sources* (either *explicit* in the form of data/model/knowledge bases or *implicit* in the user's mind) are used during the process:

1. A Priori Knowledge: in *deductive modelling*, one starts from general principles –such as mass, energy, momentum conservation laws and constraints– and deduces specific information. Deduction is predominantly used during system *design*. In the example, the a priori knowledge consists of Newton's second law of motion, as well as our knowledge about the behaviour of an ideal spring.

2. Goals and Intentions: the level of abstraction, formalisms used, methods employed, ... are all determined by the type of *questions* we want to answer. In the example, possible questions are: "what is a

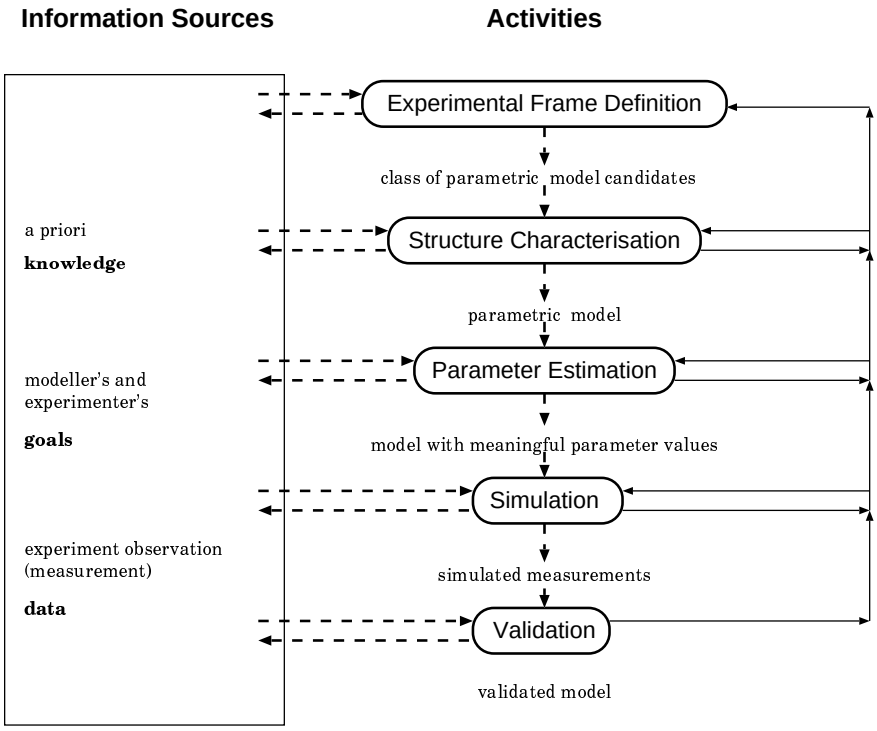

Figure 1.5: Model-based systems analysis

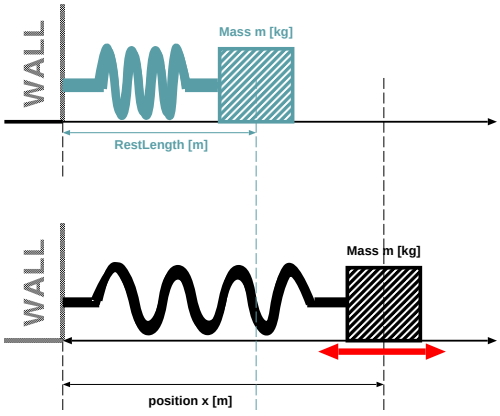

Figure 1.6: Mass-Spring example

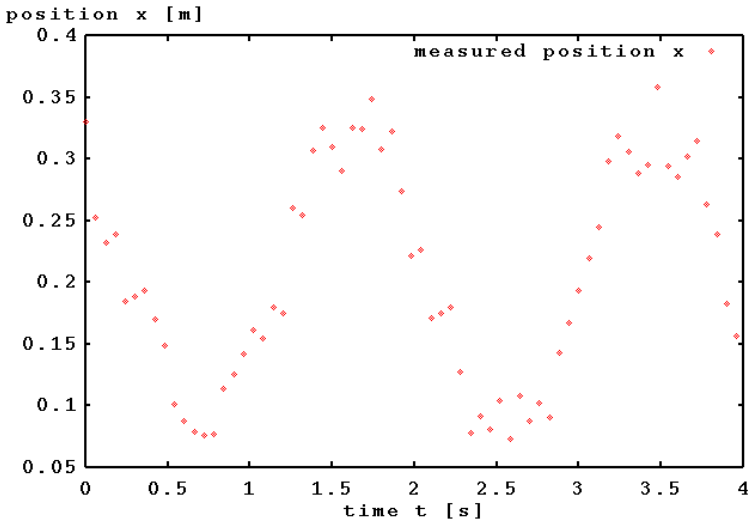

Figure 1.7: Measurement data

suitable model for the behaviour of a spring for which we have position measurements ?", "what is the spring constant ?", "given a suitable model and initial conditions, predict the spring's behaviour", "how to build an optimal spring given performance criteria ?", ...

3. Measurement data: in *inductive modelling*, we start from data and try to extract structure from it. This structure/model can subsequently be used in a deductive fashion. Such *iterative* progression is typical in systems *analysis*. Figure 1.7 plots the noisy measured position of the example's mass as a function of time.

The process starts by identifying an Experimental Frame. As mentioned above, the frame represents the experimental conditions under which the modeller wants to investigate the system. As such, it reflects the modeller's goals and questions. In its most general form, it consists of a *generator* describing possible inputs to the system, a *transducer* describing the output processing (*e.g.*, calculating performance measures integrating over the output), and an *acceptor* describing the conditions (logical expressions) under which the system (be it real or modelled) match. In the example, the experimental frame might specify that the position deviation of the mass from the rest position will/may never be larger than the rest length of the spring. Environment factors such as room temperature and humidity could also be specified, if relevant. Based on a frame, a class of matching models can be identified.

Through structure characterization, the appropriate model structure is selected based on a priori knowledge and measurement data. In the example, a *feature* of an ideal spring (connected to a frictionless mass) is that the position amplitude stays constant. In a non-ideal spring, or in the presence of friction, the amplitude descreases with time. Based on the measured data, we conclude this must be an ideal spring.

A suitable model as shown below can be built. Note how the model is non-causal (not specifying which variables are known and which need to be computed) and contains an assertion encoding the Experimental Frame acceptor.

```
CLASS Spring "Ideal Spring": DAEmodel :=
{
 OBJ F_left: ForceTerminal,
 OBJ F_right: ForceTerminal,

 OBJ RestLength: LengthParameter,
 OBJ SpringConstant: SCParameter,

 OBJ x: LengthState,
 OBJ v: SpeedState,
```

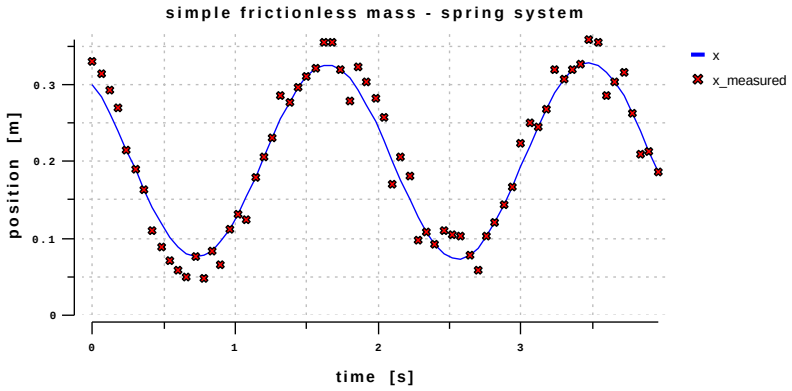

Figure 1.8: Fitted simulation results

```
  F_left - F_right = - SpringConstant *
                       (x - RestLength),
  DERIV([ x, [t,] ]) = v,

  EF_assert( x - RestLenght < RestLength/100),
},
```

Subsequently, during model calibration, parameter estimation yields optimal parameter values for reproducing a set of measurement data. From the model, a simulator is built. Due to the *contradicting aims* of modelling –meaningful model representation for *understanding and re-use*– and simulation –*accuracy and speed*–, a large number of steps may have to be traversed to bridge the gap. Using the identified model and parameters, simulation allows one to mimic the system behavior (virtual experimentation) as shown in Figure 1.8. The simulator thus obtained can be embedded in for example, an optimizer, a trainer, or a tutoring tool.

The question remains whether the model has predictive validity: is it capable not only of reproducing data which was used to choose the model and to identify parameters but also of predicting new behavior ? With every use of the simulator, this validity question must be asked. The user determines whether validation is included in the process. In a flight simulator, one expects the model to have been validated. In a tutor, validation by the user may be part of the education process.

In Figure 1.5, one notices how each step in the modelling process may introduce errors. As indicated by the feedback arrows, a model has to be corrected once falsified. A desirable feature of the validation process is the ability to provide hints as to the location of modelling errors [YVV98]. Unfortunately however, very few methods are designed to systematically provide such information. In practical use, the process is refined and embedded in more general (*e.g.*, tutoring, training, optimal experimental design, control) processes.

1.3 Verification and validation

The presentation of an experimental frame given above enables a rigourous definition of model *validity*. Let us first postulate the existence of a unique *Base Model*. This model is assumed to accurately represent the behavior of the Real System under *all* possible experimental conditions. This model is *universally valid* as the data $D_{RealSystem}$ obtainable from the Real System is always equal to the data $D_{BaseModel}$ obtainable from the model.

$$D_{BaseModel} \equiv D_{RealSystem}$$

A Base Model is distinguished from a *Lumped Model* by the limited experimental context within which the last accurately represents Real System behavior.

A particular experimental frame E may be applicable to a real system or to a model. In the first case, the data

potentially obtainable within the context of E are denoted by $D_{RealSystem}\|E$. In the second case, obtainable data are denote by $D_{model}\|E$. With this notation, a model is valid for a real system within Experimental Frame E if

$$D_{LumpedModel}\|E \equiv D_{RealSystem}\|E$$

The data equality $\equiv$ must be interpreted as "equal to a certain degree of accuracy".
The above shows how the concept of validity is *not absolute*, but is related to the experimental *context* within which Model and Real System *behavior* are compared and to the *accuracy metric* used.

One typically distinguishes between the following types of model validity:

Replicative Validity concerns the ability of the Lumped Model to *replicate* the input/output data of the Real System. With the definition of a Base Model, a Lumped Model is replicatively valid in Experimental Frame E for a Real System if

$$D_{LumpedModel}\|E \equiv D_{BaseModel}\|E$$

Predictive Validity concerns the ability to identify the *state* a model should be set into to allow *prediction* of the response of the Real System to *any* (not only the ones used to identify the model) input segment. A Lumped Model is predictively valid in Experimental Frame E for a Real System if it is replicatively valid and

$$F_{LumpedModel}\|E \subseteq F_{BaseModel}\|E$$

where F_S is the set of I/O functions of system S within Experimental Frame E. An I/O function identifies a *functional relationship* between Input and Output, as opposed to a general non-functional *relation* in the case of replicative validity.

Structural Validity concerns the *structural relationship* between the Real System and the Lumped Model. A Lumped Model is structurally valid in Experimental Frame E for a Real System if it is predictively valid and there exists a morphism $\stackrel{\triangle}{=}$ from Base Model to Lumped Model within frame E.

$$LumpedModel\|E \stackrel{\triangle}{=} BaseModel\|E$$

When trying to assess model validity, one must bear in mind that one only observes, at any time t, $D^t_{RealSystem}$, a subset of the potentially observable data $D_{RealSystem}$. This obviously does not simplify the model validation enterprise.
Whereas assessing model validity is intrinsically impossible, the *verification* of a *model implementation* can be done rigorously. A *simulator* implements a lumped model and is thus a source of obtainable data $D_{Simulator}$. If it is possible to prove (often by design) a structural realtionship (morphism) between Lumped model and Simulator, the following will hold unconditionally

$$D_{Simulator} \equiv D_{LumpedModel}$$

Before we go deeper into predictive validity, the relationship between different *refinements* of both Experimental Frames and models is elaborated. In Figure 1.9, the *derived from* relationship for Experimental Frames and the *homomorphism* relationship for Models are depicted. If we think of an Experimental Frame as a formal representation of the "context within which the model is a valid representation of the dynamics of the system", a more restricted Experimental Frame means a more specific behaviour. It is obvious that such a restricted Experimental Frame will "match" far more models than a more general Experimental Frame. Few models are elaborate enough to be valid in a very general input/parameter/performance range. Hence, the large number of "applies to" (*i.e.*, match) lines emanating from a restricted Experimental Frame. The homomorphism between models means that, when modifying/transforming a model (*e.g.*, adding some non-linear term to a model), the simulation results (*i.e.*, the behaviour) within the same experimental frame must remain the same.

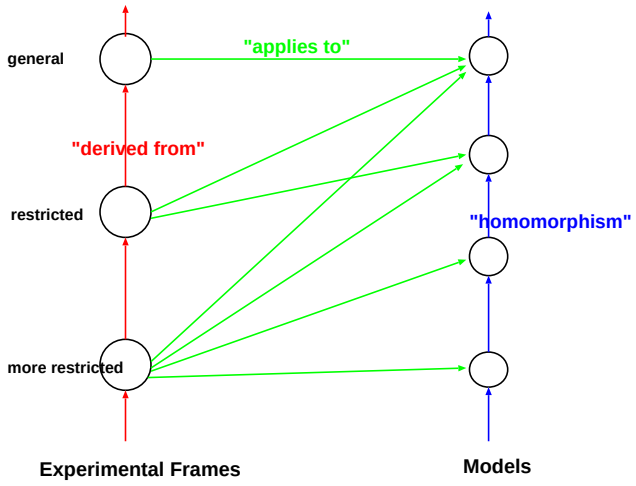

Figure 1.9: Experimental Frame – model relationship

Though it is meaningful to keep the above in mind during model development and use, the highly non-linear nature of many continuous models (as used in WEST++) makes it very difficult to "automate" the management of information depicted in Figure 1.9. Non-linear behaviour makes it almost impossible, based on a model or experimental frame symbolic representation, to make a statement about the area in state-space which will be covered (*i.e.*, behaviour). A pragmatic approach is to

1. let an "expert" indicate what the different relations are. This is based on some "insight" into the non-linear dynamics. Such expert knowledge can be built from a large number of conducted experiments.

2. constantly –with each experiment– validate the expert information.

A crucial question is whether a model has predictive validity: is it capable not only of reproducing data which was used to choose the model and parameters but also of predicting new behavior? The predictive validity of a model is usually substantiated by comparing new experimental data sets to those produced by simulation, an activity known as model validation. Due to its special importance in the communication between model builders and users, model validation has received considerable attention in the past few decades (for a survey, see for example [Bal97a]. Problems from general validation methodologies to concrete testing technologies have been extensively studied. The comparison of the experimental and simulation data are accomplished either subjectively, such as through graphical comparison, Turing test, or statistically, such as through analysis of the mean and variance of the residual signal employing the standard F statistics, Hotelling's T^2 tests, multivariate analysis of variance regression analysis, spectral analysis, autoregressive analysis, autocorrelation function testing, error analysis, and some non-parametric methods. An excellent presentation of the different issues as well as a classification of verification, validation, and testing techniques is given by Balci in [Bal97a].

As indicated by the feedback arrows in Figure 1.5, a model has to be corrected once proven invalid. The above mentioned methods are designed to determine, through comparison of measured and simulated data, the validity of a model. As one might intuitively expect, different modelling errors usually cause the behavior of the model to deviate in different ways from that of the real system. Or, in other words, different modelling errors correspond to different "patterns" in the error signal, the difference between experimental data and simulated data. These "patterns", if extract-able, can obviously be used to identify the modelling errors. In the sequel, we present a simple biological process, which will be studied in more detail in a later chapter, to introduce different modelling errors and their unified representation. This representation is the starting point for automated error detection [YVV98].

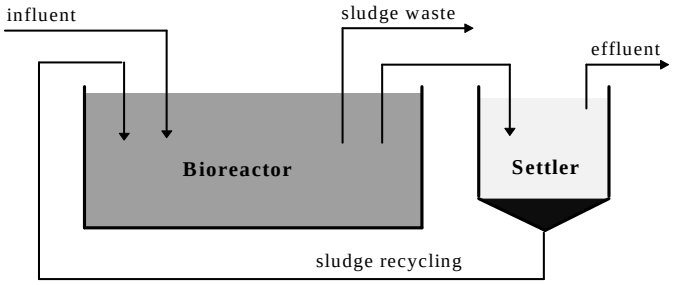

Figure 1.10: Biological Denitrification Process

1.3.1 Modelling a biological process

Figure 1.10 shows a biological denitrification plant, which aims to remove the nitrate as well as the carbon organics contained in the influent water by means of biological reactions. It consists of two functional units, a bio-reactor and a settler. In the reactor, which is often completely mixed, heterotrophic biomass is present. It biodegrades the carbon organics with nitrate as the electron acceptor. The carbon organics and the nitrate are thus both removed. The 'overflow' of the reactor, containing the substrate residuals and the sludge flocks (where the biomass resides), flows into the settler. There, the sludge settles and thus separates itself from the treated water, and is subsequently recycled to the reactor through the recycling line. In order to prevent the sludge concentration in the reactor from becoming too high due to its continuous growth in the reactor, surplus sludge is removed from the waste flow (see Figure 1.10). Models of the denitrification process usually aim to predict the effluent quality (the amount of carbon organics and nitrate in the effluent) and the sludge production. This implies that the following three variables are crucial to the model: the carbon organics concentration, the nitrate concentration, and the biomass concentration. The main biological reaction occurring in the reactor is known to be,

$$S + NO_3^- + H^+ \overset{\overset{\leftarrow}{r}}{\longrightarrow} X + N_2 + CO_2 + H_2O$$

where S, NO_3^-, H^+, X, N_2, CO_2 and H_2O denote, respectively, the carbon organics, nitrate, proton, biomass, nitrogen gas, carbon dioxide gas and water. r denotes the reaction rate. The "feedback" arrow in the scheme expresses the auto-catalytic action of the biomass X. As clearly shown in the scheme, the reaction results in the removal of the nitrate and carbon organics and in the growth of the biomass. Another reactor process is the decay of the biomass which causes the decrease of the biomass on the one hand and the consumption of the nitrate on the other hand. In the context of modelling the effluent quality, the *a priori* knowledge allows one to model the process by making mass balances for the three materials,

$$\begin{aligned}
\dot{X}(t) &= \mu(t)X(t) - bX(t) - \frac{Q_w(t)}{V}X(t) \\
\dot{S}_S(t) &= -\frac{1}{Y_S}\mu(t)X(t) - \frac{Q_{in}(t)}{V}S_S(t) + \frac{Q_{in}(t)}{V}S_{S,in}(t) \\
\dot{S}_{NO}(t) &= -\frac{1-Y_S}{2.86Y_S}\mu(t)X(t) - \frac{1-f_P}{2.86}bX(t) - \frac{Q_{in}(t)}{V}S_{NO}(t) + \frac{Q_{in}(t)}{V}S_{NO,in}(t)
\end{aligned} \tag{1.1}$$

where X, S_S, S_{NO} denote the biomass, the carbon organics and the nitrate concentrations in the bioreactor, respectively; $S_{S,in}$ and $S_{NO,in}$ denote the carbon organics and the nitrate concentrations in the influent, respectively; Q_{in} is the influent flow rate; Q_w is the waste flow rate; V is the volume of the bioreactor; Y_S is the yield coefficient; b is the biomass decay coefficient; f_P is the fraction of the inert materials in biomass; $\mu(t) = r(t)/X(t)$ is the specific biomass growth rate, which is still to be modelled.

Experiments show that μ is a nonlinear function of S_S and S_{NO}. It has been revealed that μ increases almost linearly with S_S and S_{NO} when they are low, but becomes independent of them when they are high.

Several empirical laws have been proposed to model this relationship. The following double Monod law is commonly used [HGG+86],

$$\mu(t) = \mu_{max}(\frac{S_S(t)}{K_S + S_S(t)})(\frac{S_{NO}(t)}{K_{NO} + S_{NO}(t)}) \tag{1.2}$$

where μ_{max} is the maximum specific growth rate, K_S and K_{NO} are the so-called half saturation coefficients for the carbon organics and the nitrate, respectively.

Equation (1.1), together with equation (1.2), gives a parametric model of the denitrification process. All the parameters involved are plant dependent and hence have to be specifically estimated for each individual case based on the data obtained either from on-site measurements or from laboratory analyses (of on-site samples).

1.3.2 Different types of modelling errors and their unified representation

Assume that the paramerized model to be validated takes the form,

$$\dot{x}_m(t) = f_m(x_m(t), \theta_m, u(t), t) \tag{1.3}$$

where $x_m(t) \in \mathbb{R}^n$ is the state variable vector of the model, $u(t) \in \mathbb{R}^p$ is the input vector, and θ_m is the model parameter vector, which is known. On the basis of this model, the real behavior of the system can generally be represented as,

$$\dot{x}_r(t) = f_m(x_r(t), \theta_m, u(t), t) + e_m(t) \tag{1.4}$$

where $x_r(t) \in \mathbb{R}^n$ is the state vector of the system, $e_m(t) \in \mathbb{R}^n$ is the modelling error vector. It is assumed in equation (1.4) that the real system has the same number of state variables as the model. This representation does not limit the generality of the representation since the errors introduced by erroneous state aggregations in deriving model (1.3) can also be represented by the error term $e_m(t)$.

In order to make the modelling error identification possible, an appropriate representation of the error term $e_m(t)$ in equation (1.4) is required. This representation should be obtained by making use of the *a priori* knowledge about the possible modelling errors. Basically, modelling errors may be introduced in each stage of the modelling process as depicted in Figure 1.5. In this section, it will be shown, taking the biological model developed in the previous section as an example, how the *a priori* knowledge concerning the modelling errors can be obtained through the analysis of the modelling process and the model itself. The mathematical representation of the modelling errors will also be discussed. As will be shown in the next section, such a representation allows the identification of the modelling errors based on the comparison of the observed data with data produced by simulation of the erroneous model.

Modelling Errors due to an improperly defined Experimental Frame

In defining the boundaries of the process or system to be modelled, some important components may be missed, some significant disturbances to the system may be improperly neglected and so on. All of these introduce errors into the model. The Experimental Frame is the formalisation of the experimental conditions (inputs applied to the system, outputs observed, criteria of acceptance, ...) and as such the above mentioned modelling errors can be formally expressed as Experimental Frame errors. For a rigorous treatment, see [Tak96].

For instance, an assumption underlying model (1.1) is that no other reactions occur in the process which affect the mass balance of the concerned materials. One knows, however, that this assumption is not valid when dissolved oxygen is present in the influent. In fact, when dissolved oxygen is fed to the bioreactor, the following reaction, which is called the aerobic oxidation, will also occur, accompanying the denitrification reaction described in the previous section,

$$S + O_2 \xrightarrow[r_o]{\longleftarrow} X + CO_2 + H_2O$$

where r_o denotes the oxidation reaction rate. The reaction scheme clearly shows how the aerobic oxidation affects the mass balance of the carbon organics and the biomass. This will inevitably introduce errors in the prediction of these two variables. Since, as shown in model (1.1), both $S_S(t)$ and X appear in the equation concerning the dynamics of S_{NO}, the prediction of the nitrate concentration will be affected indirectly.

A characteristic of the modelling error described above is that it does not directly affect the third equation in model (1.1). The above aerobic oxidation introduces an r_o term into the first equation of (1.1) and an $\frac{1}{Y_S}r_o$ term into the second equation. The modelling error term in equation (1.4) takes the following form,

$$e_{m,o}(t) = [1 \ -\frac{1}{Y_S} \ 0]^T r_o(t) \tag{1.5}$$

While $[1 \ -\frac{1}{Y_S} \ 0]^T$ is apparently a known vector, $r_o(t)$ is an unknown, time-variant scalar.

Modelling Errors due to an improperly characterized Model Structure

Due to for instance lack of knowledge of the mechanism of the process to be modelled, or due to an over-simplification of the model, one may assume a wrong model structure. Typical errors include choosing an incorrect number of state variables or incorrectly assuming non-linear behavior. Structural errors may accidentally be produced through incorrect choice of parameters (usually, 0), whereby some part of the model structure vanishes, thereby altering the model structure.

For instance, in model (1.1), there does not exist a fundamental law that precisely characterizes the dependence of the denitrification reaction rate on the concentrations of the materials. The "laws" which have hitherto been proposed are all quite empirical. A problem of this type of laws is that they have a limited applicability range. An inappropriate choice of the "laws" may introduce errors. For example, when the model of the denitrification rate given in equation (1.2) is not a good description of the real reaction rate: $\mu_r(t) = \mu(t) + \delta\mu_s(t)$, where $\mu_r(t)$ is the real specific reaction rate and $\delta\mu_s(t)$ is the modelling error, the following error term is found by substitution in equation (1.4),

$$e_{m,\mu}(t) = [1 \ -\frac{1}{Y_S} \ -\frac{1-Y_S}{2.86Y_S}]^T \delta\mu_s(t)X(t) \tag{1.6}$$

Modelling Errors due to inaccurate estimates of the Model Parameters

Either by improper or inadequate data used for parameter estimation or by ill designed estimation algorithms, one may use incorrect parameter values. The error terms in equation (1.4) due to the estimate errors of the parameters in model (1.1) are as follows,

modelling error of b

Assuming $b_r = b + \delta b$, where b_r is the real decay coefficient and δb is the modelling error, one obtains,

$$e_{m,b}(t) = [-1 \ 0 \ -\frac{1-f_P}{2.86}]^T \delta b X(t) \tag{1.7}$$

modelling error of f_P

Assuming $f_{P,r} = f_P + \delta f_P$, where $f_{P,r}$ is the real inert fraction in a biomass cell and δf_P is the modelling error, one obtains,

$$e_{m,f_P}(t) = [0 \ 0 \ 1]^T \frac{\delta f_P}{2.86} bX(t) \tag{1.8}$$

modelling error of Y_S

Assuming $\frac{1}{Y_{S,r}} = \frac{1}{Y_S} + \delta(\frac{1}{Y_S})$, where $Y_{S,r}$ is the real yield coefficient and $\delta(\frac{1}{Y_S})$ is the modelling error, one obtains,

$$e_{m,Y_S}(t) = [0 \ -1 \ -\frac{1}{2.86}]^T \delta(\frac{1}{Y_S})\mu(t)X(t) \tag{1.9}$$

modelling errors of μ_{max}, K_{NO} and K_S

Assuming $\mu_r(t) = \mu(t) + \delta\mu_p(t)$, where $\mu_r(t)$ is the real specific reaction rate and $\delta\mu_p(t)$ is the error caused by the modelling error of μ_{max}, K_{NO} or K_S, one obtains,

$$e_{m,\mu_{max},K_{NO},K_S}(t) = [1 \; -\frac{1}{Y_S} \; -\frac{1-Y_S}{2.86Y_S}]^T \delta\mu_p(t)X(t) \tag{1.10}$$

One finds that every single modelling error shown above takes the form of a product of a known constant vector and an unknown time-variant variable. This is not an artifact of this particular example, but is in fact a general property. Usually, each modelling error affects only a subspace of the *n*-dimensional state space, and can hence be represented in equation (1.4) with a term $F_i d_i(t)$, where $F_i \in R^{n\times s_i}$, $d_i(t) \in R^{s_i}$. The vectors of F_i span the subspace affected by the concerned modelling error. F_i is called the feature vector or feature matrix of the modelling error. $d_i(t)$ represents the magnitude of the modelling error, and is generally unknown and time-varying. Thus, equation (1.4) can be rewritten as,

$$\dot{x}_r(t) = f_m(x_r(t), \theta_m, u(t), t) + \sum_{i=0}^{l} F_i d_i(t) \tag{1.11}$$

Since it is usually not possible to predict all possible modelling errors, it is necessary to include a special feature matrix, say F_0, in equation (1.11) to represent modelling errors which were not explicitly modelled. Obviously, the *n*-dimensional identity matrix is suitable for that purpose.

To allow for meaningful error identification, some assumptions are made with respect to equation (1.11):

- The individual errors are written in "additive" form:

  $$v_r = v + \delta v$$

  Such a "choice" of individual error terms is always possible without loss of generality. One may be required to "lump" non-linear errors as in $\delta(Y_S)$ or $\delta\mu_p$ above.
- Simultaneously occurring errors are assumed to be either additive, or sufficiently small to allow for an linear approximation:

  $$f(A + \delta A, B + \delta B) \cong f(A,B) + \frac{\partial f}{\partial A}(A,B)\,\delta A + \frac{\partial f}{\partial B}(A,B)\,\delta B$$

  Though such an assumption is not necessary *per se*, as non-linear effects can always be lumped into an extra error term (using the above mentioned F_0), this would defeat our purpose of isolating individual error contributions.

1.4 Abstraction levels and formalisms

There are several reasons why abstract models of systems are used. First of all, an abstract model description of a system *captures knowledge* about that system. This knowledge can be stored, shared, and re-used. Furthermore, if models are represented in a standard way, the *investment* made in developing and validating models is paid off as the model will be understood by modelling and simulation environments of different vendors for a long time to come.

Secondly, an abstract model allows one to formulate and answer *questions* about the structure and behaviour of a system. Often, a model is used to obtain values for quantities which are non-observable in the real system. Also, it might not be financially, ethically or politically feasible to perform a real experiment (as opposed to a simulation or virtual experiment). Answering of *structure related* questions is usually done by means of symbolic analysis of the model. One might for example wish to know whether an electrical circuit contains a loop. Answering of questions about the *dynamic behaviour* of the system is done (by definition) through *simulation.* Simulation may be symbolic or numerical. Whereas the aim of modelling is to provide

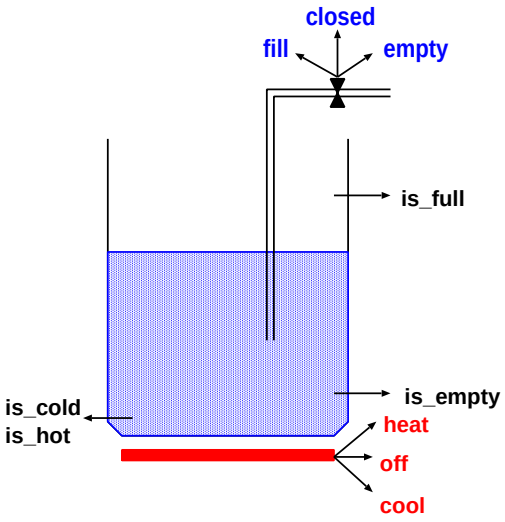

Figure 1.11: T, l controlled liquid

insight and to allow for re-use of knowledge, the aims of simulation are accuracy and execution speed (often real-time, with hardware-in-the-loop).

One possible way to construct systems models (particularly in systems design) is by copying the structure of the system. This is not a strict requirement. A neural network which simulates the behaviour of an aeration tank in an activated sludge waste water treatment plant is considered a "model" of the tank. It may accurately replicate the behaviour of the tank, though the physical structure of the tank and its contents is no longer apparent. For purposes of control, we are often satisfied with a performant (real-time) model of a system which accurately predicts its behaviour under specific circumstances, but bears no structural resemblance with the real system.

Abstract models of system behaviour can be described at different levels of abstraction or detail as well as by means of different formalisms. The particular formalism and level of abstraction used depend on the background and goals of the modeller as much as on the system modelled. As an example, a temperature and level controlled liquid in a pot is considered as shown in Figure 1.11. This is a simplified version of the system described in [BZF98], where structural change is the main issue. On the one hand, the liquid can be heated or cooled. On the other hand, liquid can be added or removed. In this simple example phase changes are not considered. The system behaviour is completely described by the following (hybrid) Ordinary Differential Equation (ODE) model:

Inputs (discontinuous $\rightarrow$ hybrid model):

- Emptying, filling flow rate ϕ
- Rate of adding/removing heat W

Parameters:

- Cross-section surface of vessel A
- Specific heat of liquid c
- Density of liquid ρ

State variables:

- Temperature T
- Level of liquid l

Outputs (sensors):

- $is\_low, is\_high, is\_cold, is\_hot$

$$\begin{cases} \frac{d\mathsf{T}}{dt} &= \frac{1}{l}\left[\frac{W}{c\rho A} - \phi\mathsf{T}\right] \\ \frac{dl}{dt} &= \phi \\ is\_low &= (l < l_{low}) \\ is\_high &= (l > l_{high}) \\ is\_cold &= (\mathsf{T} < \mathsf{T}_{cold}) \\ is\_hot &= (\mathsf{T} > \mathsf{T}_{hot}) \end{cases}$$

The inputs are the filling (or emptying if negative) flow rate ϕ, and the rate W at which heat is added (or removed if negative). This system is parametrized by A, the cross-section surface of the vessel, H, its height,

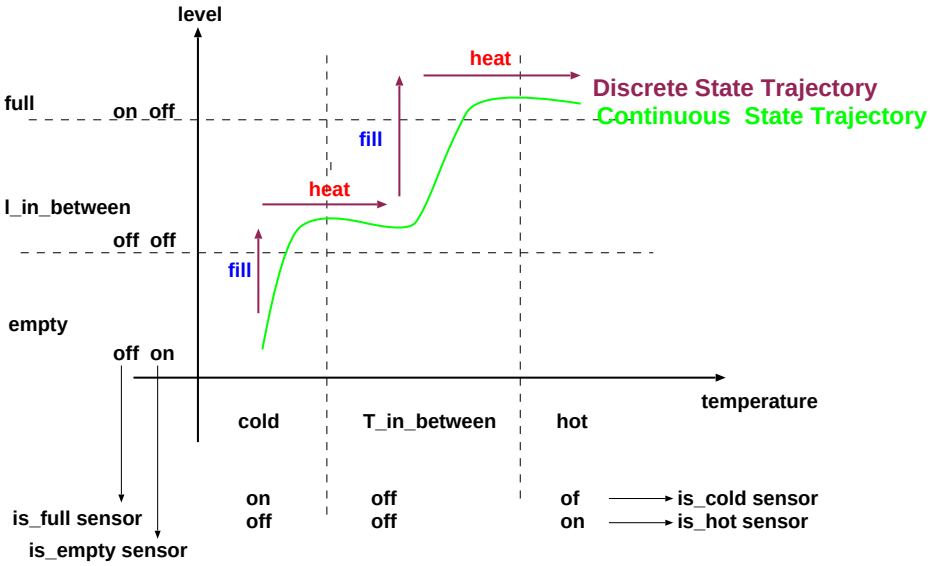

Figure 1.12: Trajectories

c, the specific heat of the liquid, and ρ, the density of the liquid. The state of the system is characterized by variables T, the temperature and l, the level of the liquid. The system is observed through threshold output sensors $is\_low, is\_high, is\_cold, is\_hot$. Given input signals, parameters, and a physically meaningful initial condition (T_0, l_0), simulation of the behaviour yields a continuous state trajectory as depicted in Figure 1.12. By means of the binary (on/off) level and temperature sensors introduced in the differential equation model, the state-space may be discretized. The inputs can be abstracted to heater heat/cool/off and pump fill/empty/closed. At this level of abstraction, a Finite State Automaton (with 9 possible states) representation of the dynamics of the system as depicted in Figure 1.13 is most appropriate. Though at a much higher level of abstraction, this model is still able to capture the essence of the system's behaviour. In particular, there is a *behaviour morphism* between both models: model discretization (from ODE to FSA) followed by simulation yields the same result as simulation of the ODE followed by discretization. This morphism is shown as a commuting diagram in Figure 1.14.

1.5 System specification

When studying existing systems, *observations* (of structure and behaviour) are the only tangible artifacts we have at our disposal [Kli85]. A modeller may, based on observations and/or insight, build progressively more complex models of a system. In this section, we present a hierarchy of abstract model structures. Each structure elaborates on the previous one, introducing (and representing) more detailed knowledge about the system. The reverse operation, going from a higher-level model to a less detailed one, must be shown to be possible. This, as some questions about the behaviour and structure of the system are better answered at lower levels in the hierarchy. In particular, explicit behaviour in the form of trajectories, described at the lowest level, is often required.

In object-oriented terminology, a simulation *model* consists of model *objects* (often used to represent real-world objects, entities, or concepts) as well as *relationships* among those objects. In general, a model object is anything that can be characterized by one or more *attributes* to which *values* are assigned. Attributes are either called *indicative* if they describe an aspect inherent to the object or *relational* if they relate the object to one or more other objects. The values assigned to attributes have a *type* in the programming language sense.

Mathematical *sets* and operations defined on those sets are the starting point for abstract system representation or modelling. Simple finite sets of numbers $\{1, 2, \ldots, 9\}$, identifiers $\{a, b, \ldots, z\}$, as well as infinite sets such as $\mathbb{N}, \mathbb{N}^+, \mathbb{R}$, and $\mathbb{R}^+$ are typically used. Often, specific *meaning* is given to sets and their members. The

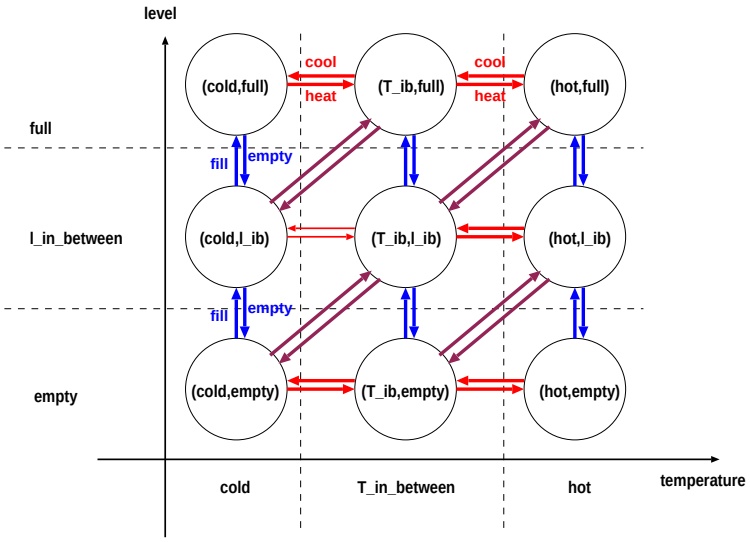

Figure 1.13: FSA formalism

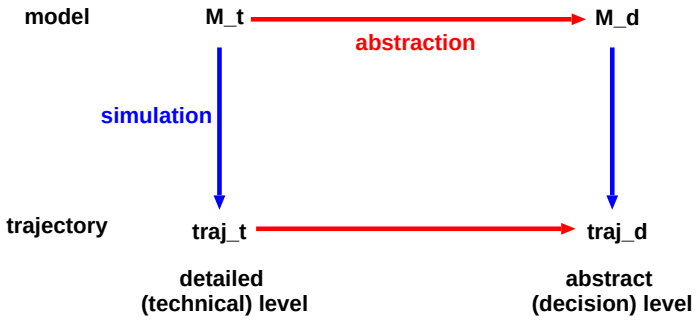

Figure 1.14: Behaviour morphism

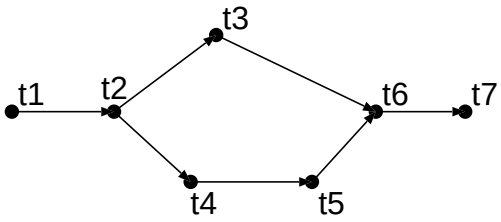

Figure 1.15: Partially ordered time base

set EV for example is a finite set denoting arrival and departure *events* in a queueing system

$$EV = \{ARRIVAL, DEPARTURE\}.$$

As in the *discrete event* abstraction, discussed later in greater detail, only a finite number of events are assumed to occur in a bounded time interval, the non-event symbol ϕ is introduced to denote the absence of events changing the state of the system. The event set is subsequently enriched with ϕ

$$EV^{\phi} = EV \cup \{\phi\}.$$

This demonstrates the use of basic set operations such as $\cup$. To describe attributes of a system, the set product $\times$ is used

$$A \times B = \{(a,b) | a \in A, b \in B\}.$$

1.5.1 Time base

Every simulation model must have an *indexing attribute* which, at some level of abstraction will enable state transitions [Nan81]. *Time* is the most common indexing attribute. Time is special in that it inexorably progresses: the current state and behaviour of a system can only modify its future, never its past. This concept is often called *causality*: a cause must always occur before a consequence. In a simulation context, the indexing attribute is referred to as *system time*. Any set T can serve as a formalisation of time. A *nominal* relationship $=$ may be added to T to denote equality. To obtain a usable *time base* however, an *order* relation on the elements of T is needed:

$$TimeBase = \langle T, < \rangle$$

This relation has properties

- transitive: $A < B \wedge B < C \Rightarrow A < C$,
- irreflexive: $A \not< A$,
- antisymmetric: $A < B \Rightarrow B \not< A$.

This formalises the notion of order in time. The ordering relationship may be *total* (linear): each element of T can be related to every other element. A *partial* ordering where not all elements of T can be compared is useful in modelling uncertainty or concurrency. In Figure 1.15 for example, the nodes denote time instants and the edges denote "precedes in time" ($<$). t_2 precedes both t_3 and t_4 in time, but no information is available about the relative position in time of t_3 and t_4. In case of concurrent behaviour, causality must not be violated within the individual concurrent threads, but the time-ordering between concurrent events may be left unspecified [Mil93]. Mathematically, this leads to a lattice structure. In case of total ordering, *intervals* may be defined. The past $T_{t[}$ and future $T_{]t}$ of an instant $t \in T$ may be defined

$$T_{t[} = \{\tau | \tau \in T, \tau < t\},$$

$$T_{]t} = \{\tau | \tau \in T, t < \tau\}.$$

Once intervals have been defined,

$$T_{\langle t_b, t_e \rangle}$$

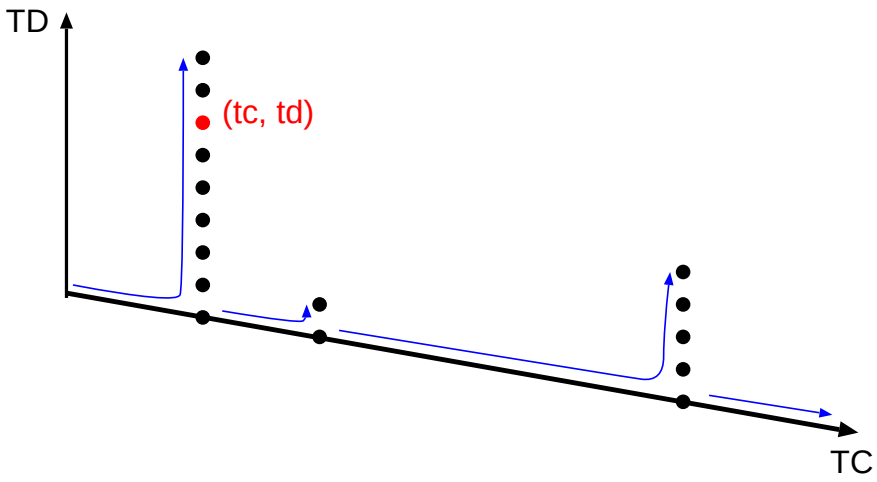

Figure 1.16: Time base for hybrid system models

denotes a time interval, where $\langle t$ means $]t$ or $[t$. In many cases, $(T,+)$ is an Abelian group with zero 0 and inverse $-t$.

Common time bases (with appropriate $<$ and $+$) are

- $T = \{NOW\}$. Models such as algebraic models are *instantaneous.* The time base is a singleton.
- $T = \mathbb{R}$. Models with this time base are called *continuous-time* models. Note how discrete event models have $\mathbb{R}$ as a time base. However, only at a finite number of time-instants in a bounded time-interval, an event different from the non-event ϕ occurs.
- $T = \mathbb{N}$ (or isomorphic). Models with this time base are called *discrete-time* models. Some formalisms such as Finite State Automata (FSA) do not have an explicit notion of time (unlike their extension, timed automata). There is however a notion of progression (from one state to another). According to our general definition, the index of progression, a natural number, is time.

In *hybrid* system models which combine aspects of continuous and discrete models [MB01], a system evolves continuously over time ($\mathbb{R}$) until a certain condition is met. Then, *instantaneously* (the continuous time does not progress), the system may go through a number of discrete states (the index of progression is discrete) before continuing its continuous behaviour. To uniquely describe progression (of generalized time) in this case, a tuple (t_c, t_d) depicted in Figure 1.16 is needed. Even when a series of discrete transitions keep returning to the *same* state, the discrete index t_c allows one to distinguish between them. The time base used is

$$T = \{(t_c, t_d) | t_c \in \mathbb{R}, t_d \in \{1, \ldots, N(t_c)\}\}.$$

Here, $N(t_c)$ (≥ 1) describes the number of discrete transitions the system goes through at continuous time t_c. Obviously, only a partial ordering will be defined over T which consists of first testing the relationship between the t_c components, and subsequently (if equal), that between the t_d components.

In case of Partial Differential Equations (PDEs), the time base remains $\mathbb{R}$. The other *independent variables* (often, space in the form of some coordinate system) should be seen as infinitely many state-variable labels or *generalized coordinates.*

Given a time base, we wish to formalize *behaviour* over time. This is done by means of a time function, called *trajectory* or *signal*

$$f : T \to A$$

describing, at each time t, the value of the signal. A denotes the set of valid values f can take over T. The time base may be restricted to a subset of T: $T' \subseteq T$. The restriction of f to T' is

$$f|T' : T' \to A,$$

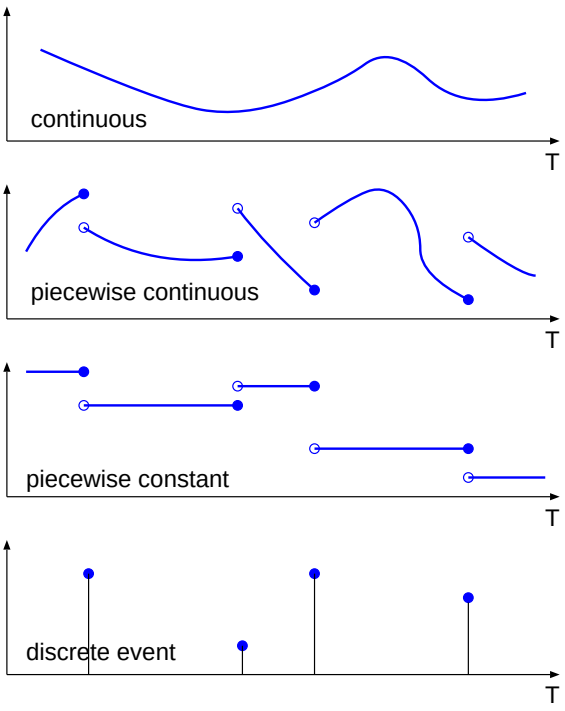

Figure 1.17: Segment types

$$\forall t \in T' : f|T'(t) = f(t).$$

The past of f is defined as $f|T_{t\rangle}$. The future of f is defined as $f|T_{\langle t}$. The restriction of f to an interval is called a *segment* ω

$$\omega : \langle t_1, t_2 \rangle \to A.$$

The set of all possible (allowed) segments is called Ω. Segments are contiguous if their domains $\langle t_1, t_2 \rangle$ and $\langle t_3, t_4 \rangle$ are contiguous: $t_2 = t_3$.

Contiguous segments may be concatenated – $\omega_1 \bullet \omega_2$:

$$\omega_1 \bullet \omega_2(t) = \omega_1(t), \forall t \in dom(\omega_1);$$

$$\omega_1 \bullet \omega_2(t) = \omega_2(t), \forall t \in dom(\omega_2),$$

where $\langle$ and $\rangle$ must denote matching open/closed interval boundaries to ensure the concatenated segment is still a *function* (*i.e.*, has a unique value in each point of its domain).

A desirable property of a set of segments Ω is that it is closed under concatenation $\bullet$: concatenating *any* left and right segment of a segment yields the same segment:

$$\forall t \in dom(\omega) : \omega_{t\rangle} \bullet \omega_{\langle t} = \omega.$$

Figure 1.17 shows some common segment types: continuous, piecewise continuous, piecewise constant and discrete event. Note how for discrete event systems, inputs and output segments are *event segments*

$$\omega : \langle t_1, t_2 \rangle \to A \cup \{\phi\},$$

with ϕ the non-event. For such systems, the internal state behaviour is piecewise constant (the internal state only changes at event times).

1.5.2 Levels of system specification

With a time base and segments defined, we can build a hierarchy of system specification structures which incorporate progressively more knowledge about the system. All these structures will view the system as a *box* interacting with its environment through a well defined *interface.* The levels presented here elaborate on the hierarchy first proposed by Klir [Kli85] and later modified by Zeigler [ZPK00].

Observation Frame

At the lowest level, the only knowledge we have of the behaviour of a system is how we wish to observe it: which time base to use and which quantities to observe at instants from the time base. This is represented in the form of an Observation Frame O:

$$O \equiv \langle T, X, Y \rangle.$$

T with appropriate operators forms a time base. X is the input value set. It is a model for the input (influencing the behaviour of the system) variables we consider. Y is the output value set. It is a model for the system response variables.

I/O Relation Observation

Once the interface variables to observe as well as their value ranges have been determined, all possible *relationships* between input and output segments can be recorded

$$IORO \equiv \langle T, X, \Omega, Y, R \rangle.$$

Here, $\langle T, X, Y \rangle$ is an Observation Frame, and Ω is the set of all possible input segments for this system. Note how Ω allows one to specify how the system's *environment* may influence the system. As such, Ω formalizes the Experimental Frame's generator presented before. Ω is a subset of all mathematically possible segments with T as domain and X as image. R is the *I/O relation*

$$R \subseteq \Omega \times (Y, T),$$

where (Y, T) stands for all possible segments with T as domain and Y as image. Input segments ω and output segments ρ are defined as

$$\omega : \langle t_i, t_f \rangle \rightarrow X;$$

$$\rho : \langle t_i, t_f \rangle \rightarrow Y.$$

Though not necessary, it is common to observe input and output segments over the same time domain. The relation R relates input and output segments

$$(\omega, \rho) \in R \Rightarrow dom(\omega) = dom(\rho).$$

As will be discussed further on, general *non-causal* relationships between interface variables, not specifying *a priori* which are input and which are output may be specified by R. Higher levels are explicitly causal.

It is possible to go from an I/O Relation Observation model specification to an Observation Frame level model by merely discarding the Ω and R information at the I/O Relation Observation level.

I/O Function Observation

At the I/O Relation Observation level, an input segment ω is not necessarily associated with a unique output segment ρ. This is due to a limited knowledge of the internal working of the system. At the I/O Function Observation level, we want to associate a *unique* output segment with every input segment. Therefore, more information needs to be specified about the system. This is done in the form of a set F of *I/O functions* f. This leads to the I/O Function Observation structure

$$IOFO \equiv \langle T, X, \Omega, Y, F \rangle,$$

where $\langle T, X, Y \rangle$ is I/O Relation Observation, Ω is the set of all possible input segments, F is the set of I/O functions:

$$f \in F \rightarrow f \subset \Omega \times (Y, T);$$
$$dom(f(\omega)) = dom(\omega).$$

f is conceptually equivalent to the system's *initial state*: For each f, an input segment will be transformed into a unique output segment.

It is possible to go from an I/O Function Observation to an I/O Relation Observation by constructing R from F:

$$R = \bigcup_{f \in F} f.$$

I/O System

In some cases, we have some insight into the *internal* working of the system. This insight usually consists of a number of *descriptive variables* and how their values evolve over time. Under certain conditions, these variables are *state variables.*

In general systems theory [Wym67], a causal (output is the consequence of given inputs), deterministic (a known input will lead to a unique output) system model *SYS* is defined. It is a template for a plethora of different formalism such as Ordinary Differential Equations, Finite State Automata, Difference Equations, Petri Nets, *etc.* Its general form is

$$SYS \equiv \langle T, X, \Omega, Q, \delta, Y, \lambda \rangle$$

T	time base
X	input set
$\omega : T \rightarrow X$	input segment
Q	state set
$\delta : \Omega \times Q \rightarrow Q$	transition function
Y	output set
$\lambda : Q \rightarrow Y$ (or $Q \times X \rightarrow Y$)	output function

$$\forall \omega, \omega' \in \Omega, \delta(\omega \bullet \omega', q_i) = \delta(\omega', \delta(\omega, q_i)).$$

The time base T is the formalisation of the independent variable time. The input set X describes all possible allowed inputs (possibly a product set). An input segment ω represents input during a time-interval. The history of system behaviour is condensed into a *state* (from a state set Q). The dynamics is described in a transition function δ which takes a current state, and applies an input segment $\omega \in \Omega$ to it to obtain a new state. The system may generate output. This output is obtained as a function λ of the state (and more generally, of the current input too). State and transition function must obey the composition or semigroup property as shown in Figure 1.18. This property, whereby a transition over a time interval $[t_i, t_f]$ can always be split into a composition of transitions over arbitrary sub-intervals, is the basis of all model simulators. Obviously, this also requires Ω to be closed under concatenation.

As the output function is described separately, efficient simulators will *only* invoke this function (which may be large and compute-intensive) when the user needs to observe output. Note how the *output intervals* (times between outputs) are not part of the model, but rather of the simulation experiment. Figure 1.19 shows how output need not be produced at each transition time. Even though a model written by a user may not distinguish between δ and λ, a simple *dependency analysis* will identify which variables and expressions are not needed to compute δ. Such variables are output variables $\in Y$ and the expressions belong in λ.

As *SYS* is a template for a host of causal, deterministic formalisms, it is possible to describe both models of the vessel example. In the Ordinary Differential Equation (ODE) case, the time base is continuous ($\mathbb{R}$). The transition function is written in integral form. Different numerical approximations of the integral can be used in the implementation of an abstract simulator.

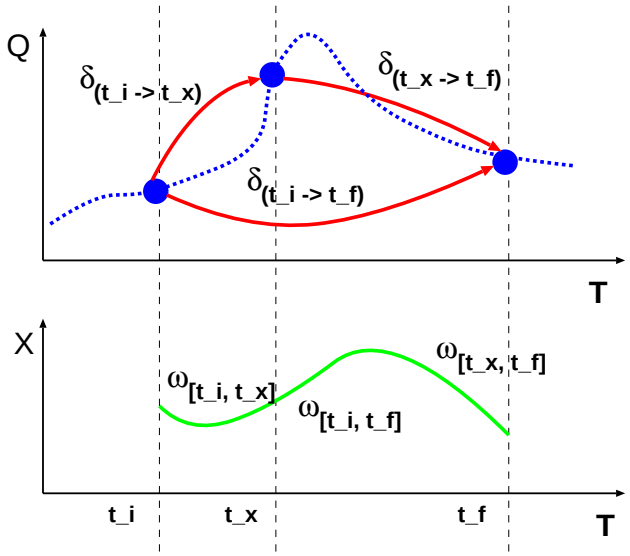

Figure 1.18: SYS state transition property

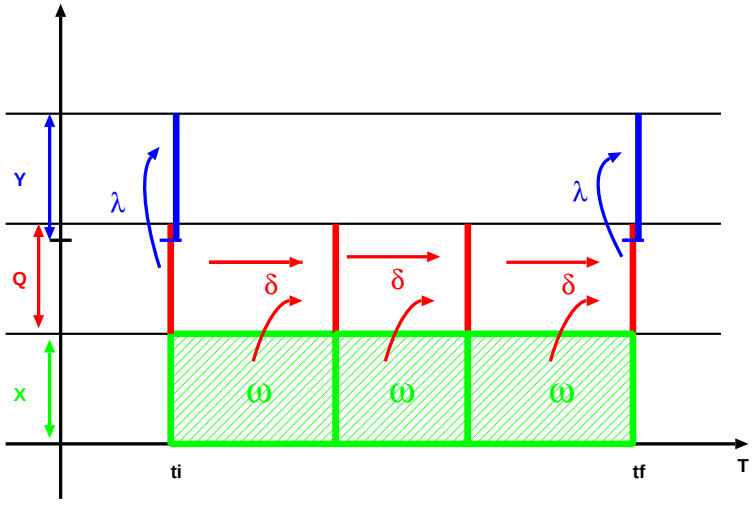

Figure 1.19: Simulation kernel operation

$$SYS^{ODE}_{VESSEL} = \langle T, X, \Omega, Q, \delta, Y, \lambda \rangle$$

$T = \mathbb{R}$
$X = \mathbb{R} \times \mathbb{R} = \{(W, \phi)\}$
$\omega : T \to X$
$Q = \{(\mathsf{T}, l) | \mathsf{T} \in \mathbb{R}^+, l \in \mathbb{R}^+\}$
$\delta : \Omega \times Q \to Q$
$\delta(\omega_{[t_i,t_f]}, (\mathsf{T}(t_i), l(t_i))) =$

$$(\mathsf{T}(t_i) + \int_{t_i}^{t_f} \frac{1}{l(\alpha)}[\frac{W(\alpha)}{c\rho A} - \phi(\alpha)\mathsf{T}(\alpha)]d\alpha,\ l(t_i) + \int_{t_i}^{t_f} \phi(\alpha)d\alpha)$$

$Y = \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} = \{(is\_low, is\_high, is\_cold, is\_hot)\}$
$\lambda : Q \to Y$
$\lambda(\mathsf{T}, l) = ((l < l_{low}), (l > l_{high}), (\mathsf{T} < \mathsf{T}_{cold}), (\mathsf{T} > \mathsf{T}_{hot}))$.

At a higher level of abstraction, we have represented time as a discrete integer index. The transition function lists all possible state transitions.

$$SYS^{FSA}_{VESSEL} = \langle T, X, \Omega, Q, \delta, Y, \lambda \rangle$$

$T = \mathbb{N}$
$X = \{heat, cool, off\} \times \{fill, empty, closed\}$
$\omega : T \to X$
$Q = \{(\mathsf{T}, l) | \mathsf{T} \in \{cold, \mathsf{T}_{between}, hot\}, l \in \{empty, l_{between}, full\}\}$
$\delta : \Omega \times Q \to Q$
$\delta((off, fill), (cold, empty)) = (cold, l_{between})$
$\delta((off, fill), (cold, l_{between})) = (cold, full))$
$\delta((off, fill), (cold, full))) = (cold, full))$
$\vdots$
$\delta((heat, fill), (hot, full))) = (hot, full))$
$Y = \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}$
$\lambda : Q \to Y$
$\lambda(\mathsf{T}, l) = ((l = low), (l = high), (\mathsf{T} = cold), (\mathsf{T} = hot))$

The Finite State Automaton formalism [CL89]

$$FSA \equiv \langle \Sigma, S, s_0, d, F \rangle,$$

where

- Σ is the input alphabet (a finite and nonempty set of symbols),
- S is the finite nonempty set of states,
- s_0 is the initial (or start) state, $s_0 \in S$,
- $d : S \times \Sigma \to S$ is the state transition function,
- $F \subseteq S$ is the set of final or accepting states,

fits the general SYS structure presented above.

The formalism is specified by elaboration of the elements of the SYS 7-tuple:

$$SYS \equiv \langle T, X, \Omega, Q, \delta, Y, \lambda \rangle.$$

In the SYS specification, the initial state is not explicitly represented. Also, accepting states are not defined.
The time base

$$T = \mathbb{N} \text{ (or isomorphic with } \mathbb{N}).$$

This can be interpreted as (implicit) discrete time-clicks. It is possible to extend the FSA formalism to assign with each state and/or with each transition, a known duration.
The input set

$$X = \Sigma.$$

The set Ω of all input segments ω. An input segment encodes a *sequence* of inputs from X.
The finite state set

$$Q = A$$

enumerates all states in the automaton.
The state transition function δ transforms a current state, through input and time-advance, to a new state

$$\delta : \Omega \times Q \to Q.$$

It is obtained by iteratively applying all FSA state transitions f in an input segment ω.
The output function λ takes the form

$$\lambda : Q \to Y$$

in case of a Moore machine (the input can only influence the output *via* the state, or

$$\lambda : \Omega \times Q \to Y$$

in case of a Mealy machine (the input can *directly* influence the output).
It is possible to go from an I/O System specification to an I/O Function Observation. For a given initial condition q and a given input segment ω, we can define a *state trajectory* $STRAJ_{q,\omega}$ from SYS

$$STRAJ_{q,\omega} : dom(\omega) \to Q,$$

with

$$STRAJ_{q,\omega}(t) = \delta(\omega_{t\rangle}, \forall t \in dom(\omega).$$

From this state trajectory, an *output trajectory* $OTRAJ_{q,\omega}$ may be constructed

$$OTRAJ_{q,\omega} : dom(\omega) \to Y,$$

with

$$OTRAJ_{q,\omega}(t) = \lambda(STRAJ_{q,\omega}(t), \omega(t)), \forall t \in dom(\omega).$$

Thus, for every q (initial state), it is possible to construct

$$\mathcal{T}_q : \Omega \to (Y, T),$$

where

$$\mathcal{T}_q(\omega) = OTRAJ_{q,\omega}, \forall \omega \in \Omega.$$

The I/O Function Observation associated with SYS is then

$$IOFO = \langle T, X, \Omega, Y, \{\mathcal{T}_q(\omega) | q \in Q\} \rangle.$$

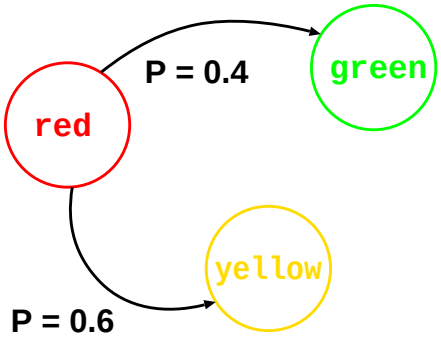

Figure 1.20: Non-deterministic model with transition probabilities

Subsequently, we may derive the I/O Relation Observation by constructing the relation R as the union of all I/O functions:

$$R = \{(\omega, \rho) | \omega \in \Omega, \rho = OTRAJ_{q,\omega}, q \in Q\}.$$

In SYS, δ is *deterministic*: applying the same input segment to the same state will always lead to the *same, unique* new state (and output). Often, deterministic simulation kernels are used to simulate non-deterministic models. Two main approaches are possible:

1. A deterministic model is decorated with transition probabilities as shown in Figure 1.20. The same model is then simulated a number of times, with the same initial conditions and parameters. Whenever a non-deterministic transition is encountered however, a unique, deterministic, transition is chosen by sampling from a stochastic distribution, taking into account the transition probabilities in the model. Thus, from the point of view of the simulation engine, it is simulating a deterministic model. To be able to make meaningful statements about the behaviour of the non-deterministic model, a sufficient number of samples must be simulated to obtain statistically relevant *estimates* of performance metrics (such as average queue lengths in a queueing model). In discrete event simulation in particular, this approach is common and its statistical aspects have been studied in great detail [LK91]. In a slightly modified form, this approach is called Monte Carlo simulation.

2. One may wish not to specify any probability distribution but leave the uncertainty of making a transition to more than one new state in the transition function. In case of State Automata, this turns the transition graph into a transition hypergraph [Har88]. Such a specification can always be transformed into a deterministic one by constructing a new state set $Q_{new} = 2^Q$, the set of all subsets (powerset) of Q [Cas93]. A new transition function is constructed describing the –now deterministic– transition to a new state, denoting the set of states from Q to which a non-deterministic transition existed in the old model. It is noted that in quantum physics, evolution over time of a wave function (a distribution interpreted as being probabilistic) is also deterministic.

Zeigler [ZPK00] presents a refinement of SYS in which behaviour is specified as an *iterative* application of *generator segments*. Arbitrary input segments are generated from elementary segments.

It should be noted that though models may be iteratively simulated, this is not necessary per se. If a symbolic (analytical) solution can be found, this is often preferable. Analytical solutions usually describe a (parametrised) class of solutions rather than a single one. Also, accumulation of numerical errors is often avoided. As an example, the following model described in the Difference Equation formalisms ($T = \mathbb{N}, Q = \mathbb{R}$)

$$\begin{cases} x_1 &= 1 \\ x_{i+1} &= ax_i + 1, \end{cases}$$

can be re-written as

$$\begin{cases} x_1 &= 1 \\ x_n &= 1 + a + a^2 + \ldots + a^{n-1} \\ ax_n &= a + a^2 + \ldots + a^{n-1} + a^n, \end{cases}$$

which leads to the instantaneous (no iteration required) solution

$$x_n = \frac{1 - a^n}{1 - a}.$$

1.5.3 Structured specifications

Upto now, no *structure* was explicitly specified for the sets $X, Y,$ *etc.* at any of the above specification levels. *Orthogonal* to the specification hierarchy, at each of the levels, the internal structure of input, output, and state sets as well as of functions may be made explicit. This allows one to construct sets from more primitive sets.

One way of introducing structure is through *multivariable sets.* A multivariable set (one possible representation of a programming language Symbol Table) uses a finite sequence V of n distinct variable names, identifiers, labels, or references

$$V = (v_1, v_2, \ldots, v_n).$$

With each of these names will be associated a *value set* of values a variable with that name may take

$$V_1, V_2, \ldots V_n.$$

The full multivariable set is then

$$S = (V, V_1 \times V_2 \times \ldots V_n).$$

A projection operator . can be defined

$$. : S \times V \to \bigcup_{j=1}^{n} V_j, S.v_i = s_i, \forall v_i \in V.$$

With A and B structured sets, a structured function may be defined

$$f : A \to B,$$

where the projection of f on a name in the image set is

$$f.b_i : A \to ((b_i), B_i),$$

$$f.b_i(a) = f(a).b_i$$

In case of interfaces or ports, the names denote individual port names. For example, in the water pot example

$$X = ((heatFlow, liquidFlow), \mathbb{R} \times \mathbb{R}).$$

With $x \in X$, we may refer to the $x.heatFlow$ input port value of the model.

In case of state variables, the names denote variable names. Again, in the water pot example

$$S = ((temperature, level),]0.0, 100.0[\times [0, H]).$$

With $s \in S$, we may refer to the $S.temperature$ state variable value. As such, structured sets and functions are similar to *variables and* their *types* in programming languages.

Figure 1.21 depicts a simple single server, single queue system. In a discrete event model of this system, the state set could be a structured set containing a simple abstraction of the queue (the queue length) and the status of the server

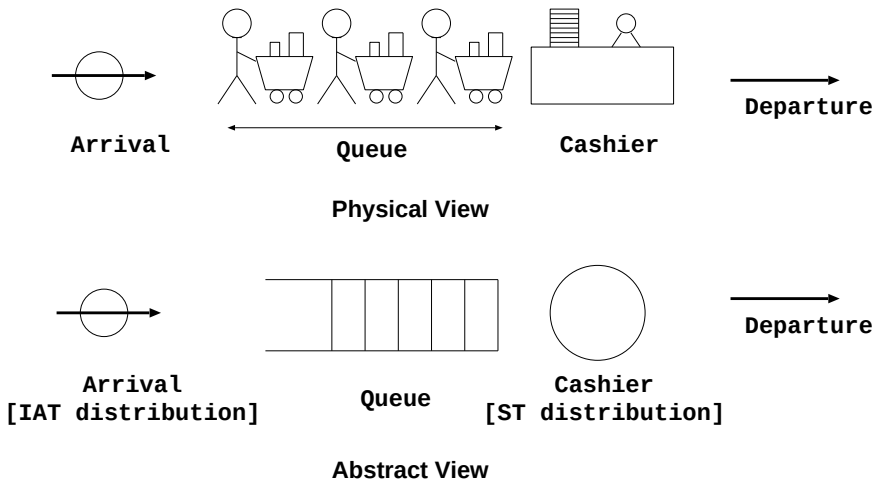

Figure 1.21: Simple single queue, single server system

$$SQ = ((qLength, cashStatus), \mathbb{N} \times \{Idle, Busy\}).$$

With $s \in SQ$, we may refer to $s.qLength$.

Structured sets may be used to add structure to I/O Observation Frame, I/O Relation Observation, I/O Function Observation, as well as to I/O system formalisms.

A modelling language (such as MSL-USER, which will be described in a subsequent chapter) will use structured sets to allow the user to describe ports and variables by name (rather than by an index in a product set).

1.5.4 Multicomponent specifications

A common means to tackle complexity is to decompose a problem *top-down* into smaller sub-problems. Conversely, complex solutions may be built *bottom-up* by combining primitive sub-problem solution building blocks. Both approaches are instances of *compositional modelling*: the connection (but more general, composition) of *interacting component* models to build new models. In case the components *only* interact via their interfaces, and do not influence each other's internal working in any other way (model information is completely *encapsulated* in object-oriented terminology [Boo98, Zei97, Weg90]), the compositional modelling approach is called *modular*. If inter-components access is not restricted to take place via interfaces or ports only, the approach is called *non-modular*.

Modular multicomponent specification

A modular multicomponent specification or *coupled* model or *network* model as depicted in Figure 1.22 can be mathematically described as a structure

$$N = \langle T, X_{self}, Y_{self}, D, \{M_d | d \in D\}, \{I_d | d \in D \cup \{self\}\}, \{Z_d | d \in D \cup \{self\}\}\rangle.$$

In this structure,

- X_{self} and Y_{self} are inputs and outputs of the network N. $self$ (`this` in `C++` terminology) allows us to treat the network object itself as any other object. In a modular specification, the network will interact with its environment through X_{self} and Y_{self} *only*.
- D is a set of component *references* or *names*.
- The M_d's are component models ($\forall d \in D$).
- $I_d \subseteq D \cup \{self\}$ is the set of *influencers* of d. Alternately, the *influencees* of a component could be specified. Both allow one to specify the coupling graph topology.

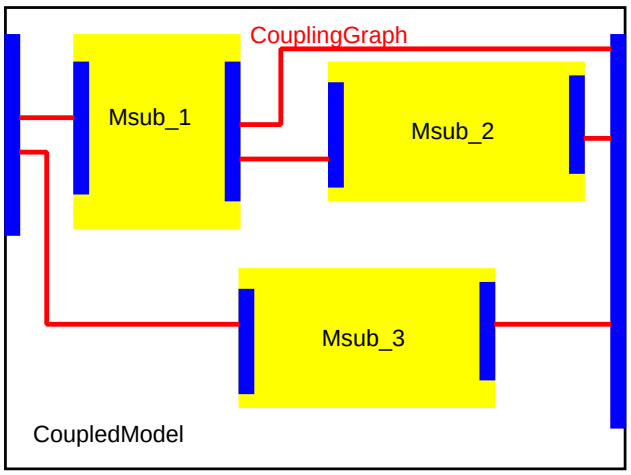

Figure 1.22: Example network model

- $Z_d : \times_{i \in I_d} YX_i \to XY_d$ is the *interface map* for $d \in D \cup \{self\}$

$$YX_i = X_i \text{ if } i = self,\ YX_i = Y_i \text{ if } i \neq self$$
$$XY_d = Y_d \text{ if } d = self,\ XY_d = X_d \text{ if } d \neq self$$

  The interface map allows one to specify mappings or conversions required when for example an output event "departure" of one sub-model is routed to the input of another sub-model. The accepting sub-model may be expecting "arrival" events only. Thus, a "departure"–to–"arrival" mapping must be carried out to allow true sub-model re-use. Sub-models should *not* have to be modified when re-used in a network ! Together, I_d and Z_d specify the network coupling completely.

The above structure does *not* reveal anything about the overall *behaviour* of the network. It only specifies structure. As such, only questions about that structure (number of components, presence of feedback, ...) can be answered at this level. The behavioural semantics of a coupled model is given by specifying a composition procedure, also known as *flattening.* If all the component models are specified using the same formalism, the composition procedure may replace the network by a single model in that same formalism. In that case, the formalism is said to be *closed under composition.* This is obviously a highly desirable property. It is the basis for

- implementing a simulation kernel which implements the simulation of coupled models by orchestrating simulators of the components.
- finding the meaning of *hierarchies* of models by applying the composition procedure recursively.

Compositional modelling may be done (as long as a composition procedure is given) at *any* of the specification levels mentioned before (I/O observation frame, I/O Relation Observation, I/O Function Observation, I/O System). The semantics is given by the elaboration of closure under coupling. Within the I/O System level, there are obviously many formalisms for which a composition procedure may be defined.

Assuming the component models M_d in

$$\langle T, X_{self}, Y_{self}, D, \{M_d | d \in D\}, \{I_d | d \in D \cup \{self\}\}, \{Z_d | d \in D \cup \{self\}\}\rangle$$

are all specified at the I/O System level, implementing closure would mean this model is replaced by

$$\langle T, X_{self}, \Omega, Q, \delta, Y_{self}, \lambda\rangle.$$

In continuous models, composition is achieved by replacing connections by algebraic equalities and combining these with the components' mathematical equations into one large set of equations. In discrete event

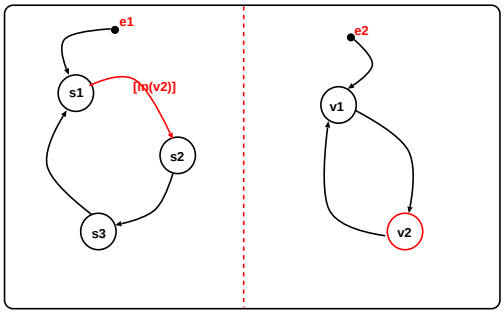

Figure 1.23: Composition of statechart components

models, composition is achieved by describing how different component events are globally *scheduled* in a causality-preserving order. Both cases will be described in more detail in the next chapter.

If structured sets are used to describe input and output sets of the network as well as of the components, connections may be described between specific (named) ports making the mathematical structure more intuitive and closer to the form commonly used in modelling languages such as MSL-USER. We defer presentation of a structured coupled specification until the next section, when non-causal modelling is introduced.

Non-modular multicomponent specification

Often, models in some formalism are graphically specified by connecting building blocks. Though this gives the illusion of being modular, often the transition functions of the building blocks interact *directly*, without going through interfaces. This limits

- the insight into the semantics a user gets directly from the graphical represenation,
- the potential for component-based re-use as a component depends on its environment in unexpected ways,
- the potential for distributed implementation as hidden component interactions need to be passed as messages between distributed processors.

In programming languages, non-modularity corresponds to the use of *global variables*, and to the ensuing *side-effect* functions may have. Object oriented design discourages this in encapsulating information inside objects (instances of classes) and only giving access through well-defined interface methods.

The process interaction discrete event language GPSS [Sch74] is a notable, but popular, example of non-modular model specification. In his 1992 thesis, Claeys [Cla92] has added encapsulation and hierarchy to GPSS, making it more modular. A similar approach, modifying the process interaction world view, was advocated recently by Cota and Sargent [CS92].

Statecharts [Har88, HN96, HG97] are modular in their treatment of external events, but non-modular in their treatment of concurrent (orthogonal) behaviour. In Figure 1.23, modular coupling of the model to its environment is possible via the input-ports e_1 and e_2. The concurrent state automata (concurrency is denoted by the dotted vertical line) however interact in a non-modular fashion: a transition in the left automaton from s_2 to s_1 will occur if the automaton on the right is in state v_2.

The mathematical representation of a nonmodular multicomponent system is

$$MC = \langle T, X, \Omega, Y, D, \{M_d | d \in D\} \rangle$$

where

$$M_d = \langle Q_d, E_d, I_d, \delta_d, \lambda_d \rangle, \forall d \in D.$$

In the above,

- D is a set of component *references* or names,

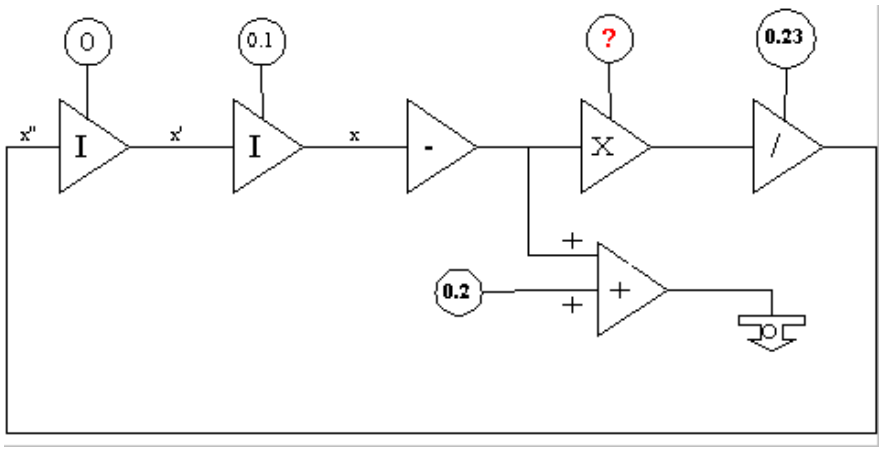

Figure 1.24: A causal block diagram model for the Mass-Spring problem

- Q_d is the *state set* of component d,
- $I_d \subseteq D$ is the set of *influencers* of d,
- $E_d \subseteq D$ is the set of *influencees* of d,
- δ_d is the *state transition function* of d
  $\delta_d : \times_{i \in I_d} Q_i \times \Omega \to \times_{j \in E_d} Q_j$.
  The new states in all of the influencees are determined by the old states in all of the influencers.
- λ_d is the *output function* of d
  $\lambda_d : \times_{i \in I_d} Q_i \times \Omega \to Y$
  Output is determined by the state of all the influencers.

Together, the nonmodular multicomponent system is an I/O System level specification. Its state set is the product set of all component state sets and its transition function is given by combining all individual component transition function. Note how the components are themselves *not* I/O Systems and it is thus not possible to construct a hierarchy of nonmodular multicomponent systems. Also, the particular structure of these systems implies they only make sense at the I/O Systems level.

In Figure 1.24, we show the Mass-Spring model presented when discussing the modelling and simulation process, described in the *causal block diagram* formalism. In the figure, `I` blocks denote Integrators. The other blocks denote algebraic operations. Causal block diagram models may be seen as nonmodular multicomponent systems with

- $E_d = \{d\}$
- $Y = \times_{d \in D} Y_d$
- $Q = \times_{d \in D} Q_d$
- $\delta(q, \omega).d = \delta_d(\times_{i \in I_d} q_i, \omega)$
- $\lambda(q, \omega).d = \lambda_d(\times_{i \in I_d} q_i, \omega)$

1.5.5 Non-causal modelling

Though quite generic, the formalisms presented above at the I/O systems level do only describe *causal* models. When describing physical systems, *non-causal* models express conservation laws and constraints without imposing a *computational causality.* The dynamics of a simple resistor for example (Figure 1.25) is described by Ohm's law

$$V - Ri = 0.$$

Depending on whether the current i through (when connected to a current source) or the voltage drop V over (when connected to a voltage source) the resistor is known, the causal equations

$$V := Ri$$

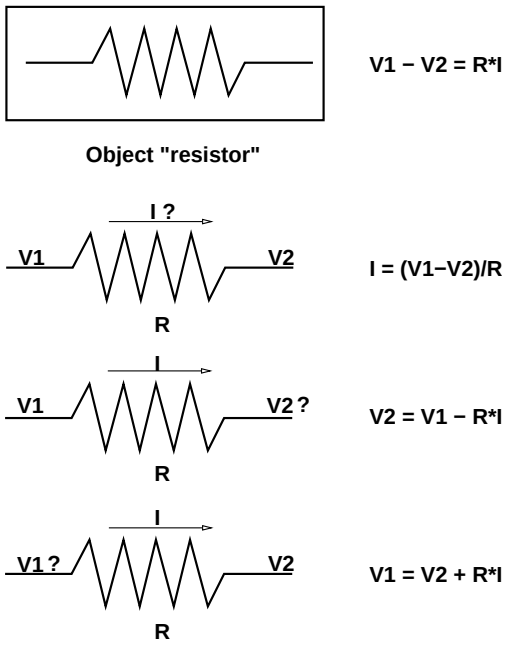

Figure 1.25: Electrical resistor

or

$$i := \frac{V}{R}$$

would be used. Mathematically, those two expressions are evidently equivalent, but if we are interested in writing a *computational block*, with an input and an output, they are distinct. In the case of the resistor, which, as a physical *object*, is non-causal in nature, a choice can not be made a priori as it depends on what the resistor will be connected to. As such, it does not make sense to define two objects (as in Matlab/Simulink), one for each causal orientation. The whole problem stems from the fact that we are dealing with assignment instructions, which we have stressed by using the assignment operator `:=`, and not mathematical equalities. One says that computational blocks are *causally oriented.* From the point of expressiveness and *re-usability* (in different causal contexts), the non-causal, implicit representation is preferred. From a computational point of view however, the causal representation is preferred as solving implicit equations, though possible, is highly inefficient compared to solving their causal counterparts. Inefficiency is mainly due to the iterative nature of implicit solvers. As will be demonstrated in the next chapter, it is in many cases possible to automatically transform a non-causal representation into a causal one.

Most formalisms and corresponding modelling languages are causally oriented, which strongly limits the *re-usability* of sub-models, and in general, their fitness to serve as a basis for the construction of general sub-model languages. Causally non-oriented formalisms and languages (non-causal, for short) exist as well, and some of them will be presented further on.

At the different levels of the above specification hierarchy, it is meaningful to distinguish between input and output ports. Leaving computational causality unspecified, turns the model into a non-causal one. As mentioned before, this increases re-usability. This does require the relationship between the ports to be specified in a non-causal fashion. In continuous models, this is achieved by using implicit mathematical equations. As discrete event formalisms are inherently causal (a cause event schedules an effect event), it is not meaningful to remove causality. Attempts at generalisation are based on logical foundations [RS94].

To allow for both causal and non-causal port specifications, it is sufficient to include causality *type* information in the structured sets describing ports

$$Ports = \langle V, V_1' \times V_2' \times \cdots V_n' \rangle,$$

where

$$V_i' = Type_i \times V_i, \forall i \in \{1,\ldots,n\},$$

and

$$Type_i \in \{Input, Output, InOrOut\}, \forall i \in \{1,\ldots,n\}.$$

If *InOrOut* is disallowed, the above can be used to specify causal interfaces. If *InOrOut* is allowed, a *causality assignment procedure* can correctly replace *InOrOut* by the appropriate causality. This procedure is discussed in the next chapter.

We can now describe a structured, modular, possibly non-causal network specification:

$$SMN = \langle Ports_{self}, D, \{M_i | i \in D\}, C\rangle,$$

where for each $i \in D$, the component models have at least the following information:

$$M_i = \langle T, Ports_i\rangle.$$

Depending on the specification level, more information can be added to these structures.

To uniquely identify a port (of either the overall network or one of the components), a tuple (c, n_c) is needed. c identifies the component (possibly $self$), and n_c is the name or reference of one of the ports in c's structured set of ports (as defined above). The set of all individual ports is

$$AllPorts = \{(c,n) | c \in D \cup \{self\}, n \in proj_1(Ports_c)\}.$$

The *coupling graph* C in the network specification above describes couplings between individual ports

$$C \subseteq AllPorts \times AllPorts.$$

As in Statecharts, the relationships between ports may actually be many-to-many, in which case the graph C becomes a hypergraph

$$C \subseteq 2^{AllPorts} \times 2^{AllPorts}.$$

As before, we may wish to define a port-to-port mapping. In practice, this is only used for causal systems with simple graph connections between (c,n) and (c',n'). In this case, the mapping is described by $Z_{(c,n),(c',n')}$

- if $proj_1(Ports_{self}.n) = proj_1(Ports_{c'}.n') = Input$, $c' \neq self$

$$Z_{(self,n),(c',n')} : proj_2(Ports_{self}.n) \rightarrow proj_2(Ports_{c'}.n')$$

- if $proj_1(Ports_c.n) = proj_1(Ports_{self}.n') = Output$ $c \neq self$

$$Z_{(c,n),(self,n')} : proj_2(Ports_c.n) \rightarrow proj_2(Ports_{self}.n')$$

- otherwise, if $proj_1(Ports_c.n) = Output$, $proj_1(Ports_{c'}.n') = Input$

$$Z_{(self,n),(c',n')} : proj_2(Ports_c.n) \rightarrow proj_2(Ports_{c'}.n')$$

It is noted that in the implementation of the modelling language MSL-USER, the above mathematical structure was used as the basis for the semantics of coupled models.

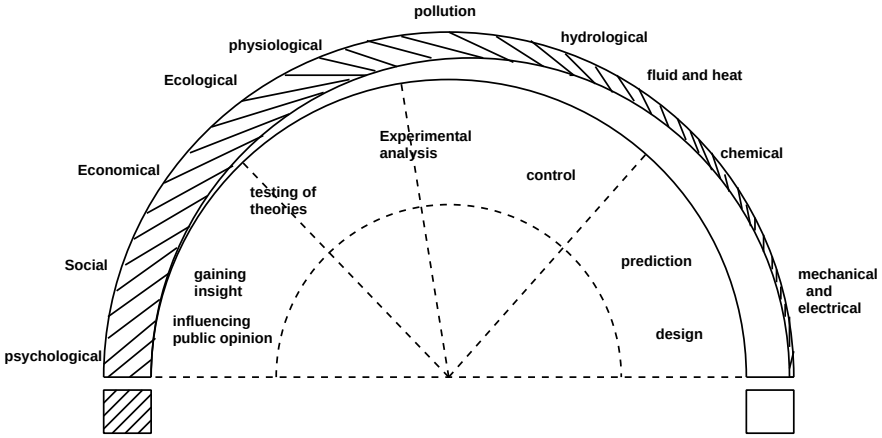

Figure 1.26: Karplus' classification

1.6 Classifications

Previously, the importance of making the *process* of modelling and simulation explicit was stressed. By doing so, different sub-processes could be identified, thus simplifying the generic (application-independent) study of modelling and simulation. Now, a further classification of different types of models, their corresponding simulation kernels, and supporting methods and tools will be presented. The main focus is on classifying models. All other issues are in a sense derived from that. The essence of classification is to group in *equivalence classes* according to a number of criteria. A classification may help in choosing the most appropriate formalism when modelling a system. This in turn may help one choose the most appropriate modelling and simulation tool.

One possible classification make a destinction between *graphical* and *textual* model representations. This is really a tool issue, but the underlying formalism may be more or less amenable to graphical represenation.

A major classification is the distinction between *deterministic* and *stochastic* models. As mentioned before, even stochastic models may be solved by a deterministic simulator.

1.6.1 Application-based classification

The Directory of Simulation Software published yearly by the Society for Computer Simulation is a list of companies and product descriptions. Hundreds are listed, with only a rough, 2-level classification:

- application based: telecom, robotics, resource management, training, process control, power applications, operating systems, networks, manufacturing, industrial engineering, finance, education, chemical, business, biomedical, automotive, batch processes, and aerospace.
- generic: operations research, program generators, graphics, animation, fluid dynamics, discrete event, differential eqation solvers, continuous languages, CAD/CAM, CAE/CASE, and AI/Expert Systems.

This classification does not give much insight in the generic nature of modelling and simulation. However, already, needs and trends can be identified. Most of the software tools are self-contained and closed. To allow exchange and re-use of models as well as inter-operability of simulators, tools must become open and model representation languages standardized.

1.6.2 Ill-definedness classification

At a far more abstract level, Karplus' Arch shown in Figure 1.26 depicts a spectrum of systems ordered according to the amount of a priori information we have about them. On the very right, *white box* systems are characterized by complete insight in their working. Based on general governing laws, a model can *deductively* be derived from a priori knowledge. This usually means we build models at the I/O System level. On the very left, *black box* systems are characterized by a complete lack of knowledge about their internal

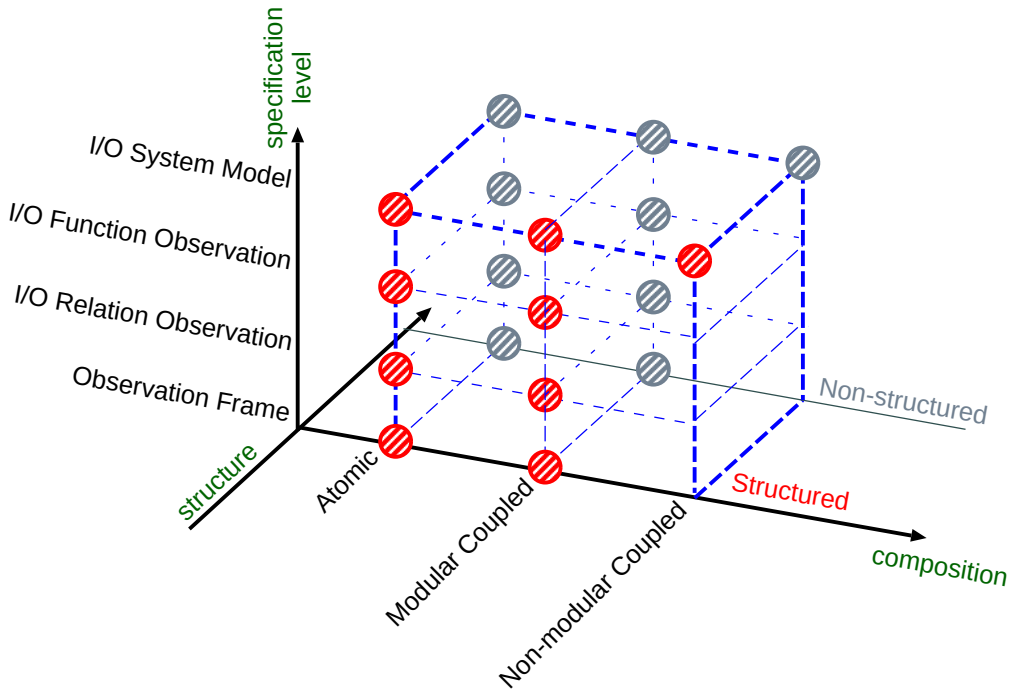

Figure 1.27: Levels of system specification

working. Only input-output relationships can be observed. Initially, we can only build models at the I/O Observation Frame level. Using *inductive* techniques, it may be possible to climb up the specification hierarchy, inferring structure from behaviour, to reach higher levels. Often, this is an iterative process whereby a structure is proposed, and after parameter estimation, simulation results are compared to observation data. This comparison (in particular, discrepancies) may yield new insights, which lead to new hypotheses, *etc.* . In between is a *grey* area, often called *ill-defined* systems. The study of these systems is most challenging as both observation data and limited a priori knowledge are available. As can be seen in the figure, application areas range from electrical and mechanical systems for white box systems, over biological and physiological models for grey systems, to social and psychological systems for black box systems.

1.6.3 System specification classification

In Figure 1.17, we have shown some common types of segments (trajectories). These may be used to characterize model formalisms.

In Figure 1.27, we bring together different classifications presented before. The specification level is one dimension of classification. At each level, structured sets may be introduced, to make the specification structured. At each level, and both for structured and for non-structured formalisms, either atomic or coupled models may be constructed. Whereas modular coupled models can be constructed at each level of the specification hierarchy, non-modular coupled models only make sense at the I/O Sytem level.

1.6.4 I/O System classification

Many formalisms are situated at the I/O System specification level. For these formalisms, the nature (type) of time base T and state set Q allow for classification. Table 1.1 shows a few formalisms classified according to the nature of their time base and state set.

1.6.5 Discrete event world views

Formalisms belong to the *discrete event* category in case the time base is continuous, but only a finite number of *events* occur in a bounded time-interval, and only at those event times does the discrete state of the system

	T: **Continuous**	T: **Discrete**	T: {**NOW**}
Q: **Continuous**	DAE	Difference Equations	Algebraic Equations
Q: **Discrete**	Discrete event Naive Physics	Finite State Automata Petri Nets	Integer Equations.

Table 1.1: I/O system model classification

change. Discrete event models are used when

- the behaviour of physical systems is abstracted by means of time-scale or parameter abstraction [MB01]. In many cases, this leads to queueing models.
- non-physical systems such as software are studied.

Traditionally, several *world views* have been distinguised:

1. Event Scheduling
2. Activity Scanning
3. Three Phase Approach
4. Process Interaction

These will be studied in more detail in the next chapter.

1.6.6 Tool-oriented classification

In the following, a classification is presented which takes into account the relationship between modelling formalisms and representation languages on the one hand and their *applicability* on the other hand. Here, a modelling and simulation tool user's point of view is taken. This is done by means of a simple example, demonstrating different model representations and their respective merits and drawbacks. In its original form, this is due to Françis Lorenz, during Simulation in Europe discussions [VV96c, KVVG94, KVVG95].

The fundamental principle of simulation code generators (simulation CASE tools) is to start from a description of a model that is to be simulated and to produce the corresponding simulation program in a –to the user– automatic way. The comparative study of modelling formalisms and corresponding representation languages is therefore of great importance as the language implemented in a particular code generator, with its advantages and flaws, will determine the features of the generator.

The models considered here assume a hypothesis of parameter aggregation (*lumped parameter assumption*). In mathematical terms, they are represented by algebraic differential equations with one single independent variable, time.

Physical view

The problem we will discuss is a car suspension (Figure 1.28), studied here in one dimension. This implies a model of a quarter of the vehicle, because we only study one of the wheels, supporting one quarter of the weight. We assume all elements are linear as it is not possible to represent non-linearities in some of the formalisms we will discuss. Also, we will not represent the lifting of the tire from the road which would introduce a structural discontinuity which could only be represented in a *hybrid* formalism.

Physical domain specific views

This example is chosen in the domain of mechanics. A mechanical *concept schema* can be drawn as shown in Figure 1.29. One could also consider an electrical equivalent of this system as shown in Figure 1.30.

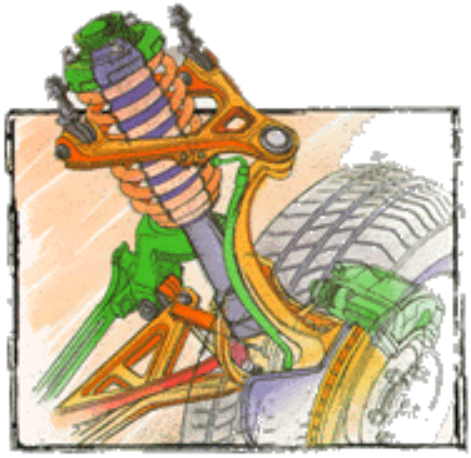

Figure 1.28: Car suspension, physical view

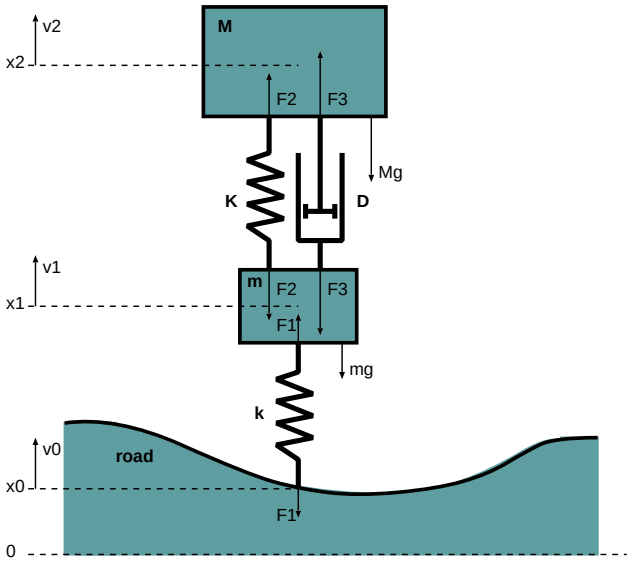

Figure 1.29: Car suspension, mechanical view

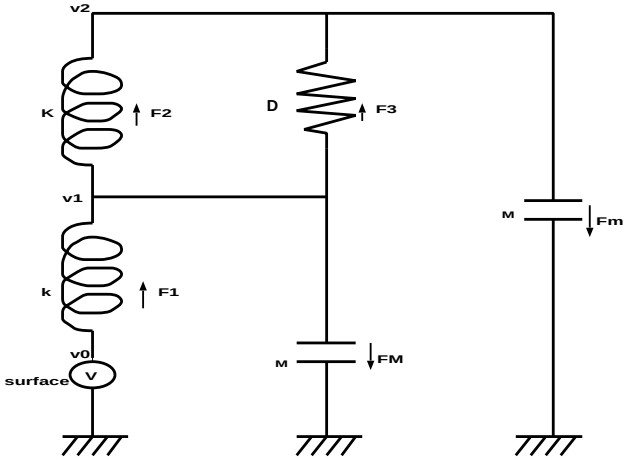

Figure 1.30: Electrical analogy of car suspension

Physical domain	Effort e	Flow f	Momentum p	Displacement q
Electrical	Voltage u [V]	Current i [A]	Flux Φ [V s]	Charge q [A s]
Translational	Force F [N]	Velocity v [m s^{-1}]	Momentum I [N s]	Displacement x [m]
Rotational	Torque T [N m]	Angular Velocity ω [rad s^{-1}]	Angular Momentum L [N m s]	Angle ϕ [rad]
Hydraulic	Pressure p [N m^{-2}]	Volume Flow q [m^3 s^{-1}]	Pressure Momentum Γ [N m^{-2} s]	Volume V [m^3]
Thermodynamical	Temperature T [K]	Entropy Flow $\frac{dS}{dt}$ [W K^{-1}]	—	Entropy S [J K^{-1}]

Table 1.2: Physical analogy

The analogy chosen is *force = current* ("through" variables), *velocity = voltage* ("across" variables). In this case, masses become capacitors and springs, self-inductances. The analogy *force = voltage, velocity = current* could also be used (as shown in the following table). Masses are then self-inductances and springs, capacitors; one also has to invert serial and parallel connections. We will come back to this analogy duality.

The analogy between diffent physical domains is summarized in Table 1.2.

Note that the concept schema is a language for system description. It is not without *ambiguities* (for example, the same zig-zag symbol is used to represent a spring in the context of mechanics and a resistor in the context of electricity) but it is by far the most comprehensible. One can express virtually anything in this language, although some uncertainty sometimes remains in the exact interpretation of the graph. This language is therefore not totally rigourous and complete. In this respect, where does the schema say that all elements are linear ? The computational causality need not be specified, but there is still much work left to do before simulation code can be generated. In particular, transforming a concept schema into a mathematical formulation is not straightforward.

Equations

The most "natural" (or at least the one that is designated as such, for simulation purposes) method to present our suspension system is simply by writing down the mathematical equations.

$$\begin{cases} F_1 = k.[I - (x_1 - x_0)] \\ F_2 = K.[L - (x_2 - x_1)] \\ F_3 = -D.(v_2 - v_1) \\ \\ \ddot{x}_1 = (F_1 - F_2 - F_3)/m \\ \ddot{x}_2 = (F_2 + F_3)/M \end{cases}$$

One sees the system is of the fourth order (two second order equations). Note that the position (x_0) is unknown. Also note that it has not been explicitly stated that v_1 and v_2 are the first derivatives of x_1 and x_2. Though this relationship may seem obvious to a human, a code generator will only be able to infer this relation with difficulty. To the generator, v_1 and v_2 will be "not computed" (reflected by issuing an error message) or will be considered as unknowns.

Even an example as simple as this can be represented in various ways depending on the analyst and his "style".

Below, another representation of the problem is presented, which uses other variables. Note that this time

the velocity (v_0) is unknown. The system is of course still of fourth order, but each of the four variables chosen is only integrated once (the accelerations of the two masses and the velocities of depressing the two springs), contrary to the integration of two times two variables (the accelerations of the two masses). This formulation is more directly usable for simulation.

It goes without saying that this variety in formulation does not favour the readability (and hence, re-usability) of large models.

$$\begin{cases} F_1 = k.\Delta I \\ F_2 = K.\Delta L \\ F_3 = D.\Delta v_{12} \\ F_m = F_1 - F_2 - F_3 \\ F_M = F_2 + F_3 \\ \dot{v}_1 = F_m/m \\ \dot{v}_2 = F_M/M \\ \Delta v_{01} = v_0 - v_1 \\ \Delta v_{12} = v_1 - v_2 \\ \Delta \dot{I} = \Delta v_{01} \\ \Delta \dot{L} = \Delta v_{12} \end{cases}$$

Using equations, one can express a plethora of laws of behaviour, but the discontinuities are not easily representable in this "language". The language is causally free (non-causal), but the re-use of sub-models requires an almost complete re-examination of each new situation.

At the mathematical level, it is possible to write equations which are not physically meaningful (*e.g.*, do not conserve energy, or lead to negative concentrations).

State equation

The second set of equations leads us to state equations (state-space form). The four variables to integrate are called state variables and are organised into a vector (in fact, a column matrix). The equations are then written in matrix form.

$$\underline{\dot{X}} = \underline{A}.\underline{X} + \underline{B}.\underline{U}$$

$$\underline{A} = \begin{bmatrix} \frac{-D}{m} & \frac{D}{m} & \frac{k}{m} & \frac{-K}{m} \\ \frac{D}{M} & \frac{-D}{M} & 0 & \frac{K}{M} \\ -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \quad \underline{X} = \begin{bmatrix} v_1 \\ v_2 \\ \Delta I \\ \Delta L \end{bmatrix} \quad \underline{B} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \underline{U} = [v_0]$$

This time, the language is causally oriented. We find again the previous characteristics, however amplified by a stricter environment. However, the state equation is relatively easy to solve (by matrix multiplication), except in non-linear cases (the matrices depend on the state vector). Furthermore, it is possible to analyse the eigenvalues of the matrix $\underline{A}$, which yields valuable information about the system's dynamics.

As at the mathematical level, it is possible to write state-space equations which are not physically meaningful.

Algorithm

We can express the model in the form of a directly usable algorithm. The algorithm below is the transcription in FORTRAN of the above equations.

```
C---+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
C
      SUBROUTINE SUSPEN ( V0,
     +                    V1, V2, DL01, DL12,
```

```
     +                   V1D, V2D, DL01D, DL12D,
     +                   K01, K12, D12, M1, M2 )
C
C INPUT VARIABLE, STATE VARIABLES
      REAL*8 V0,
     +       V1, V2, DL01, DL12
C
C OUTPUT VARIABLES
      REAL*8 V1D, V2D, DL01D, DL12D
C
C PARAMETERS
      REAL*8 K01, K12, D12, M1, M2
C
C INTERNAL VARIABLES
      REAL*8 DV01, DV12, F1, F2, F3
C
C-------------------------------------------------------------------------------
C
      DV01 = V0 - V1
      DV12 = V1 - V2
C
C SPRINGS, SHOCK DAMPING
      F1 = K01 * DL01
      F2 = K12 * DL12
      F3 = D12 * DV12
C
C MASSES
      V1D = (F1 - F2 - F3) / M1
      V2D = (F2 + F3)      / M2
C
      DL01D = DV01
      DL12D = DV12
C
      RETURN
      END
C
C-------------------------------------------------------------------------------
```

It is clear that one may describe behaviour laws in algorithmic form, but, as with mathematical and state-space models, no fool-proof mechanism is present to guard one from writing physically meaningless models. It is therefore a dangerous expression form. Moreover, it is causally oriented. The current practice of presenting a library of FORTRAN sub-programs as a library of sub-models is therefore to be avoided. Also, this representation does not allow for symbolic analysis of the model.

Transfer function

Another fashionable way of expression is the transfer function, obtained by the Laplace transform of the equations. They are written here for an unknown velocity; one could have written them for an unknown position as well.

$$\frac{v_1}{v_0} = \frac{k(Ms^2 + Ds + K)}{mMs^4 + D(M+m)s^3 + (mK + MK + Mk)s^2 + kDs + kK}$$

$$\frac{v_2}{v_0} = \frac{k(Ds + K)}{mMs^4 + D(M+m)s^3 + (mK + MK + Mk)s^2 + kDs + kK}$$

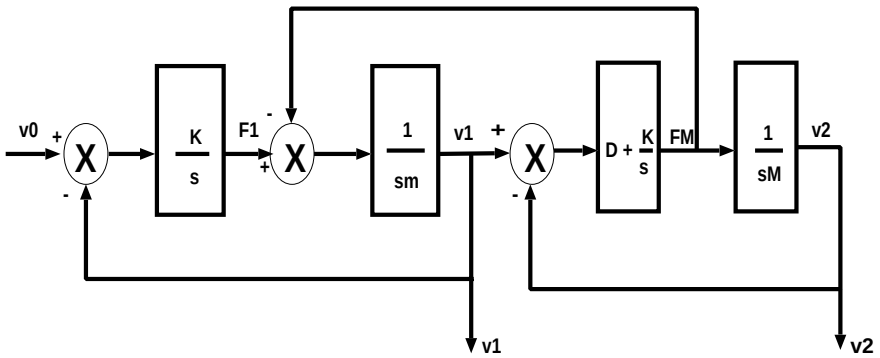

Figure 1.31: (Transfer Function) block diagram

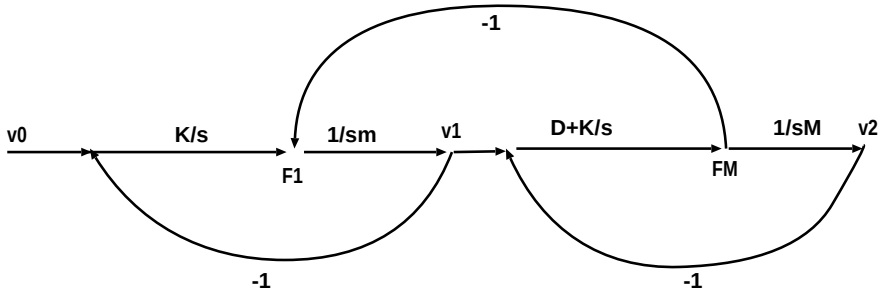

Figure 1.32: Signal flow graph

This way of representation is only usable in linear cases, which strongly limits its applicability. The readability of the formulas is poor and they are causally oriented (the transfer function pushes the input-output view of the system to its extreme). They are re-usable in a certain range, but most of the time it is easier to re-study a deviating system than it is to modify preceding work. Nevertheless, the utilities developed by control-specialists on the basis of this formalism are numerous and extremely useful (*e.g.*, the study of dynamics and stability on Bode, Nyquist or Black graphs, the location of Evans poles, *etc.*), which largely justifies its use in the context of process control problems.

Block diagram

The *causal block diagrams* were initially nothing but the graphical representation of the transfer functions (Figure 1.31).

As a graphical representation of the transfer functions, block diagrams strongly improve readability, but leave the other characteristics unchanged. Nevertheless, when used in the *temporal domain*, they become a graphical representation of equations, with all advantages involved. Unfortunately, they also impose a computational causality (as in Matlab/Simulink) on the equations, decreasing their re-usability.

If block diagrams are given a non-causal semantics, this disadvantage vanishes. *Non-causal block diagrams* are called *generalised* block diagrams by Cellier [Cel91].

Signal flow graph

The signal flow graph is another graphical representation of the transfer functions (Figure 1.32). It is therefore very similar to the block diagram.

The signal flow graph is only used in the Laplace plane, and is therefore limited to linear systems. Its characteristics are more or less those of transfer functions. It however adds to the arsenal of analysis tools, like the Mason formula, based on a study of the system topology.

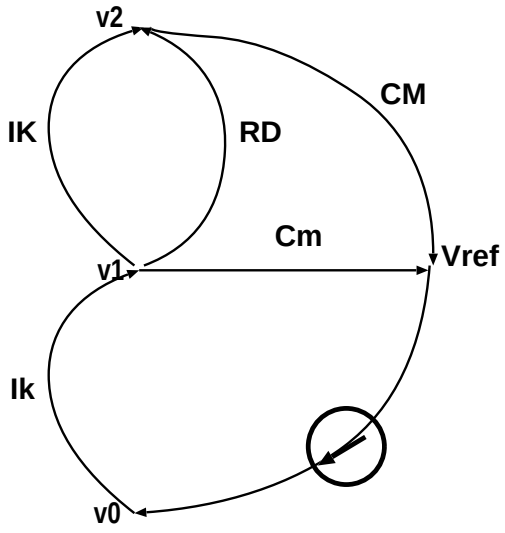

Figure 1.33: Linear graph

Linear graph

The linear graph is in a way an equivalent electrical schema of the system, relieved of all electrical connotations. It suffices to compare the linear graph model of our car suspension in Figure 1.33 to the equivalent electrical concept schema shown above. This method is applicable in linear as well as in non-linear cases. Its principal advantage lies in the fact that it allows for a causally free representation and it also proposes an automatic method for attributing computational causalities called “normal tree search”, based on the topological analysis of the model to allow for automatic code generation. In other words, it allows for the creation of real libraries of sub-models and also delivers the necessary tools to use them.

Note that this topological analysis algorithm does not only determine causalities. It also allows the detection of certain modelling errors and certain numerical difficulties (algebraic loops, dependent state variables). These possibilities are present because the language is based on physical concepts (potentials, fluxes, inertial effects, capacitative, dissipative) instead of mathematical concepts (variables, operators). In the model presented here, we have retained the analogy *force = flux* (current) and *velocity = potential* (voltage). We could just as well have chosen the dual analogy.

Bond graphs

In the linear graph method, the “parallel” and “serial” links (note that these notions are not as trivial to apply in the mechanical domain) are represented by combining the graph elements serially or in parallel. The bond graph formalism [Cel91, Bro90, Bre84, LW95, SB95] is in principle very similar to the linear graph formalism. In particular, it is based on the same physical concepts, but *all* aspects of the problem are represented here as graph nodes, including its topology: the “0” nodes correspond to “parallel” connections and the “1” nodes correspond to “serial” connections (Figure 1.34).

This method yields a real formalization of the circuit topology. The structure of the bond graph represents the *physical meta-structure* of the system under study, different by nature from the *technological structure* of the assembly of system constituents. Once the principles of the language have been assimilated, it allows one to understand and master the most intricate physics phenomena (albeit under the lumped parameter assumption).

This time we have abandoned the analogy *force = flux* and *velocity = potential*, although it remains possible, in favour of the analogy *force = potential* and *velocity = flux*, more common in bond graphs (due to their origin in mechanical engineering). Masses are then inertial elements and springs capacitative elements. This duality of analogy, already mentioned, should not trouble the analyst. In particular, the dualisation of a

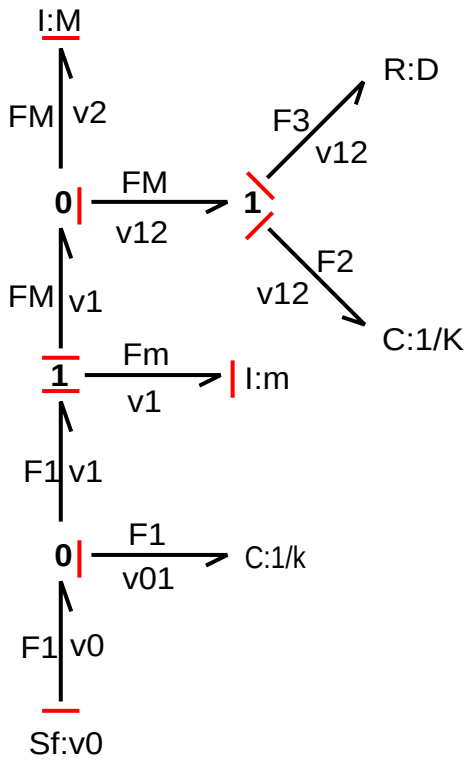

Figure 1.34: Causal Bond Graph

bond graph is extremely easy: it suffices to replace all elements by their duals ($I \leftrightarrow C, O \leftrightarrow 1, \cdots$) but the meta-structure of the model stays unchanged!

This method, like the linear method, is applicable in linear as well as in non-linear cases. The representation it allows for is also causally free and it proposes an automatic method of attributing computational causalities (equivalent to the linear method) as well, which also detects the same modelling errors and numerical difficulties.

The graph has again been constructed for an unknown velocity and one can not construct one for an unknown position.

Comparison

In Table 1.3, the formalisms (representation languages) presented are compared in a more structured way, on the basis of the following criteria:

- *Popularity*: general knowledge of the method with engineers and scientific researchers.
- *Readability*: ease with which someone familiar with the method can understand models composed by someone else.
- *Expressiveness*: possibility to describe non-linear systems and to use complex functions (pure delay, tabulated functions, ...) or other special functions (dead zone, hysteresis, ...).
- *Discontinuous models*: possibility to describe functional and structural discontinuities.
- *Structured variables*: possibility to organize model variables and matrices in record complexes.
- *Modularity*: possibility to hierarchically decompose models into sub-models.
- *Re-usability*: possibility to re-use previously composed models in the context of different problems.
- *Adaptability*: possibility to modify models previously constructed to create a variant of the model (without drastically changing the model).
- *Numerical analysis*: existence of means of analysis linked to the method and yielding information of numerical nature on the model studied.

	Popularity	Readability	Expressiveness	Discontinuous models	Structured variables	Modularity	Re-usability	Adaptability	Numerical analysis	Topological analysis
Concept schema	⊕	⊕	+	+	+	⊕	⊕	+	⊖	⊖
Equations	⊕	-	+	-	+	+	□	□	⊖	⊖
State equations	+	-	□	⊖	+	□	-	-	⊕	⊖
Algorithms	+	-	⊕	□	⊕	□	□	-	⊖	⊖
Transfer functions	⊕	-	-	⊖	⊖	□	□	-	⊕	⊖
Block diagrams	⊕	□	+	□	+	⊕	□	+	+	-
Signal flow graphs	-	□	-	⊖	-	+	□	□	+	+
Linear graphs	+	+	+	-	-	+	+	⊕	□	⊕
Bond graphs	-	+	+	-	+	⊕	⊕	⊕	□	⊕

Legend: ⊖: very bad; −: bad, □: neutral; +: good; ⊕: very good

Table 1.3: Comparison of representation formalisms/languages

- *Topological analysis*: existence of means of analysis linked to the method and yielding information of topological nature on the model studied.

Formalism/language levels

The comparison makes explicit that none of the languages is universally ideal. Each language has its advantages and its flaws. Moreover, for those of the languages for which we have found many flaws and only a few advantages, it may very well be that those advantages are of such importance (large weight factor) that they justify the use of the language in their own right. This is for example the case with transfer functions, which have few advantages according to our criteria, but which are best suited for the study of process control, due to the powerful analysis methods available.

It would thus be useless to search for the “best” language. This does not exist, or rather each language could reveal itself as the best in the context of a specific problem for a specific class of users.

Instead of the segregational view of “better” and “worse” languages, we prefer a more qualified view in which one looks for the optimal use that can be made of these ways of representation. This has led to the classification of the languages using five levels, each level built on top of lower levels, but each level having its use with respect to the problem posed (Table 1.4). I/O data refers to trajectory data generated from simulations. The algorithmic levels refers to causal models. The mathematical level refers to non-causal models. The physical level refers to non-causal models with physical properties (such as energy conservation) built into the formalism. The technological level refers to representations incorporating technological information which may not reveal any information about the physics of the system.

On the basis of this reflection, one can envision a multi-formalism modelling and simulation environment,

Level	Formalism
Technological	Concept schemas
Physical	Linear graphs Bond graphs
Mathematical	Equations State equations Transfer Functions Non-causal block diagrams Flow graphs
Algorithmic	Causal block diagrams CSSLs/DSblock FORTRAN/C
I/O data	Trajectory data

Table 1.4: Classification of representation formalisms/languages

in which the user has a choice of formalisms (and corresponding languages) depending on the sub-model he wishes to construct and in which he can freely assemble sub-models. This environment would be able to use one of the "solvers" currently available as a numerical kernel.

The classification of Table 1.4 is presented in three dimensions in Figure 1.35. It shows the position of different languages/formalisms with respect to application domain, formalism class (paradigm), and description level.

1.7 Multi-formalism modelling

Though coupled models were introduced earlier, we implicitly assumed that the components were all described in the same formalism. Furthermore, to meaningfully associate behavioural semantics to a coupled model, closure under coupling of the components' formalism was assumed. In *complex* systems, this assumption may no longer be true. These systems are characterized, not only by a large number of components, but also by the diversity of the components. One of the observations of the European Commission's ESPRIT Basic Research Working Group 8467 [VV96c] "Simulation for the Future: New Concepts, Tools and Applications" was that for the analysis and design of such complex systems, it is no longer sufficient to study the diverse components separately, using the specific formalisms these components were modelled in. Rather, it is necessary to answer questions about properties (most notably behaviour) of the whole system.

To focus the attention, Figure 1.36 gives an example of a complex system. The complexity lies in the diversity of the different components, both in abstraction level and in formalism used:

- A paper and pulp mill produces paper from trees with polluted water as a side-effect. This system is modelled as a process interaction discrete-event scheduling system (in particular, in GPSS [Gor96]).
- A Waste Water Treatment Plant (WWTP) takes the polluted effluent from the mill and purifies it. Some solid waste is taken to a landfill whereas the partially purified water flows into a lake. This system is modelled using Differential Algebraic Equations (DAEs) describing the biochemical reactions in the WWTP.
- A Fish Farm grows fish in the lake which feed on algae which are highly sensitive to polluted water. The water is also used for a tree plantation which supplies the paper mill. This system is modelled using the System Dynamics formalism. The dotted feedback arrow from the fish farm to the paper mill indicates the possible disastrous impact of poisoned fish on the productivity of workers in the mill.

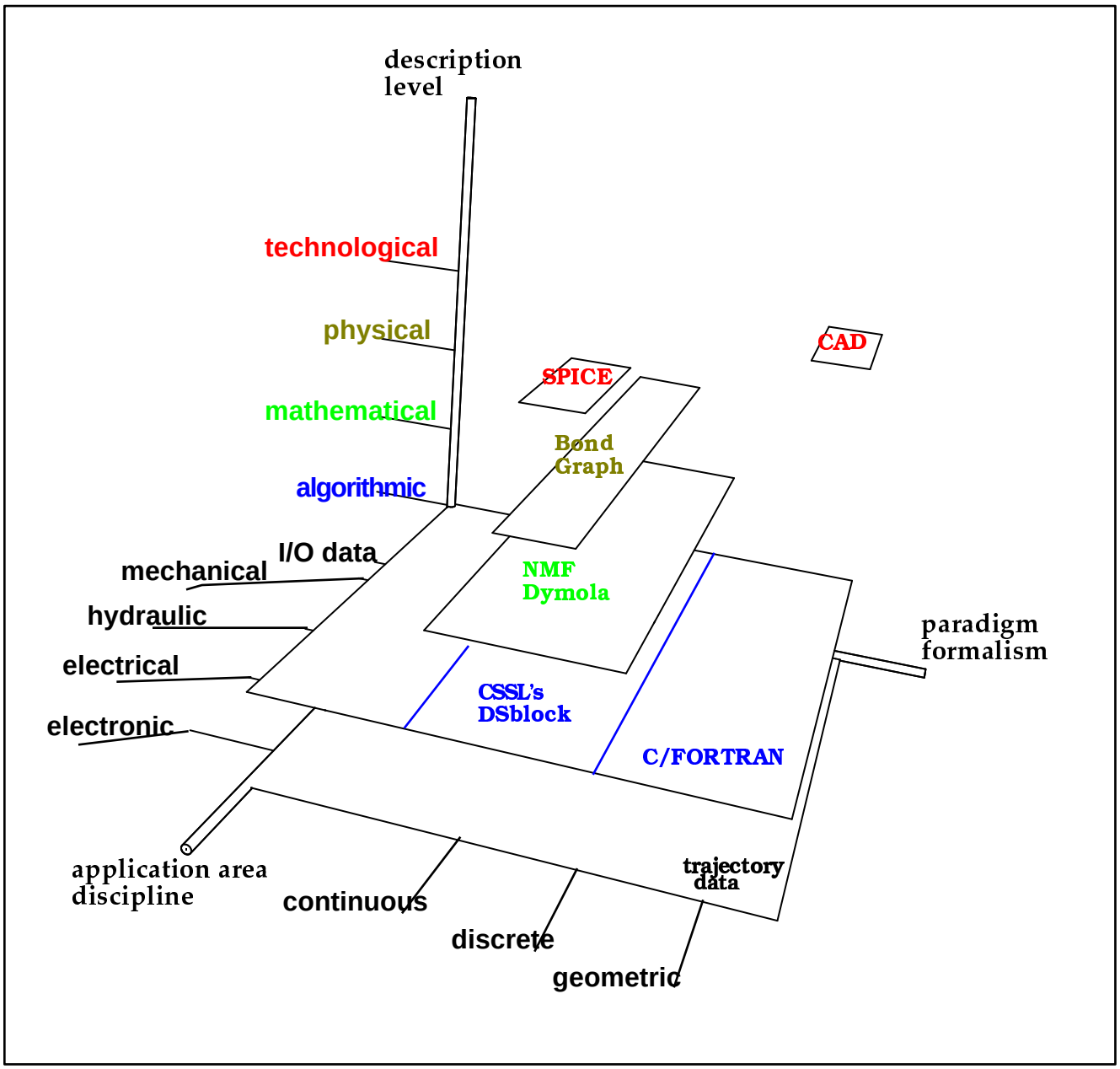

Figure 1.35: Description level/formalism/application domain classification

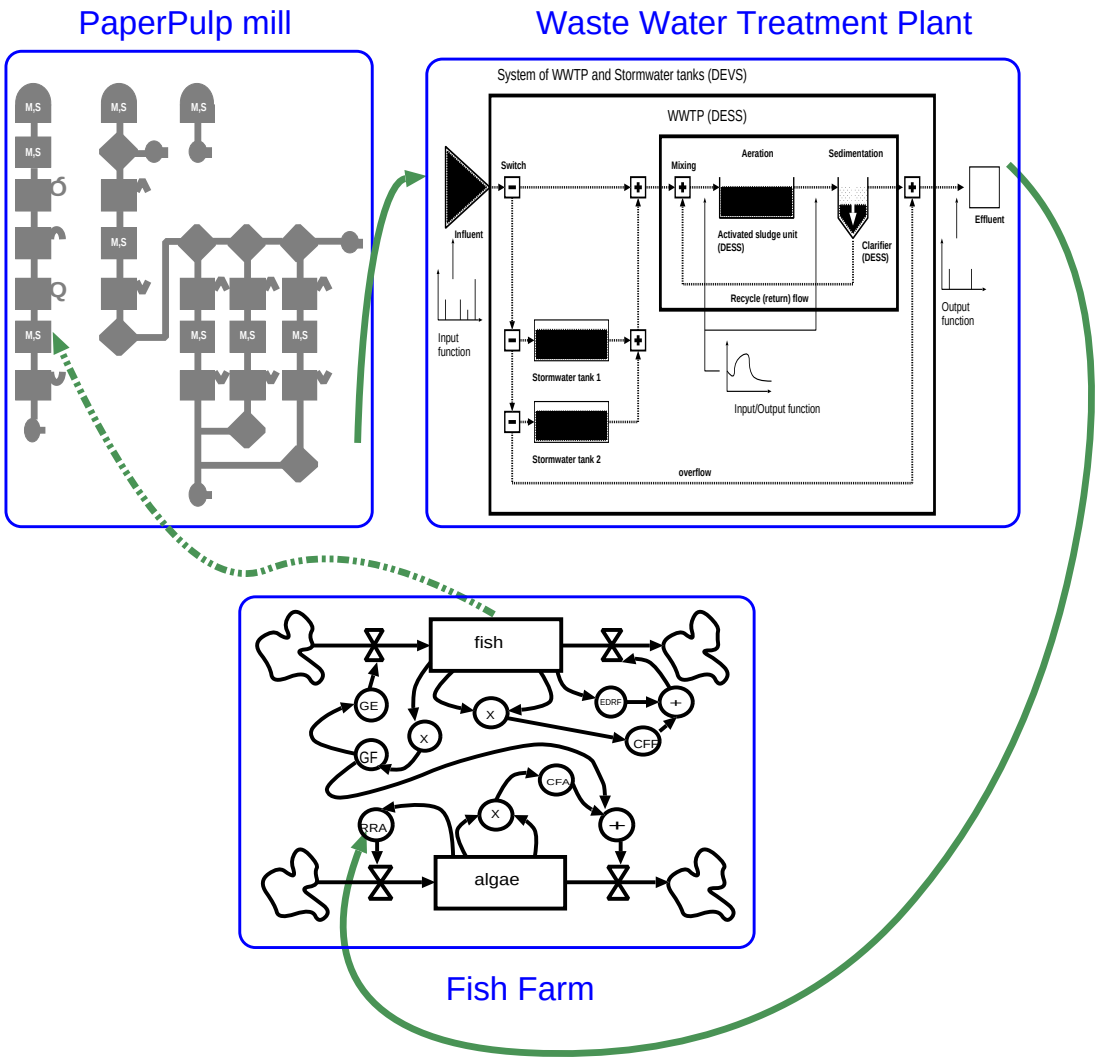

Figure 1.36: Complex system example

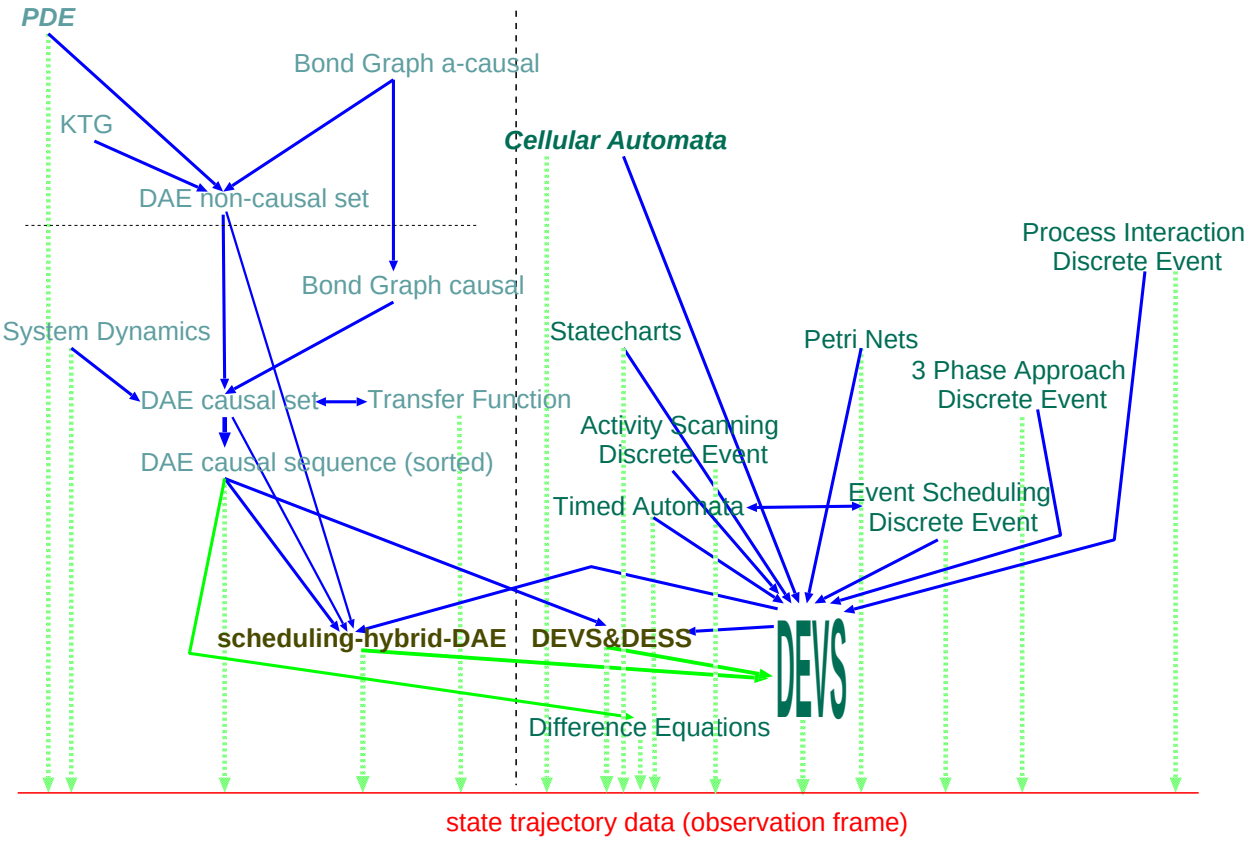

Figure 1.37: The Formalism Transformation Graph (FTG)

It is obvious that decision-making for this system will require understanding of the behaviour of the overall system. Studying the individual components will not suffice. The complexity of this system and its model is due to

- the number of interacting, coupled, concurrent components. Complex behaviour is often a consequence of a large number of feedback loops.
- the variety of components such as software/hardware, continuous/discrete.
- the variety of views, at different levels of abstraction.

A model of a system such as the one described above may be valid (within a particular experimental context) at a certain *level of abstraction*. This level of abstraction, which may be different for each of the components, is determined by the available knowledge, the questions to be answered about the system's behaviour, the required accuracy of answers, *etc*. Orthogonal to the choice of model abstraction level is the selection of a suitable *formalism* in which the model is described. The choice of formalism is related to the abstraction level, the amount of data that can be obtained to calibrate the model, the availability of solvers/simulators for that formalism as well as to the kind of questions which need to be answered. Milner, in his Turing Award lecture [Mil93] rejects the idea that there can be a unique conceptual model, or one preferred formalism, for all aspects of something as large as concurrent systems modelling. Rather, many different levels of explanation, different theories, languages are needed. We believe this view is amplified when arbitrarily complex systems are studied. In our example, different formalisms are obviously used.

In the next chapter, we will elaborate on the semantics of multi-formalism models. As an introduction, we present the Formalism Transformation Graph (FTG) in Figure 1.37. The nodes in the graph are formalisms. The horizontal line at the bottom denotes the I/O Observation Frame at which trajectories can be described. The vertical dotted arrows denote the availability of a simulator for a formalism, producing behaviour from a model in that formalism. Other arrows indicate the existence of behaviour-conserving formalism transfor-

mations. This will be detailed later. The vertical dashed line demarcates continous model formalisms (on the left) from discrete model formalisms (on the right). The FTG shows a plethora of formalisms, indicating that in general, many classifications are possible. It suffices to annotate the nodes in the FTG with attributes (possibly derived from the formalism structure) and determine equivalence classes based on those attributes.

1.8 Modelling the modelling and simulation process

Models, experiments, input and simulation data are different kinds of knowledge manipulated during the course of the modelling and simulation process. They are the entities whose life-cycle we are interested in. The dynamic evolution (over time) of these entities is *de*scribed in the form of a *process model* [HK89]. If a process model is detailed enough, it may be used to *pre*scribe the dynamics of the modelling and simulation enterprise and form the basis for the design and implementation of Computer Aided Modelling and Simulation tools.

There are different levels of impact of a process description:

1. A process is *de*scribed: this is the WEST++ situation; all the actions a user performs in setting up an experiment are saved in the form of a `Tcl` [Ous94] script. Thus, we have a description of what the user did. Note that this script can be used to "replay" the experiment.

2. A process is *pre*scribed: in a process description or *process program* we can provide a template for the user process, *i.e.*, the steps the user has to go through. Such a prescription can more or less restrict (*i.e.*, protect from making mistakes) the user. It does allow the user to be creative however. From a process description or *process program*, a concrete process can be enacted.

3. A process is *pro*scribed: we describe what should *not* be done. This knowledge should also be put into the process description (as assertions or constraints).

A common misconception about process descriptions is that they are sequential rather than parallel. This is in contrast to the parallel nature of many of the processes. A process description need not be sequential if the process description language (such as Petri Nets) is powerful enough to support the description of concurrent processes as well as synchronisation. The scripts (in case of WEST++) generated from the process description must be such that their enactment is parallel, with synchronisations.

1.8.1 Modelling and simulation process models

In Figure 1.5, a simple model of the modelling and simulation process was presented. This informal process model may be refined, as depicted in Figure 1.38. On the extreme left, the real-world process of experimentation is shown. This process may make use of abstract model knowledge during the course of *experimental design.* On the extreme right, the refined modelling and simulation process is shown. The double horizontal bars indicate the possibility of performing multiple instances of the tasks between two sets of double bars in *parallel.* The (matching) second set of double bars, if present, denotes *synchronisation.* One may for example wish to implement and test multiple models matching a given Experimental Frame concurrently. Eventually, those models which pass the tests are collected and model calibration can commence. In the centre, the blocks denote information sources/destinations for the process.

In analogy with software processes, the modelling and simulation process never ends. Models are constantly refined, formulated problems change, different types of simulation experiments are needed using the same model, *etc.* From time to time, this process communicates tangible results in the form of model (base) and simulation output (*e.g.*, performance metrics) "releases" (the deliverables). As requirements (in software parlance) change, the Experimental Frame is adjusted. As with software, it is necessary to manage this form of evolution by means of *version management* (see Figure 1.39). As this problem is not unique to modelling and simulation, standard version management tools such as RCS and CVS [BB95] can be used.

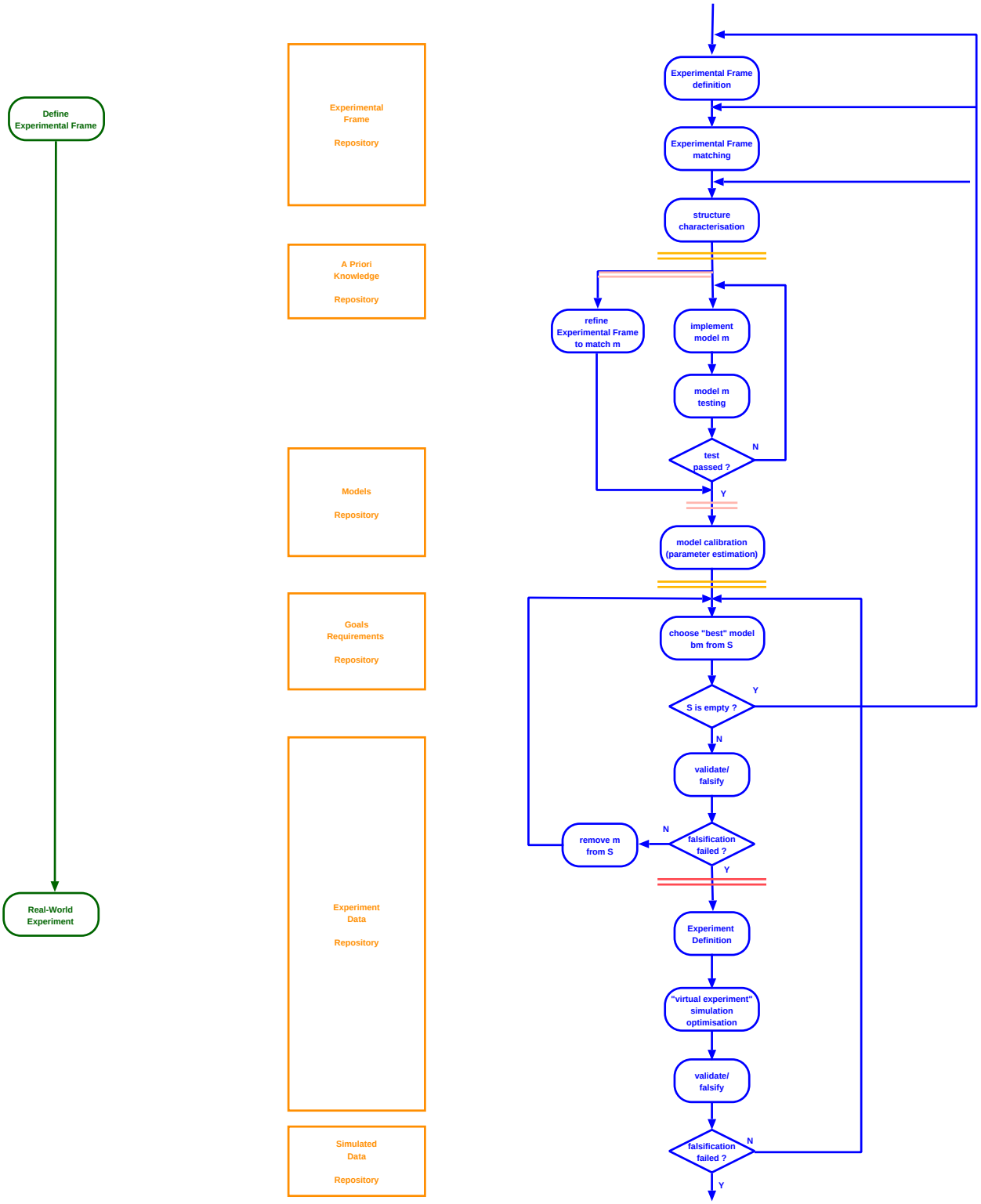

Figure 1.38: M&S Process Detail

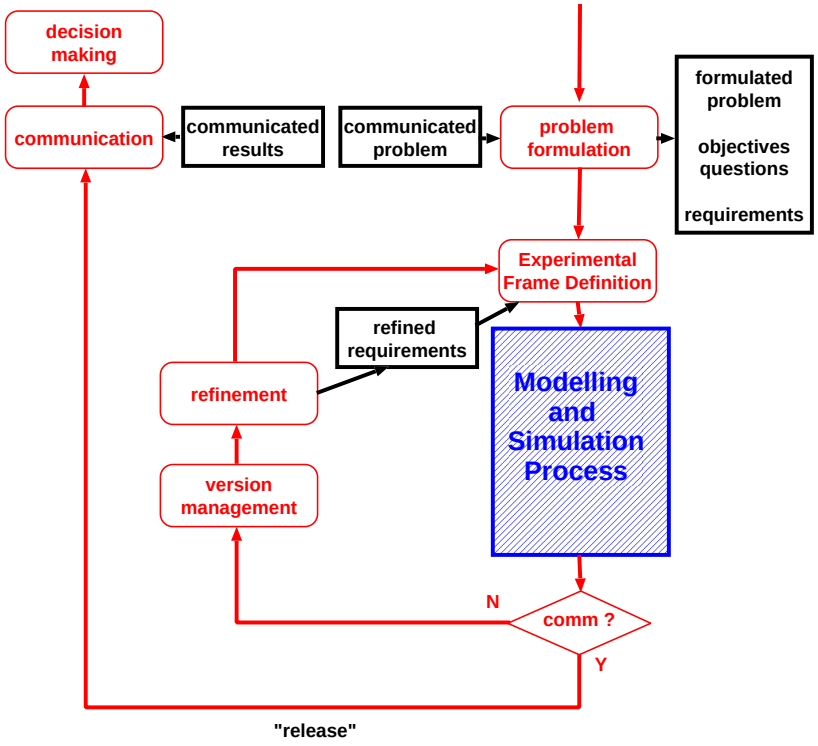

Figure 1.39: Versions and Releases of Models

1.8.2 The Virtual Product Life-cycle

In Figure 1.40, the process underlying MoSS-CC, a scripted extension of WEST++ and its software implementation is depicted. The process is called the Virtual Product Life-cycle (VPL) [VVV+94] as it describes (and partially prescribes) the evolution of the system-to-be-designed ("virtual" product as all experiments are conducted through simulation). The process consists of two major phases:

1. The design phase during which structural decisions are made. Mostly, given an average input and desired output for the system to be built, steady-state models are used to determine structural parameters (such as dimensions).

2. The dynamic analysis phase, during which behavioural choices are made (optimal parameter choice, controller tuning, *etc.*) through forward simulation.

The rationale behind MoSS-CC is that at each phase in the process, system and cost models are linked. Design decisions are based on "minimal cost" considerations. In the first phase, only investment cost are considered. In the second phase, exploitation (running) costs are added. When exploitation costs, obtained from dynamic analysis, are sufficiently high compared to investment costs, it may be necessary to revise structural decisions made in the first phase.

At the core of MoSS-CC is the realisation that both system behaviour and cost are explicitly represented in the form of models which are subsequently used for simulation.

The cycle in both processes denotes an iterative process for which core support is given by the WEST++ interactive modelling and simulation environment [VCV98].

The process (see Figure 1.41) iterates over

1. (interactively) building a model from basic building blocks ("AND" choice as multiple blocks are combined);

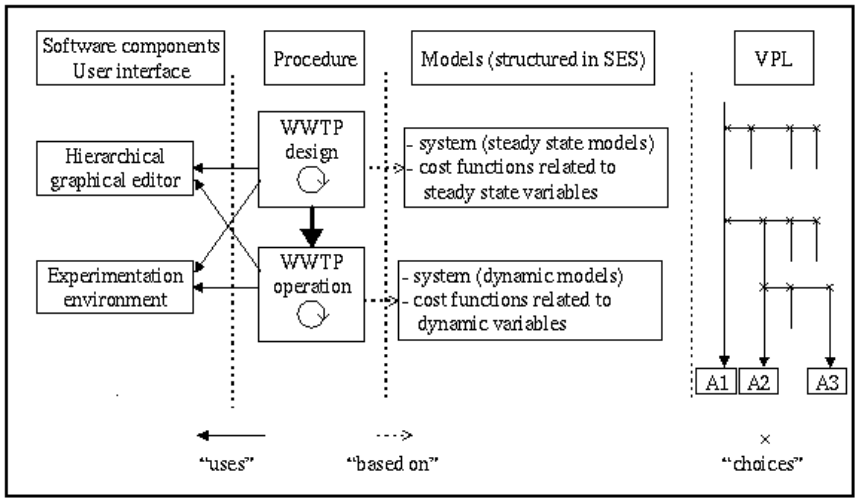

Figure 1.40: MoSS-CC design procedure

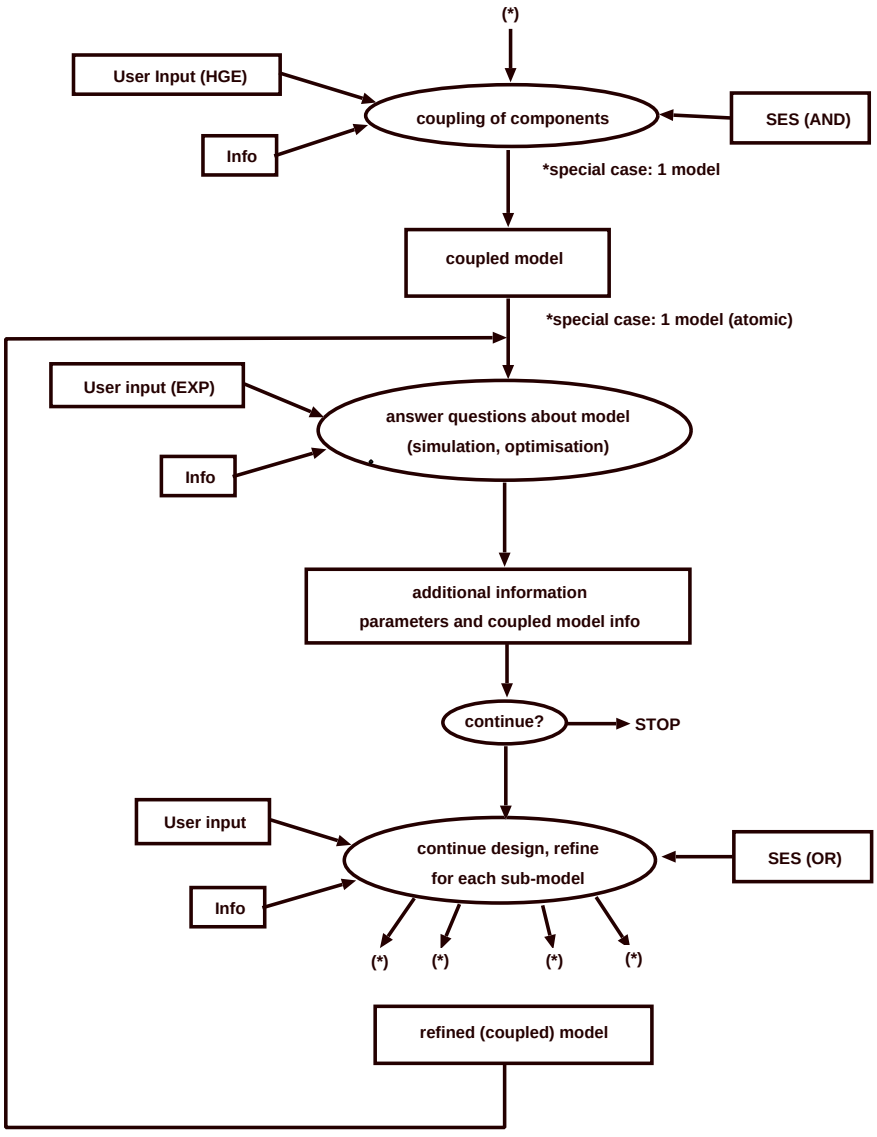

Figure 1.41: VPL for SES-based design

2. through experimentation (simulation, model calibration, optimisation), answer questions about this model;

3. either the process stops here or further refinement is needed. In case of refinement, the model, augmented with the results of the experimentation, allows one to choose from a number of alternatives ("OR" choice) for each of the sub-models. Note that here, the process is recursively instantiated for each of the sub-models, as denoted by the (*) references in Figure 1.41. Sub-model refinement may be carried out in an iterative fashion (one after the other), or concurrently. In the design phase, an "OR" choice corresponds to alternative structural choices. In the dynamic analysis phase, this typically corresponds to choosing from different levels of biological detail (*e.g.*, IAWQ ASM1 or ASM2, see the last chapter).

A System Entity Structure (SES) [Zei84a], a tree-shaped knowledge structure in which AND and OR alternatives are listed, provides the choice space for the user. In essence, the SES encodes design and modelling knowledge.

In the *VPL tree* data structure, the MoSS-CC environment keeps track of all choices made during the evolution of the process described above. This allows the user (by means of a VPL browser) to trace back to any previous choice and to try other alternatives. Thus, arbitrary feedback is added to the process and scenario analysis, whereby consequences of different choices are compared.

Summary

In this chapter, modelling and simulation concepts were introduced. It was noted how it is as important to study the modelling and simulation process as it is to study the objects that process acts upon: the models. The various steps in this process were discussed. In particular, verification and validation were defined. Different types of modelling errors and a unified representation for these were presented. To formalize the structure of abstract models, levels of system specification were introduced. This provides a rigourous basis for a classification of modelling formalisms. A variety of classifications was presented, including a tool-oriented view. Eventually, this leads to the concept of multi-formalism modelling, needed to describe the structure and behaviour of truly complex systems. Finally, the modelling and simulation process is elaborated. This leads to the concept of a Virtual Product Life-cycle, a generic description of a class of modelling and simulation processes.

2
Formalisms

In this chapter, the structure of diverse formalisms is presented. These formalisms are nodes in the Formalism Transformation Graph introduced in the previous chapter. Where applicable, the behaviour-conserving transformation between different formalisms is described.

2.1 The Data formalism

In the previous chapter, the concept of *segments* was introduced. A segment describes the evolution of an entity over time. As such, it can be used to describe trajectories of input, output, and state of a system. In essence, any simulator maps an abstract model onto a behaviour at the data (trajectory) level.

2.2 Discrete event formalisms

For a class of formalisms labelled *discrete-event*, system models are described at an abstraction level where the time base is continuous ($\mathbb{R}$), but during a bounded time-span, only a *finite number* of relevant *events* occurs. These events can cause the state of the system to change. In between events, the state of the system does *not* change. This is unlike *continuous* models in which the state of the system may change continuously over time.

Discrete-event formalisms are clearly at a high level of abstraction. This abstraction is often appropriate for realistic representation of a system's behaviour. Furthermore, as in between events, the state of the system does not change, a discrete-event simulator need not explicitly represent the state of the system at non-event times. This allows for highly *efficient simulation* as compared to continuous simulation, where in principle, state information must be represented at each point in continuous time.
The high level of abstraction may however introduce simulation *artifacts* which do not pertain to real-world behaviour. In particular, event *simultaneity* whereby multiple distinct events occur at exactly the same time may be due to an insufficiently detailed discrete-event model. The DEVS formalism and its derivatives rigourously describe the semantics of such event *collisions.* A detailed presentation of the semantics of *pinnacles* and *mythical states,* which occur when events are used for respectively *time-scale abstraction* and *parameter abstraction* of continuous phenomena in hybrid models, is given by Mosterman and Biswas in [MB01].

The simple example system depicted before in Figure 1.21 will be used to illustrate relevant concepts. At the physical level, the system consists of a cashier serving arriving customers, one at a time. Customers queue if the cashier is not available (serving another customer). Here, the state of the system consists of the state of the queue and that of the cashier. The queueing discipline is First In First Out (FIFO) and individual

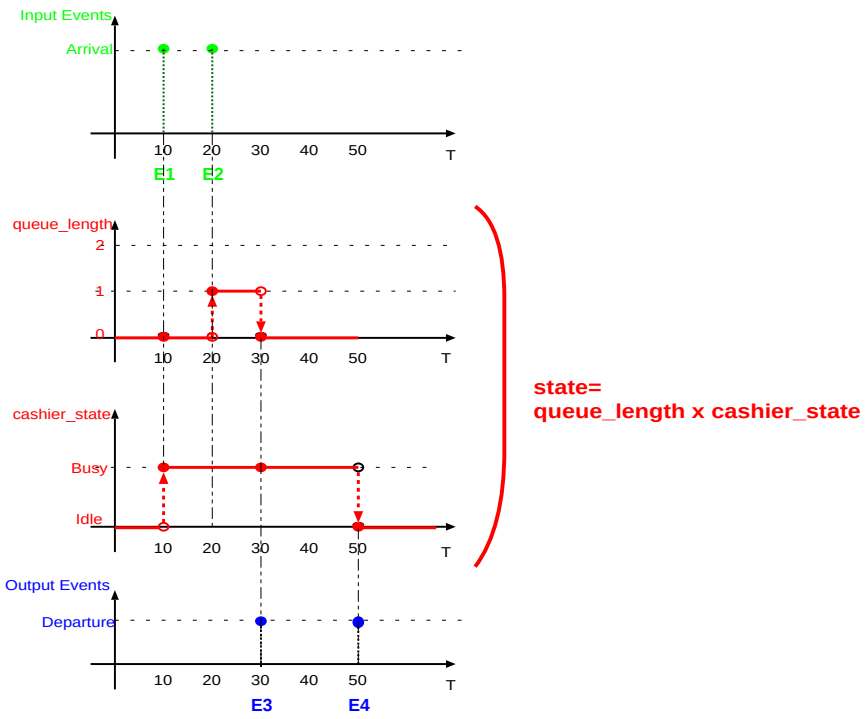

Figure 2.1: Queueing system state trajectory

customers are assumed not to have any distinguishing features (such as age, or number of items bought). Thus, it is meaningful to model the state of the queue by means of the queue length, a natural number. The cashier can be in either the Idle or the Busy state. The dynamics of the system is determined by:

- the arrival pattern of customers characterized by their Inter Arrival Time (IAT) distribution,
- the time required by the cashier to serve a customer characterized by the Service Time (ST) distribution,
- the logical sequence of customers progressing through the system under different conditions (queue empty/not empty, cashier Busy/Idle).

In Figure 2.1, an example behaviour of the cashier/queue system, its reaction to a particular input segment of customer arrivals, starting from an initial state, is depicted.

2.2.1 Definitions

The following (due to Nance [Nan81]) enable a correct understanding of different types of discrete-event simulation models.

- An *instant* is a value of system time at which the value of at least one attribute of an object can be assigned.
- An *interval* is the duration between two successive instants.
- A time *span* is the contiguous succession of one or more intervals.
- The *state* of an object at a particular instant is the enumeration of all attribute values of that object at that instant (mathematically a tuple, element of the product set of all attribute value sets). The state consists of all the object states at a particular instant.

A simulation model has a *static structure* and a *dynamic structure*. The static structure specifies the possible states of the model. The dynamic structure specifies how the state changes over time. The static structure is usually described as a collection of objects and their attributes [CS92]. There are different approaches,

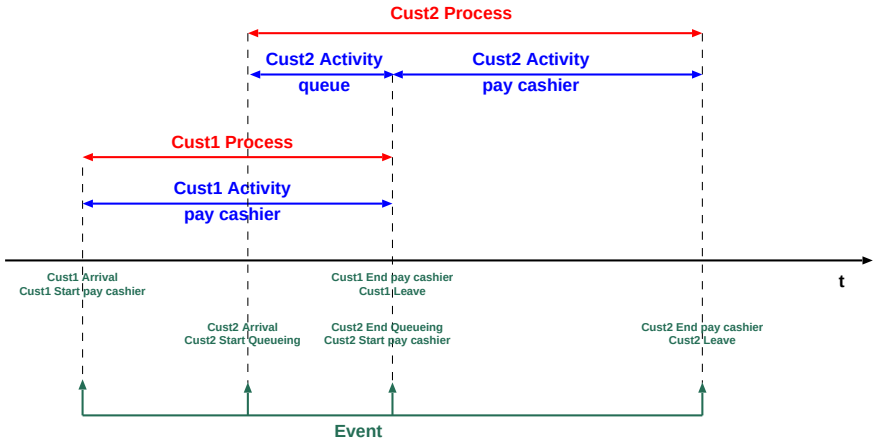

Figure 2.2: Event/Object Activity/Process

known as *world views*, to representing the dynamic structure of a model. The following concepts are at the basis of the different world views:

- An *activity* is the state of an object over an interval.
- An *event* is a change of object state, occuring at an instant, that initiates an activity precluded prior to that instant. An event is *determined* if the condition for event occurence depends exclusively on system time. In hybrid simulation modelling this is called a *time event*. Otherwise, the event is *contingent* (dependent on system conditions). In hybrid modelling, this is called a *state event*.
- An *object activity* is the state of an object between two events describing successive state changes *for that object*. Other events may occur, related to state changes of other objects.
- A *process* is the succession of states of an object over a time span. This is equivalent to the contiguous succession of one or more object activities.

Events, activities and processes for the cashier/queue example are depicted in Figure 2.2. For a given problem, the following steps are followed to determine what the events are:

1. Identify objects and their attributes.
2. Identify attributes of the system.
3. Define what causes changes in attribute value as an *event*.

Often extra state variables are added to allow calculation of *performance metrics* such as counters, minima and maxima, averages, and frequency distributions of relevant variables. In discrete event simulation, one is mostly interested in the values of performance metrics such as average queue length and utilization of resources. This is in contrast with continuous simulation, where one is mostly interested in the explicit state trajectory. The performance metrics are output at the end of a simulation run.

In the following sections, the different world views are presented by means of an *operational* definition of their simulation kernels.

2.2.2 The Event Scheduling world view

In the *Event Scheduling* world view, a model describes, for each of the events, the event's effect

- on the state,
- on the future behaviour of the system. This is achieved by *scheduling* new events into the future.

An event scheduling model for the single queue, single server example is given below:

```
declare variables:
 queue_length in PosInt
 cashier_state in {Idle, Busy}

declare events:
 start, arrival, departure, end

define events:

 start event
  /* scheduled first automatically by simulator */

  /* initializations */
  queue_length  = 0
  cashier_state  = Idle

  /* schedule end of simulation */
  schedule end absolute end_time

 arrival event
  schedule arrival relative Random(mean, spread)
  if (queue_length == 0)
   if (cashier_state == Idle)
    cashier_state = Busy
    schedule departure relative Random(mean, spread)
   else
    queue_length++
  else /* queue_length != 0 */
   queue_length++

 departure event
  if (queue_length == 0)
   cashier_state = Idle
  else /* queue_length != 0 */
   queue_length--
   schedule departure relative Random(mean, spread)

 end event
  /* dummy, terminates simulation */
```

As shown in Figure 2.3, an event scheduling simulation kernel uses two (global) data structures. One contains the state variables declared in the model. The other contains scheduled event notices in an event list, ordered by increasing time and decreasing priority. When scheduled, events are always added from the rear. Priorities are used to choose between events occurring at the same time (collisions). The state variables may be augmented by additional performance variables for calculation of minima, maxima, mean, standard deviation, *etc.* of state variables and combinations of them. An event scheduling kernel operates by ordering (according to increasing time) scheduled events in the event list and iteratively removing and processing the head of that list until the list becomes empty. The event time of the event notice is used to advance the simulation time. Depending on the event type of the event notice, the appropriate event notice is invoked. This routine may modify the system's state and schedule new events into the future by placing event notices in the event list.

In the spirit of the Event Scheduling formalism,

- initialization of the system state as well as pre-scheduling of events may be put in a "start" event. This event is automatically put in the event list and subsequently processed (first) using the same procedure as for any other event.
- halting the simulation at a certain simulation time can be achieved by scheduling a special "end" event which is recognized by the simulation procedure as the last event to be processed (even if more event notices are present on the event list). This event may contain terminal processing instructions, mainly

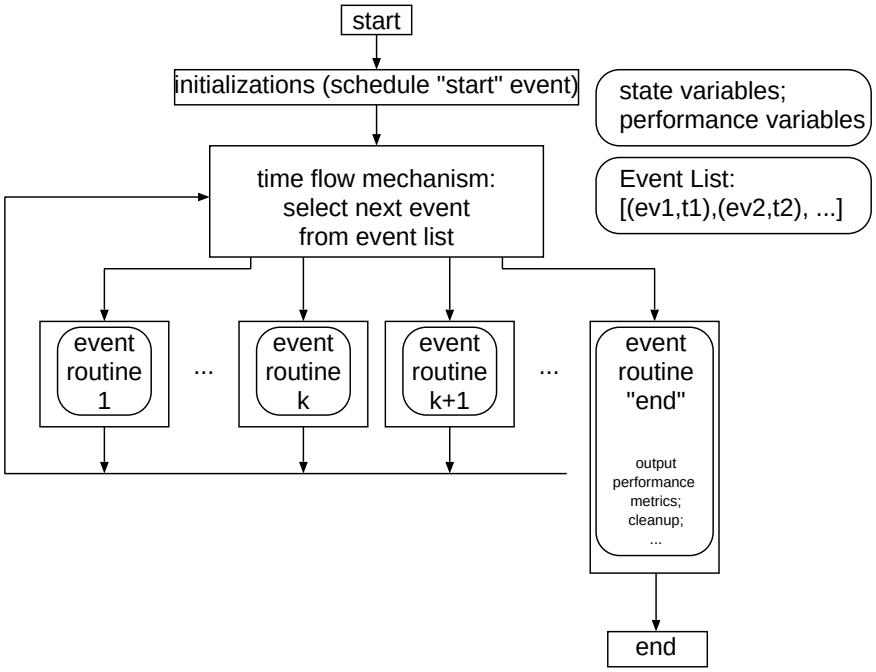

Figure 2.3: Event Scheduling simulation kernel

generating output of performance measures and other gathered statistics. Caveat: it may be necessary to re-schedule the "end" event or to give it the lowest priority to avoid missing an event occurring exactly at the time of the "end" event.

The above informally describes an event scheduling model in terms of a system state, events, an event list, and how an event influences the system state and event list (schedule new events in future). An event scheduling model is simulated by a simulation procedure which iteratively advances simulation time, updates the event list, as well as the system state.

The model representation as well as the simulation procedure are presented here in mathematical form, to facilitate the description (in the next section) of the mapping onto the DEVS formalism.

For the sake of simplicitly, we currently ignore

- output of the model, as this can easily be added and does not change the essence of the formalism;
- external events interrupting the autonomous behaviour of the system, as this is not normally part of the event scheduling formalism. This implies that hierarchies of event scheduling models can not be described. External events and hierarchy can be added easily after mapping onto DEVS.

The structure of an event scheduling model ES is

$$ES = \langle T, E, S, EL, \delta_t, \delta_\eta, \delta_S \rangle.$$

In this structure, T is the Time Base

$$T = \mathbb{R}.$$

The finite set E contains unique *event types* η such as "arrival of customer 1" and "departure of customer 5". Note that an actual *event* occurrence is characterized by a tuple (η, t) including the event type and the event instant. When present on an event list, this tuple is called an *event notice.* Events may be divided into "classes" (C) such as arrivals and departures (of different entities):

$$E = \bigcup_{i \in C} E_i.$$

It may be necessary to define an order relationship $\leq$ over E: $(E, \leq)$ to "encode" *priorities.* For example, $arrival_{customer} < arrival_{manager} < departure_{customer} < departure_{manager}$ means *arrival* events have lower priority than *departure* events and within these two event classes, managers have higher priority than customers. This is a common approach to encode event selection when it is necessary to choose between multiple events occurring simultaneously (a collision) such as $\{(\eta_1, t), (\eta_2, t), \ldots\}$ on the event list. If priorities do *not* resolve a collision to one single (η, t), the selection becomes implementation dependent. This is not portable across implementations and leads to different simulation results on different platforms. This means simulation experiments are not *repeatable.* If this situation occurs, more detail will typically be added to the priorities ordering $(<)$.

The event list EL is a possibly empty set (or even a bag if the same event is allowed to occur multiple times at the same time) of event notices

$$EL \in 2^{E \times T}.$$

For example,

$$EL = \{\underbrace{(\eta_1, t_1)}_{ev_1}, \underbrace{(\eta_2, t_2)}_{ev_2}, \cdots, \underbrace{(\eta_n, t_n)}_{ev_n}\}.$$

Note how, in an implementation-oriented description of ES, an event list would be described as an ordered list. In a more denotational fashion, not insisting on a particular implementation data structure, we use a set, with the order imposed by a $select\_first()$ operator. This leaves room for efficient, possibly parallel, implementation.

$$\begin{array}{rlll} select\_first: & 2^{E \times T} \setminus \emptyset & \rightarrow & E \times T, \\ & EL & \rightarrow & ev^* = (\eta^*, t^*). \end{array}$$

where

$$\begin{array}{rcl} t^* & = & \min_{(T,\leq)}\{t \mid (\eta, t) \in EL\}; \\ \eta^* & = & select\{(\eta \mid (\eta, t^*) \in EL\}. \end{array}$$

In the above, it is assumed $EL \neq \emptyset$. Simulation halts when the event list becomes empty and thus $select\_first$ will never be applied to an empty event list (as specified in the simulation procedure Algorithm 1).

The *select* tie-breaking function is needed to select between simultaneously occuring events. As mentioned before, *select* is typically implemented based on an ordering relationship $\leq$ over E:

$$\begin{array}{rlll} select: & 2^E & \rightarrow & E, \\ & \{\eta_1, \cdots, \eta_n\} & \rightarrow & \min_{(E,\leq)}\{\eta_1, \cdots, \eta_n\}. \end{array}$$

Applying *select* should yield a unique result. This will only be the case if there is a *strict* ordering (no equalities) over the set of events E.

The state set S is modified at event times by *event handlers*. S may obviously be a product set: $S = \times_i S_i$.

For each $\eta \in E$, an event handler is a structure

$$(\delta_t, \delta_S, \delta_\eta).$$

This allows one to specify the effects of handling an event:

1. modification of the system state $\in S$,
2. scheduling a new event η in the future.

In the Event Scheduling formalism as described here, there is only a single modification of the system state, as well as only one event scheduled. This can be done without loss of generality as multiple state changes and future events scheduled are all done at the same instant of time. Multiple state changes and events scheduled can be emulated (modelled) by a sequence of events, each only performing one state change and one scheduling. Note how this is only true if no output is generated for intermediate state changes.

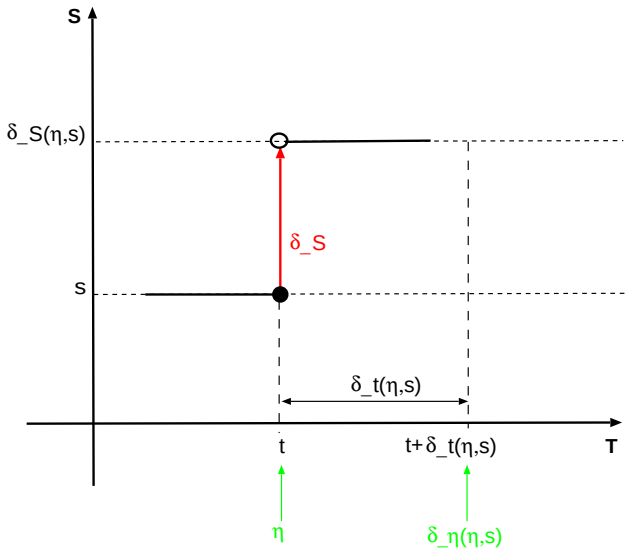

Figure 2.4: Event handler $= (\delta_t, \delta_S, \delta_\eta)$

Alternately, the formalism could easily be adopted to lump a sequence of state changes into one resultant state change, and to schedule a series of events in the future.

As shown in Figure 2.4, the event handler structure specifies:

1. At what time in the future to schedule a new event. The time delay δ_t between the current time and the time the new event is scheduled at is based on the current event and system state:

$$\delta_t : E \times S \to \mathbb{R}^+_{0,+\infty}.$$

2. How to modify the system state based on the current event and system state:

$$\delta_S : E \times S \to S.$$

3. The event type to be scheduled. This is given by δ_η based on the current event and system state:

$$\delta_\eta : E \times S \to E.$$

The above concludes the *structure* of the model formalism. Algorithm 1 describes a *simulation procedure* for simulating an event scheduling model ES. To carry out the simulation, the model (event types, state set, and event handlers) is given together with intial conditions:

- the initial event list (pre-scheduled events),
- the initial state $s \in S$.

Note how lines 4 and 5 in Algorithm 1 are a generalization of what is common in list-based event scheduling implementations:

$$(\eta_{first}, t_{first}) \leftarrow \text{head}(EL),$$

where "head" is the standard ordered list operator which selects the first element of the list.

Similarly, the approach on line 10 is a generalization of updating of the event list in list-based event scheduling implementations:

$$EL \leftarrow \text{insert}_{((T,\leq),(E,\leq))}((\eta', t + \Delta t), \text{tail}(EL)),$$

with "insert" and "tail", the standard ordered list operators. The position where an event notice is "insert"ed is determined by the its timestamp as well as its priority. If both are equal, notices are added from the rear. "tail" produces the remainder of EL after removing its "head".

Algorithm 1 Event Scheduling simulation procedure

1: $s \leftarrow$ initial state $\in S$ {initialize the state}
2: $EL \leftarrow$ initial event list {initialize the event list (pre-scheduled events)}
3: **while** $(EL \neq \emptyset)$) **do**
4: $t_{first} \leftarrow \min_{(T,\leq)}\{t \mid (\eta, t) \in EL\}$
5: $\eta_{first} \leftarrow select(\{\eta \mid (\eta, t_{first}) \in EL)$
6: $t \leftarrow t_{first}$ {advance current time to t_{first}}
7: $\eta \leftarrow \eta_{first}$ {event type currently processed}
8: $\Delta t \leftarrow \delta_t(\eta, s)$
9: $\eta^*_{new} \leftarrow \delta_\eta(\eta, s)$
10: $EL \leftarrow (EL \setminus (\eta, t)) \cup \{(\eta', t + \Delta t)\}$ {$\setminus$: remove the current event, $\cup$: add a scheduled event}
11: $s \leftarrow \delta_S(\eta, s)$ {update state}
12: **end while**

2.2.3 The Activity Scanning world view

In the *Activity Scanning* world view, a model describes *conditions* which will activate *activities.* This representation (and its semantics described below) resembles that used in declarative AI languages such as Prolog [CM87, Van88].

An activity scanning model for the single queue, single server example is given below.

```
declare (and initialize) variables:
 queue_length in PosInt = 0
 cashier_state in {Idle, Busy} = Idle
 t_arrival = 0, t_depart  = plusInf

define conditions:

arrival condition: t >= t_arrival
 if (queue_length == 0)
  if (cashier_state == Idle)
   keep queue_length == 0
   cashier_state = Busy
   t_depart = t + Random(mean, spread) /* service time */
  else
   queue_length++
 else /* queue_length != 0 */
  queue_length++, keep cashier_state == Busy
 t_arrival = t + Random(mean, spread)   /* inter arrival time */

departure condition: t >= t_departure
 if (queue_length == 0)
  cashier_state = Idle
 else /* queue_length != 0 */
  queue_length--, keep cashier_state == Busy
  t_depart = t + Random(mean, spread) /* service time */
```

As shown in Figure 2.5, an activity scanning simulation kernel uses a discrete time step to advance time. During the activity scan phase, the solver checks for an activity whose condition (a boolean function of the time variable and the state variables) is true and processes it. This scan is continued as long as some activity condition evaluates to true. If none of the activities is enabled, the time flow phase is executed again, advancing time. In the spirit of the Activity Scanning formalism, a “start” and “end” activity may be defined with semantics similar to their Event Scheduling counterparts.

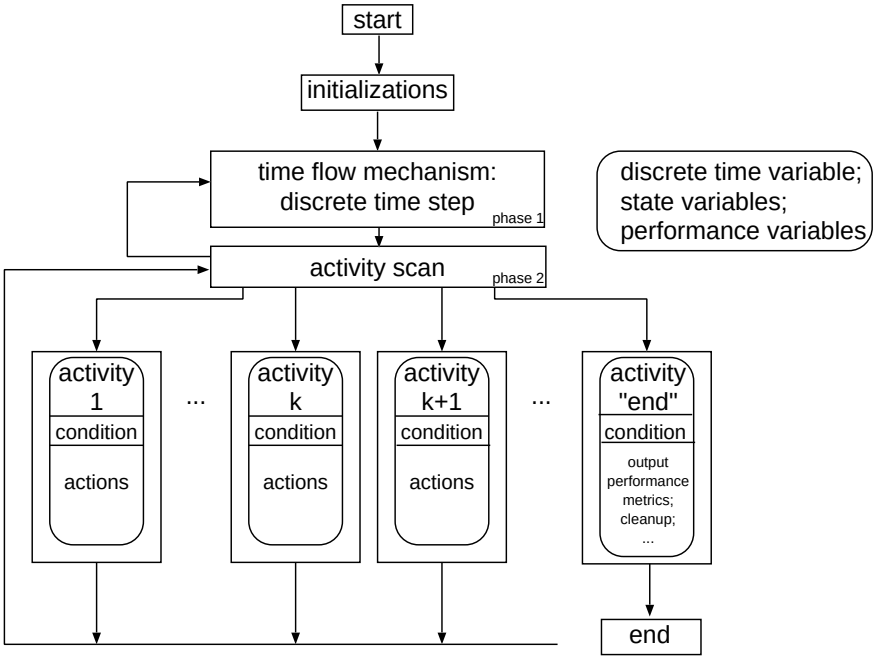

Figure 2.5: Activity Scanning simulation kernel

2.2.4 The Three Phase Approach world view

As Activity Scanning uses a fixed time step, it is not efficient. On the one hand, the time step needs to be chosen as small as the smallest time interval possible between two events to correctly model behaviour. On the other hand, some events may be extremely far apart in time (many times the smallest time between events). For such long time intervals, an activity scanning simulator will unnecessarily check all conditions at each point in time despite the fact that the conditions do not change.

In the *Three Phase Approach* world view, Activity Scanning is combined with Event Scheduling. Activities may be scheduled explicitly into the future as in the Event Scheduling world view. In addition, at event times, all activity conditions are checked as in the Activity Scanning world view. Two types of activities are represented:

- "bound to occur activities" (B): are scheduled in an Event Scheduling fashion and describe the effect of unconditional state changes on the current state and on the future (by scheduling new B activities into the future).
- "conditional activities" (C): are invoked at event times if their condition evaluates to true. Describe the effect of unconditional state changes on the current state and on the future (by scheduling new B activities into the future).

As shown in Figure 2.6, a three phase approach simulation kernel combines the scheduling of B activities of the Event Scheduling world view (with its associated time flow mechanism of advancing time to the time of the first event on the Event List) with the invocation of conditionional C activities of the Activity Scanning world view. Again "start" and "end" activites may be defined. It will be noticed that these and all other activities may be described as B *or* as C activities. This shows the conceptual flaw in the Three Phase Approach: mixing different world views in a single model makes the model hard to understand and maintain.

2.2.5 The Process Interaction world view

At the highest level of abstraction, in the *Process Interaction* world view, a template is given for the life of *transactions* or *processes* as they progress through a number of *activities* or *blocks*. In Figure 2.7, a process

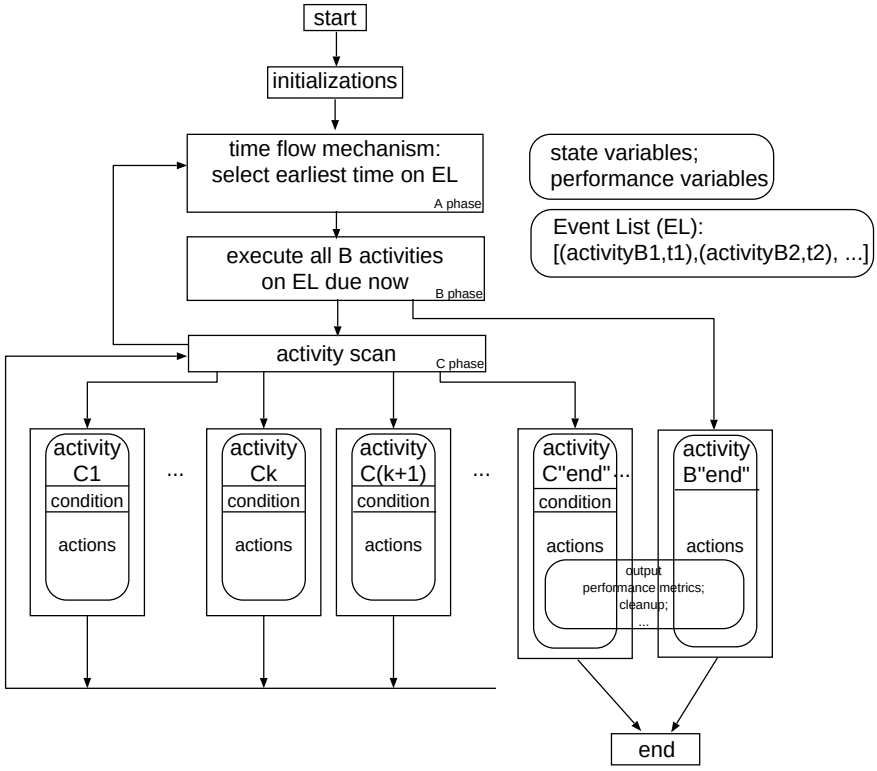

Figure 2.6: Three Phase Approach simulation kernel

interaction model for the single queue, single server example is given in the General Purpose Simulation System (GPSS) language and its corresponding graphical notation [Gor96, Sch74, BCIN98, LK91]. The arrival of transactions (customers) is modelled in the `GENERATE` block. The inter-arrival time of customers is uniformly distributed over the interval 10 +/- 5 time units. The `QUEUE/DEPART` block combination collects queueing statistics of the queue formed by customers waiting for the capacity 1 resource “cashier” modelled by the `SEIZE/RELEASE` block combination. Once the cashier facility is seized, a customer is served for a time sampled from a uniform distribution over the interval 5 +/- 3 time units. This is modelled by an `ADVANCE` block. At the `TERMINATE` block, the life of transaction ends.

As shown in Figure 2.8, a process interaction simulation kernel employs three main data structures: the Future Event List (FEL) and the Current Event List (CEL) (the “chains” FEC and CEC in GPSS terminology) are internal to the simulator whereas the third one represents the Process Interaction model. A transaction is

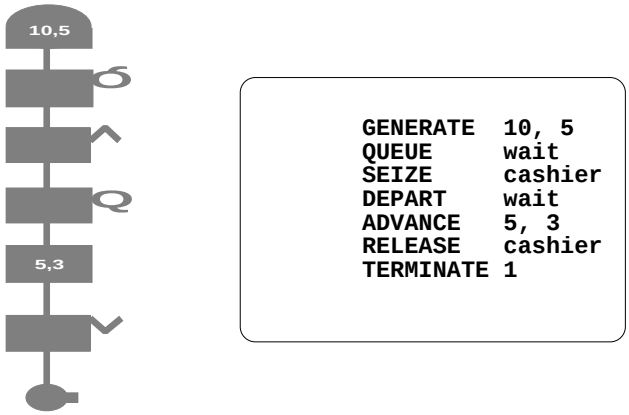

Figure 2.7: Process Interaction (GPSS) model of a cashier/queue system

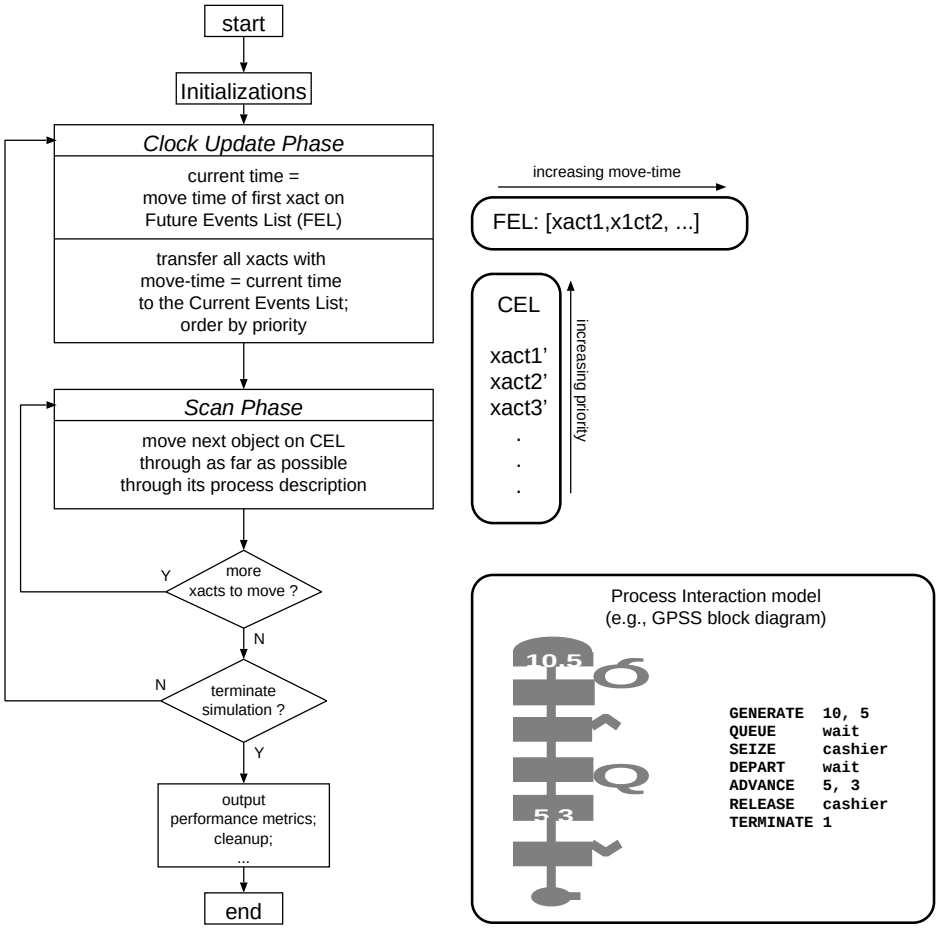

Figure 2.8: Process Interaction simulation kernel

always present in exactly one of the two lists. A transaction data structure contains

- a unique identifier,
- a priority,
- a move-time, the time at which the transaction is scheduled (by an `ADVANCE` block for example) to the next block in the model.
- a number of transaction attributes described by the modeller ("parameters" in GPSS terminology).

The FEL is a list of transactions ordered by increasing move-time (starting from the head). For equal move-times, the order is not specified. The CEL is a list of transactions ordered by decreasing priority (starting from the head). After an initialization phase, the simulation proceeds iteratively through two phases:

1. During the *clock update phase*, the current simulation time is advanced to the move time of the first transaction on the FEL. Subsequently, all transactions on the FEL with move-time equal to the current time are moved to the CEL. On the CEL, transactions are ordered by priority.

2. During the *scan phase*, the CEL is searched from beginning (high priority transactions) till end (low priority transactions) for transactions which can be moved through the model. If a transaction is found, it is moved *as far as possible* through the model. A transaction may become *blocked* when it enters an `ADVANCE` for example. Then, the simulator schedules it to leave the block at a later time by putting it on the FEL. When a transaction reaches a `TERMINATE` block, it is destroyed (removed from the CEL). The scan phase is repeated as long as it is possible to move transactions through the model. When no more transactions can be moved, and the simulation's termination condition does not yet evaluate to

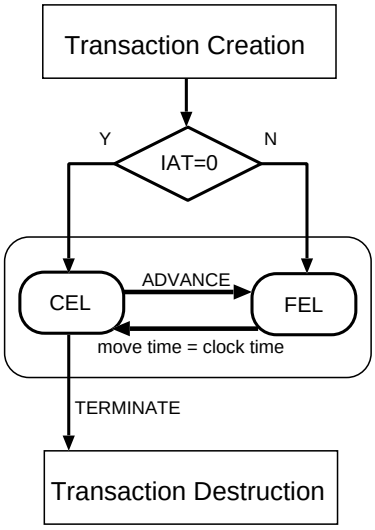

Figure 2.9: A transaction's life during a Process Interaction simulation

true, the clock update phase is invoked again. The termination condition is often encoded as a global Termination Counter being decremented to zero or below. It is obvious that blocked transactions may reside on the CEL well beyond their move-time, waiting for some system condition.

Before completely terminating a simulation, performance metrics are output and data structures are cleaned up.

Figure 2.9 depicts the life of a single transaction from its creation, over its repeated migration to and from the Current Event List and the Future Event List, until its final destruction in a `TERMINATE` block.

2.2.6 Relationships between discrete event world views

Figure 2.10 gives an overview of the relationships between "different" discrete formalisms. On the left hand side, formalisms are shown whose time flow mechanism is a fixed time advance. On the right hand side, formalisms with a "discrete event" time flow (clock advances to event times only) are shown. It is noted that the Activity Scanning world view really belongs under Discrete Time formalisms though it is always erroneously included with Discrete Event formalisms.

All discrete event formalisms presented are *non-modular* (see Chapter 1). Model components such as event handlers (Event Scheduling world view), activities (Activity Scanning world view) and process blocks (Process Interaction world view) are not encapsulated entities, only interacting with their environment through interfaces. Rather, they directly influence global state variables as well as other components.

The dashed arrow lines in Figure 2.10 denote transformation of a model described in a source formalism (start of arrow line) into that same model described in the target formalism (end of the arrow line). Original and transformed model are considered "equivalent" when they produce the same state trajectory when simulated from identical intial conditions. In the figure, all transformations are towards the DEVS formalism which is described in the next section. There, it is also shown how an Event Scheduling model may be transformed into an "equivalent" DEVS model. None of the other transformations in Figure 2.10 is described. A rigourous treatment of these transformations (including equivalence proofs) is future work. The essence of all these transformations is the construction of a *modular* Coupled DEVS model from the non-modular specifications. This is achieved by explicitly representing *dependencies* by means of couplings between modular components. To automate such transformations, dependency analysis needs to be automated. This approach is deemed feasible thanks to the experience with dependency analysis for continuous models (Differential Algebraic Equations) as described in a later section.

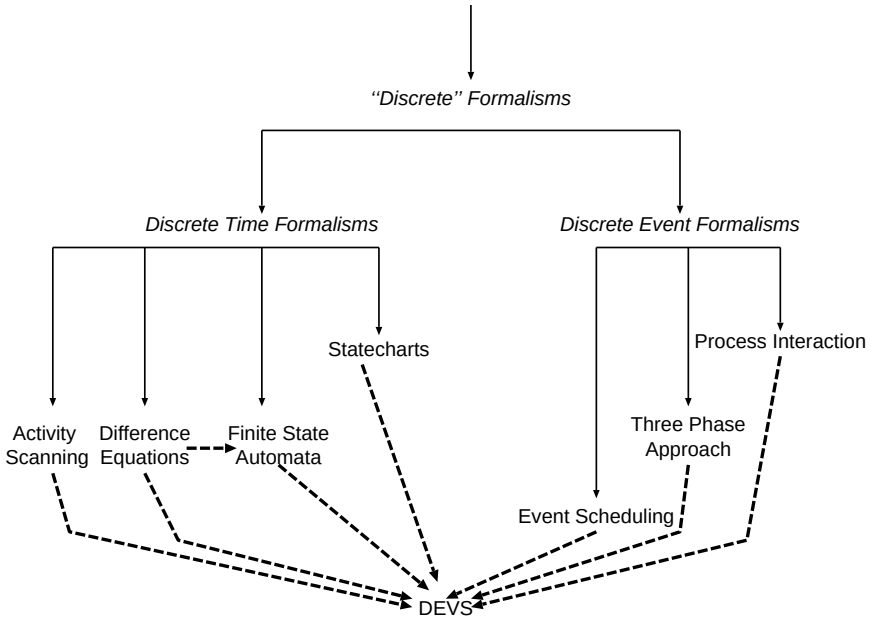

Figure 2.10: World Views classification

2.3 The DEVS formalism

The DEVS formalism was conceived by Zeigler [Zei84a, Zei84b] to provide a rigourous common basis for discrete-event modelling and simulation. For the class of formalisms denoted *discrete-event* [Nan81], system models are described at an abstraction level where the time base is continuous ($\mathbb{R}$), but during a bounded time-span, only a *finite number* of relevant *events* occurs. These events can cause the state of the system to change. In between events, the state of the system does *not* change. This is unlike *continuous* models in which the state of the system may change continuously over time.

As an extension of Finite State Automata, the DEVS (Discrete Event Systems) formalism captures *concepts* from discrete event simulation. As such it is a sound basis for meaningful model exchange in the discrete event realm. In Figure 2.11, the state trajectory of a traffic light is shown. Finite State Automata can be extended to include the time the system stays in a particular state before making a transition to the next state. This is the approach taken in DEVS. It is however always possible to construct an Event Graph which has as nodes, the transitions, and as edges, the time interval after which the next transition is scheduled to occur. This demonstrates the link with the event scheduling discrete event world view. In a later section, a more formal presentation of the relationship between the discrete event world views and DEVS will be presented.

The DEVS formalism fits the general I/O Systems structure of deterministic, causal systems in classical systems theory described in the first chapter. DEVS allows for the description of system behaviour at two levels. At the lowest level, an *atomic DEVS* describes the autonomous behaviour of a discrete-event system as a sequence of deterministic transitions between sequential states as well as how it reacts to external input (events) and how it generates output (events). At the higher level, a *coupled DEVS* describes a system as a *network* of coupled components. The components can be atomic DEVS models or coupled DEVS in their own right. The connections denote how components influence each other. In particular, output events of one component can become, via a network connection, input events of another component. It is shown in [Zei84a] how the DEVS formalism is *closed under coupling*: for each coupled DEVS, a *resultant* atomic DEVS can be constructed. As such, any DEVS model, be it atomic or coupled, can be replaced by an atomic DEVS. The construction procedure of a resultant atomic DEVS is also the basis for the implementation of an *abstract simulator* or solver capable of simulating any DEVS model. As a coupled DEVS may have coupled DEVS components, *hierarchical* modelling is supported.

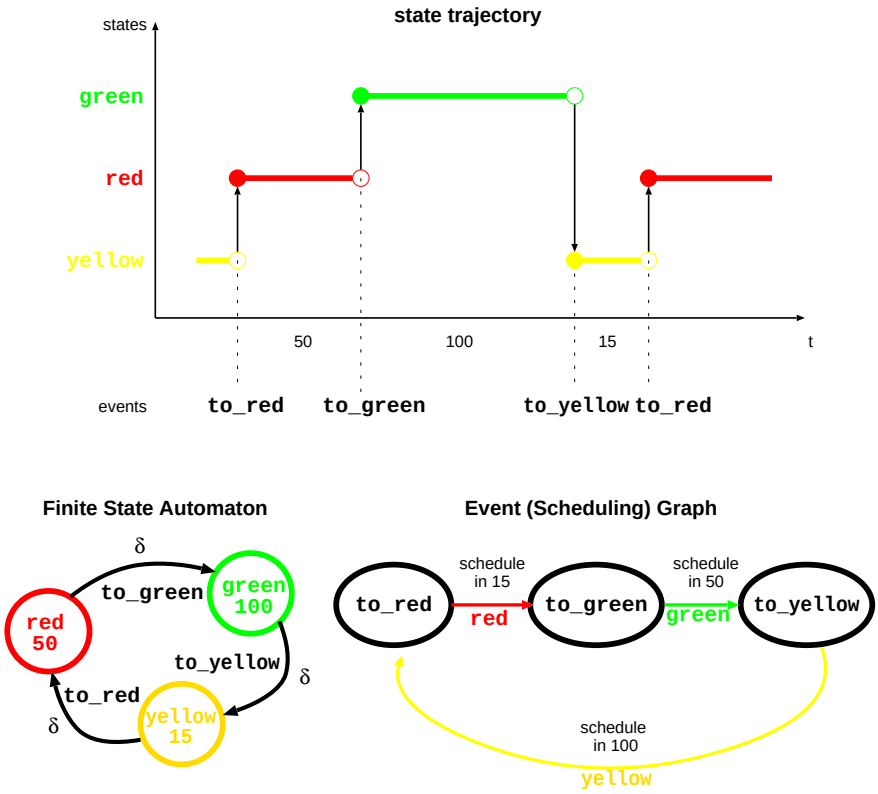

Figure 2.11: Finite State Automaton and Event Graph models

In the following, the different aspects of the DEVS formalism are explained in more detail.

2.3.1 The atomic DEVS formalism

The atomic DEVS formalism is a structure describing the different aspects of the discrete-event behaviour of a system:

$$atomicDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle.$$

The *time base* T is continuous and is not mentioned explicitly:

$$T = \mathbb{R}.$$

The *state set* S is the set of admissible *sequential states*: the DEVS dynamics consists of an ordered sequence of states from S. Typically, S will be a *structured* set (a product set)

$$S = \times_{i=1}^{n} S_i.$$

This formalizes multiple (n) *concurrent* parts of a system. It is noted how a structured state set is often synthesized from the state sets of concurrent components in a coupled DEVS model.

The time the system *remains* in a sequential state before making a transition to the next sequential state is modelled by the *time advance function*

$$ta : S \rightarrow \mathbb{R}_{0,+\infty}^{+}.$$

As time in the real world always advances, the image of ta must be non-negative numbers. $ta = 0$ allows for the representation of *instantaneous* transitions: no time elapses before transition to a new state. Obviously, this is an abstraction of reality which may lead to simulation *artifacts* such as infinite instantaneous loops which do not correspond to real physical behaviour. If the system is to stay in an end-state s *forever*, this is modelled by means of $ta(s) = +\infty$.

The internal transition function

$$\delta_{int} : S \to S$$

models the transition from one state to the next sequential state. δ_{int} describes the behaviour of a Finite State Automaton; ta adds the progression of time.

It is possible to *observe* the system output. The output set Y denotes the set of admissible *outputs*. Typically, Y will be a *structured* set (a product set)

$$Y = \times_{i=1}^{l} Y_i.$$

This formalizes multiple (l) output ports. Each port is identified by its unique index i. In a user-oriented modelling language, the indices would be derived from unique port *names*.

The output function

$$\lambda : S \to Y \cup \{\phi\}$$

maps the internal state onto the output set. Output events are *only* generated by a DEVS model at the time of an *internal* transition. At that time, the state *before* the transition is used as input to λ. At all other times, the non-event ϕ is output.

To describe the *total state* of the system at each point in time, the sequential state $s \in S$ is not sufficient. The *elapsed* time e since the system made a transition to the current state s needs also to be taken into account to construct the total state set

$$Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$$

The elapsed time e takes on values ranging from 0 (transition just made) to $ta(s)$ (about to make transition to the next sequential state). Often, the *time left* σ in a state is used:

$$\sigma = ta(s) - e.$$

Up to now, only an *autonomous* system has been described: the system receives no external inputs. Hence, the *input set* X denoting all admissible input values is defined. Typically, X will be a *structured* set (a product set)

$$X = \times_{i=1}^{m} X_i$$

This formalizes multiple (m) input ports. Each port is identified by its unique index i. As with the output set, port indices may denote *names*.

The set Ω contains all admissible input segments ω

$$\omega : T \to X \cup \{\phi\}.$$

In discrete-event system models, an input segment generates an input *event* different from the *non-event* ϕ only at a finite number of instants in a bounded time-interval. These *external events*, inputs x from X cause the system to interrupt its autonomous behaviour and react in a way prescribed by the external transition function

$$\delta_{ext} : Q \times X \to S.$$

The reaction of the system to an external event depends on the sequential state the system is in, the particular input *and* the elapsed time. Thus, δ_{ext} allows for the description of a large class of behaviours typically found in discrete-event models (including synchronization, pre-emption, suspension and re-activation).

When an input event x to an atomic model is not listed in the δ_{ext} specification, the event is *ignored*.

In Figure 2.12, an example state trajectory is given for an atomic DEVS model. In the figure, the system made an internal transition to state $s2$. In the absence of external input events, the system stays in state $s2$ for a duration $ta(s2)$. During this period, the elapsed time e increases from 0 to $ta(s2)$, with the total state $= (s2, e)$. When the elapsed time reaches $ta(s2)$, first an output is generated: $y2 = \lambda(s2)$, then the system transits instantaneously to the new state $s4 = \delta_{int}(s2)$. In autonomous mode, the system would stay in state $s4$ for $ta(s4)$ and then transit (after generating output) to $s1 = \delta_{int}(s4)$. Before e reaches $ta(s4)$ however, an

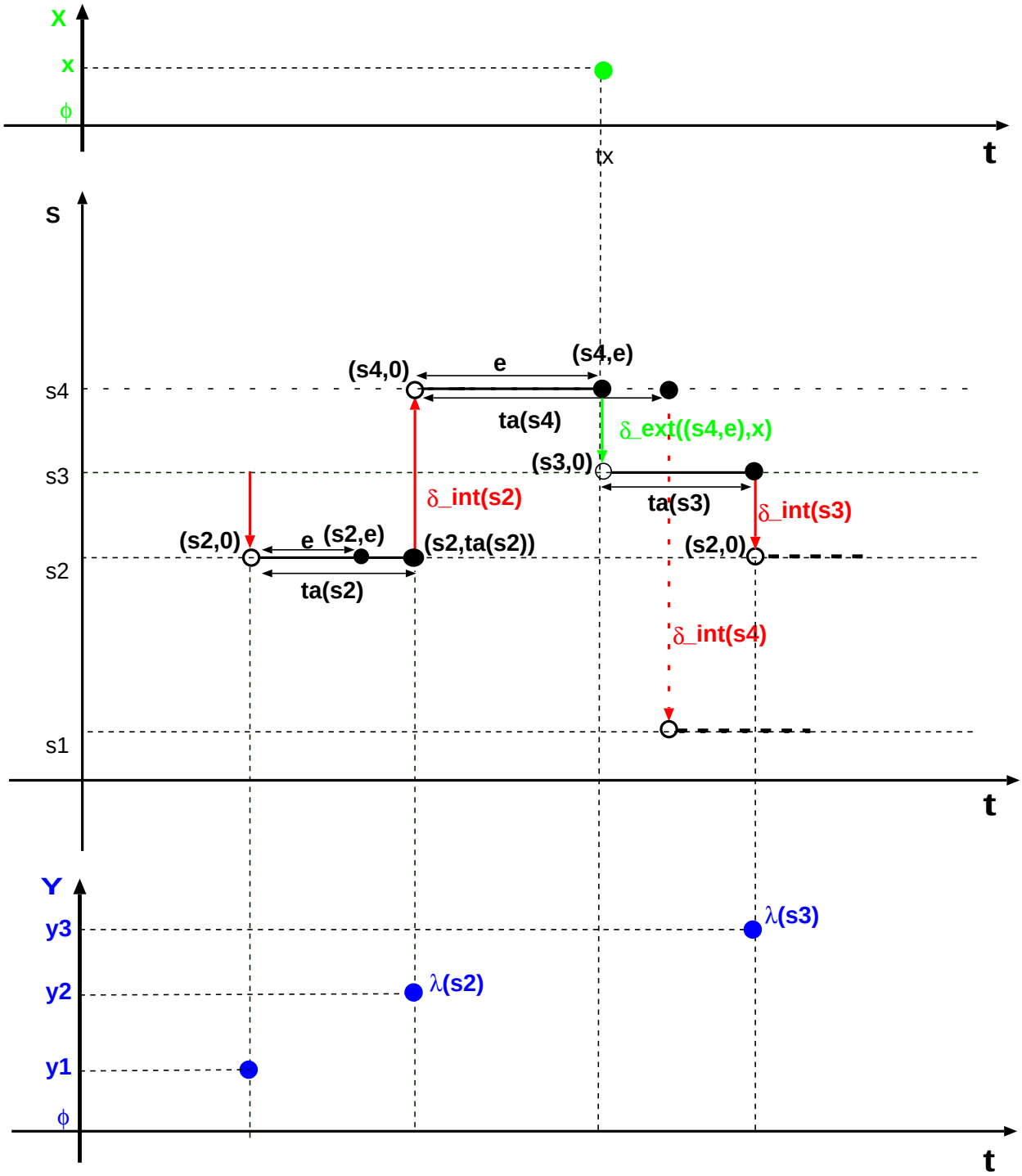

Figure 2.12: State Trajectory of a DEVS-specified Model

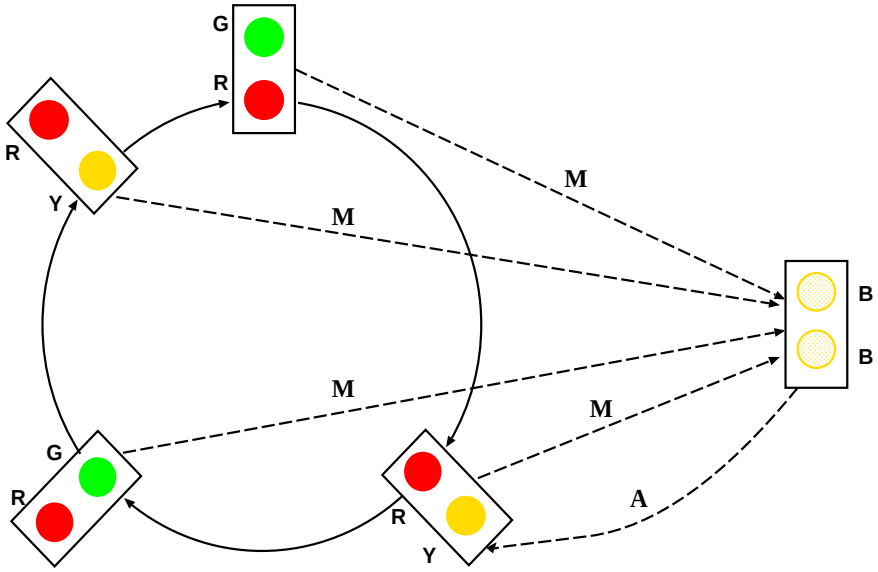

Figure 2.13: Traffic light example

external input event x arrives. At that time, the system forgets about the scheduled internal transition and transits to $s3 = \delta_{ext}((s4, e), x)$. Note how an external transition does not give rise to an output. Once in state $s3$, the system continues in autonomous mode.

As an example atomic DEVS, consider the model of two traffic lights depicted in Figure 2.13. In autonomous mode, the light transits in intuitive fashion. If the "switch to manual" (M) external event is received, lights in both directions blink yellow. If the "switch to automatic" (A) event is received, the system switches back deterministically to state RY to resume autonomous mode. Colour-blind observation of the system is encoded in λ. The atomic DEVS representation is given below.

$$DEVS = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

$$
\begin{aligned}
&T = \mathbb{R} \\
&X = \{M, A\} \\
&\omega : T \to X \cup \{\phi\} \\
&S = \{RG, RY, GR, YR, BB\} \\
&\delta_{int}(RG) = RY;\ \delta_{int}(RY) = GR \\
&\delta_{int}(GR) = YR;\ \delta_{int}(YR) = RG \\
&ta(RG) = 60s;\ ta(RY) = 10s \\
&ta(GR) = 50s;\ ta(YR) = 10s \\
&ta(BB) = +\infty \\
&\delta_{ext}((RG, e), M) = BB \\
&\delta_{ext}((RY, e), M) = BB \\
&\delta_{ext}((GR, e), M) = BB \\
&\delta_{ext}((YR, e), M) = BB \\
&\delta_{ext}((BB, e), A) = RY \\
&Y = \{GREY, YELLOW, BLINK\} \times \{GREY, YELLOW, BLINK\} \\
&\lambda(RG) = \lambda(GR) = (GREY, GREY) \\
&\lambda(YR) = \lambda(YG) = (YELLOW, GREY) \\
&\lambda(GY) = \lambda(RY) = (GREY, YELLOW) \\
&\lambda(BB) = (BLINK, BLINK)
\end{aligned}
$$

2.3.2 The coupled DEVS formalism

The coupled DEVS formalism describes a discrete-event system in terms of a network of coupled components.

$$coupledDEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

The component $self$ denotes the coupled model itself. X_{self} is the (possibly structured) set of allowed external inputs to the coupled model. Y_{self} is the (possibly structured) set of allowed (external) outputs of the coupled model. D is a set of unique component references (names). The coupled model itself is referred to by means of $self$, a unique reference not in D.

The set of *components* is

$$\{M_i | i \in D\}.$$

Each of the components must be an atomic DEVS

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i \rangle, \forall i \in D.$$

The set of *influencees* of a component, the components influenced by $i \in D \cup \{self\}$, is I_i. The set of all influencees describes the coupling network structure

$$\{I_i | i \in D \cup \{self\}\}.$$

For modularity reasons, a component (including $self$) may not influence components outside its scope –the coupled model–, rather only other components of the coupled model, or the coupled model $self$:

$$\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}.$$

This is further restricted by the requirement that none of the components (including $self$) may influence itself directly as this could cause an instantaneous dependency cycle (in case of a 0 time advance inside such a component) akin to an algebraic loop in continuous models:

$$\forall i \in D \cup \{self\} : i \notin I_i.$$

Note how one can always encode a self-loop ($i \in I_i$) in the internal transition function.

To translate an output event of one component (such as a departure of a customer) to a corresponding input event (such as the arrival of a customer) in influencees of that component, *output-to-input translation functions* $Z_{i,j}$ are defined:

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\},$$

$$\begin{array}{rcll} Z_{self,j} & : & X_{self} \to X_j, & \forall j \in D, \\ Z_{i,self} & : & Y_i \to Y_{self}, & \forall i \in D, \\ Z_{i,j} & : & Y_i \to X_j, & \forall i, j \in D. \end{array}$$

Together, I_i and $Z_{i,j}$ completely specify the coupling (structure and behaviour).

As a result of coupling of concurrent components, multiple state transitions may occur at the same simulation time. This is an artifact of the discrete-event abstraction and may lead to behaviour not related to real-life phenomena. A logic-based foundation to study the *semantics* of these artifacts was introduced by Radiya and Sargent [RS94]. In sequential simulation systems, such transition *collisions* are resolved by means of some form of *selection* of which of the components' transitions should be handled first. This corresponds to the introduction of priorities in some simulation languages. The coupled DEVS formalism explicitly represents a *select* function for *tie-breaking* between simultaneous events:

$$select : 2^D \to D.$$

select chooses a unique component from any non-empty subset E of D:

$$select(E) \in E.$$

The subset E corresponds to the set of all components having a state transition simultaneously.

2.3.3 Closure of DEVS under coupling

As mentioned at the start of this section, it is possible to construct a *resultant* atomic DEVS model for each coupled DEVS. This *closure under coupling* of atomic DEVS models replaces *any* coupled DEVS by an atomic DEVS. The replacement or *flattening* procedure actually *defines* the semantics of a coupled DEVS. The procedure is defined such that it is "natural": its semantics is compatible with that of the discrete event world views presented before. In particular, of all possible events, the earliest one is always processed and collisions due to simultaneous events (internal and external transitions in DEVS) are resolved by some tie-breaking procedure. By induction, any *hierarchically* coupled DEVS can thus be flattened to an atomic DEVS. As a result, the requirement that each of the components of a coupled DEVS be an atomic DEVS can be relaxed to be atomic *or* coupled as the latter can always be replaced by an atomic DEVS.

The core of the closure procedure is the selection of the most *imminent* (*i.e.,* soonest to occur) event from all the components' scheduled events [Zei84a]. In case of simultaneous events, the *select* function is used. The resultant construction is described below.

From the coupled DEVS

$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle,$$

with all components M_i atomic DEVS models

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i \rangle, \forall i \in D$$

the atomic DEVS

$$\langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle$$

is constructed.

The resultant set of sequential states is the product of the total state sets of all the components

$$S = \times_{i \in D} Q_i,$$

where

$$Q_i = \{(s_i, e_i) | s \in S_i, 0 \leq e_i \leq ta_i(s_i)\}, \forall i \in D.$$

The time advance function ta

$$ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}$$

is constructed by selecting the *most imminent* event time, of all the components. This means finding the smallest time *remaining* until internal transition, of all the components

$$ta(s) = min\{\sigma_i = ta_i(s_i) - e_i | i \in D\}.$$

A number of *imminent* components may be scheduled for a *simultaneous* internal transition. These components are collected in a set

$$IMM(s) = \{i \in D | \sigma_i = ta(s)\}.$$

From IMM, a set of elements of D, *one* component i^* is chosen by means of the *select tie-breaking* function of the coupled model

$$\begin{array}{rcccc} select & : & 2^D & \rightarrow & D \\ & & IMM(s) & \rightarrow & i^* \end{array}$$

Output of the selected component is generated *before* it makes its internal transition. Note also how, as in a Moore machine, input does not directly influence output. In DEVS models, *only* an internal transition produces output. An input can only influence/generate output via an internal transition similar to the presence of *memory* in the form of integrating elements in continuous models. Allowing an external transition to produce output could lead to infinite instantaneous loops. This is equivalent to algebraic loops in continuous

systems. The output of the component is translated into coupled model output by means of the coupling information

$$\lambda(s) = \begin{cases} Z_{i^*,self}(\lambda_{i^*}(s_{i^*})), & \text{if } self \in I_{i^*}, \\ \phi, & \text{if } self \notin I_{i^*}. \end{cases}$$

If the output of i^* is not connected to the output of the coupled model, the non-event ϕ can be generated as output of the coupled model. As ϕ literally stands for no event, the output can also be ignored without changing the meaning (but increasing performance of simulator implementations).

The internal transition function transforms the different parts of the total state as follows:

$$\delta_{int}(s) = (\ldots,(s'_j,e'_j),\ldots), \text{ where}$$

$$(s'_j,e'_j) = \begin{cases} (\delta_{int,j}(s_j),0), & \text{for } j = i^*, \\ (\delta_{ext,j}(s_j,e_j+ta(s),Z_{i^*,j}(\lambda_{i^*}(s_{i^*}))),0), & \text{for } j \in I_{i^*} \text{ and } Z_{i^*,j}(\lambda_{i^*}(s_{i^*})) \neq \phi, \\ (s_j,e_j+ta(s)), & \text{otherwise.} \end{cases}$$

The selected imminent component i^* makes an internal transition to sequential state $\delta_{int,i^*}(s_{i^*})$. Its elapsed time is reset to 0. All the influencees of i^* change their state due to an external transition prompted by an input which is the output-to-input translated output of i^*, with an elapsed time adjusted for the time advance $ta(s)$. The influencees' elapsed time is reset to 0. Note how i^* is not allowed to be an influencee of i^* in DEVS. The state of all other components is not affected and their elapsed time is merely adjusted for the time advance $ta(s)$.

The external transition function transforms the different parts of the total state as follows:

$$\delta_{ext}((s,e),x) = (\ldots,(s'_i,e'_i),\ldots), \text{ where}$$

$$(s'_i,e'_i) = \begin{cases} (\delta_{ext,i}((s_i,e_j+e),Z_{self,i}(x)),0) & \text{, for } i \in I_{self}, \\ (s_i,e_j+e) & \text{, otherwise.} \end{cases}$$

An incoming external event is routed, with an adjustment for elapsed time, to each of the components connected to the coupled model input (after the appropriate input-to-input translation). For all those components, the elapsed time is reset to 0. All other components are not affected and only the elapsed time is adjusted.

Some limitations of DEVS are that

- a conflict due to simultaneous internal and external events is resolved by ignoring the internal event. It should be possible to explicitly specify behaviour in case of conflicts;
- there is limited potential for parallel implementation;
- the *select* function is an artificial legacy of the semantics of traditional sequential simulators based on an event list;
- it is not possible to describe variable structure.

Some of these are compensated for in parallel DEVS (see further).

2.3.4 Implementation of a DEVS solver

The algorithm in Figure 2.14 is based on the closure under coupling construction and can be used as a specification of a –possibly parallel– implementation of a DEVS solver or "abstract simulator" [Zei84a, KSKP96]. In an atomic DEVS solver, the last event time t_L as well as the local state s are represented. In a coordinator, only the last event time t_L is represented. The next-event-time t_N is sent as output of either solver. It is possible to also keep t_N in the solvers. This requires consistent (recursive) initialization of the t_Ns. If kept, the t_N allows one to check whether the solvers are appropriately synchronized. The operation of an abstract simulator involves handling four types of messages. The $(x, from, t)$ message carries external input information. The $(y, from, t)$ message carries external output information. The $(*, from, t)$ and $(done, from, t_N)$ messages are used for scheduling (synchronizing) the abstract simulators. In these

message m	**simulator**	**coordinator**
$(*, from, t)$	simulator correct only if $t = t_N$	

	$y \leftarrow \lambda(s)$
	if $y \neq \phi$:
	  send $(\lambda(s), self, t)$ to *parent*
	$s \leftarrow \delta_{int}(s)$
	$t_L \leftarrow t$
	$t_N \leftarrow t_L + ta(s)$
	send $(done, self, t_N)$ to *parent*
		send $(*, self, t)$ to i^*, where
		  $i^* = select(imm\_children)$
		  $imm\_children = \{i \in D \mid M_i.t_N = t\}$
		$active\_children \leftarrow active\_children \cup \{i^*\}$

$(x, from, t)$	simulator correct only if $t_L \leq t \leq t_N$ (ignore δ_{int} to resolve a $t = t_N$ *conflict*)	

	$e \leftarrow t - t_L$
	$s \leftarrow \delta_{ext}(s, e, x)$
	$t_L \leftarrow t$
	$t_N \leftarrow t_L + ta(s)$
	send $(done, self, t_N)$ to *parent*
		$\forall i \in I_{self}$:
		  send $(Z_{self,i}(x), self, t)$ to i
		  $active\_children \leftarrow active\_children \cup \{i\}$

$(y, from, t)$		$\forall i \in I_{from} \setminus \{self\}$:
		  send $(Z_{from,i}(y), from, t)$ to i
		  $active\_children \leftarrow active\_children \cup \{i\}$
		if $self \in I_{from}$:
		  send $(Z_{from,self}(y), self, t)$ to *parent*

$(done, from, t)$		$active\_children \leftarrow active\_children \setminus \{from\}$
		if $active\_children = \emptyset$:
		  $t_L \leftarrow t$
		  $t_N \leftarrow min\{M_i.t_N \mid i \in D\}$
		  send $(done, self, t_N)$ to *parent*

Figure 2.14: DEVS Simulation Procedure

$t \leftarrow t_N$ of topmost coordinator
repeat until $t = t_{end}$
  send $(*, meta, t)$ to topmost coupled model top
  wait for $(done, top, t_N)$
  $t \leftarrow t_N$

Figure 2.15: DEVS Simulation Procedure Main Loop

messages, t is the simulation time and t_N is the next-event-time. The $(*, from, t)$ message indicates that an internal event $*$ is due.

When a coordinator receives a $(*, from, t)$ message, it selects an imminent component i^* by means of the tie-breaking function *select* specified for the coupled model and routes the message to i^*. Selection is necessary as there may be more than one imminent component (with minimum next remaining time).
When an atomic simulator receives a $(*, from, t)$ message, it generates an output message $(y, from, t)$ based on the old state s. It then computes the new state by means of the internal transition function. Note how in DEVS, output messages are only produced while executing internal events. When a simulator outputs a $(y, from, t)$ message, it is sent to its parent coordinator. The coordinator sends the output, after appropriate output-to-input translation, to each of the influencees of i^* (if any). If the coupled model itself is an influencee of i^*, the output, after appropriate output-to-output translation, is sent to the the coupled model's parent coordinator.

When a coordinator receives an $(x, from, t)$ message from its parent coordinator, it routes the message, after appropriate input-to-input translation, to each of the affected components.
When an atomic simulator receives an $(x, from, t)$ message, it executes the external transition function of its associated atomic model.
After executing an $(x, from, t)$ or $(y, from, t)$ message, a simulator sends a $(done, from, t_N)$ message to its parent coordinator to prepare a new schedule. When a coordinator has received $(done, from, t_N)$ messages from all its components, it sets its next-event-time t_N to the minimum t_N of all its components and sends a $(done, from, t_N)$ message to its parent coordinator. This process is recursively applied until the top-level coordinator or *root coordinator* receives a $(done, from, t_N)$ message.

As the simulation procedure is synchronous, it does not support a-synchronously arriving (real-time) external input. Rather, the environment or Experimental Frame should also be modelled as a DEVS component.

To run a simulation experiment, the *initial conditions* t_L and s must first be set in *all* simulators of the hierarchy. If t_N is kept in the simulators, it must be recursively set too. Once the initial conditions are set, the main loop described in Figure 2.15 is executed.

2.3.5 The parallel DEVS formalism

As DEVS is a formalization and generalization of sequential discrete-event simulator semantics, it does not allow for drastic parallelization. In particular, simultaneously occurring internal transitions are serialized by means of a tie-breaking *select* function. Also, in case of *collisions* between simultaneously occurring internal transitions and external input, DEVS ignores the internal transition and applies the external transition function. Chow [Cho96] proposed the parallel DEVS (P-DEVS) formalism which alleviates some of the DEVS drawbacks. In an atomic P-DEVS

$$atomic\ P-DEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{conf}, Y, \lambda \rangle,$$

the model can explicitly define collision behaviour by using a so-called *confluent* transition function δ_{conf}. Only δ_{ext}, δ_{conf}, and λ are different from DEVS.

The external transition function

$$\delta_{ext} : Q \times X^b \to S$$

now accepts a *bag* of simultaneous inputs, elements of the input set X, rather than a single input.

The confluent transition function

$$\delta_{conf} : S \times X^b \to S$$

describes the state transition when a scheduled internal state transition and simultaneous external inputs collide.

An atomic P-DEVS model can generate multiple simultaneous outputs

$$\lambda : S \to Y^b$$

in the form of a bag of output values from the output set Y.

As conflicts are handled explicitly in the confluent transition function, the *select* tie-breaking function can be eliminated from the coupled DEVS structure:

$$coupled\ P-DEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}\rangle.$$

In this structure, all components M_i are atomic P-DEVS

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, \delta_{conf,i}, Y_i, \lambda_i\rangle, \forall i \in D.$$

For the proof of closure under coupling of P-DEVS as well as for the description of an efficient parallel abstract simulator, see [Cho96].

2.3.6 Mapping Event Scheduling models onto atomic DEVS

The event scheduling formalism ES was presented before with mapping onto DEVS in mind. Hence the general, set-based representation of the event list in that presentation. The execution (*i.e.,* the trajectories generated through simulation) semantics has remained the same, but there is more implementation freedom. As mentioned before, external events as well as output are ignored in the description. An event scheduling model

$$ES = \langle T, E, S, EL, \delta_t, \delta_\eta, \delta_S\rangle$$

is mapped onto an atomic DEVS model

$$DEVS = \langle T^{DEVS}, S^{DEVS}, \delta_{int}^{DEVS}\rangle$$

with a time base

$$T^{DEVS} = T = \mathbb{R}$$

and state set

$$S^{DEVS} = S \times 2^{(E \times \mathbb{R}^+_{0,+\infty})}.$$

As DEVS does only deal with time-differences rather than with absolute time, a DEVS event list EL^{DEVS} contains not absolute times, but rather time-till-scheduled-event σ:

$$EL^{DEVS} = \{(\eta, \sigma = t - t_{current}) \,|\, (\eta, t) \in EL\}.$$

This is depicted in Figure 2.16.

Note how the EL only contains events in the future, hence all $\sigma \geq 0$. This is due to the *causality principle* which states that events can only be scheduled in the future (a cause leads to an effect at a later time, the past can not be influenced). $\sigma = 0$ for the current event.

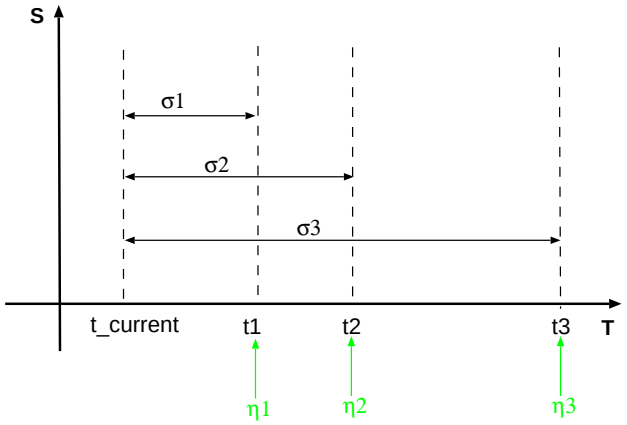

Figure 2.16: Time remaining σ until event

The DEVS time advance will advance to the next most imminent event time.

$$\begin{array}{rrcll} t_a^{DEVS}: & S^{DEVS} & \rightarrow & \mathbb{R}^+_{0,+\infty} & \\ & s^{DEVS} & \rightarrow & +\infty, & \text{if } EL^{DEVS} = \emptyset \\ & s^{DEVS} & \rightarrow & \min\{\sigma \,|\, (\eta,\sigma) \in EL^{DEVS}\}, & \text{if } EL^{DEVS} \neq \emptyset. \end{array}$$

Here, the semantics of an empty event list is correctly described as an infinite time advance. In practice, if no external events are supported, an empty event list should halt the simulation.

The DEVS internal transition function δ_{int}^{DEVS}

$$\begin{array}{rrcl} \delta_{int}^{DEVS}: & S^{DEVS} & \rightarrow & S^{DEVS} \\ & (s, EL) & \rightarrow & (s', EL') \end{array}$$

where t_{first}, η_{first} have definitions similar to those in the ES simulation procedure:

$$\begin{aligned} \sigma_{first} &= \min_{(T,\leq)}\{\sigma \,|\, (\eta,\sigma) \in EL^{DEVS}\} \\ \eta_{first} &= select\left(\{\eta \,|\, (\eta,\sigma_{first}) \in EL^{DEVS}\}\right) \\ s' &= \delta_S(\eta_{first}, s) \\ EL' &= (EL \setminus (\eta_{first}, \sigma_{first})) \cup (\delta_\eta(\eta_{first}, s), \delta_t(\eta_{first}, s)) \,. \end{aligned}$$

Again, the result of *select* is assumed to be unique (a singleton).

The above construction does not include a *proof* of equivalence. A proof would use induction. It would list all possible state/input combinations and for each of these, show how the resulting state trajectories for Event Scheduling and the derived DEVS model are identical.

An alternative to the above Event Scheduling to DEVS transformation is to construct a coupled rather than an atomic DEVS. The components (sub-models) of the coupled model are then

- one atomic DEVS encapsulating all state variables in the Event Scheduling model,
- one atomic DEVS for each event routine in the Event Scheduling model.

The component coupling should reflect the dependencies due to state modifications as well as scheduling of new events in the future.

This construction and its proof as well as similar transformations to DEVS from Activity Scanning and Process Interaction world views are the subject of future work.

2.4 The Cellular Automata formalism

Cellular automata (CA) were originally conceived by Ulam and von Neumann in the 1940s to provide a formal framework for investigating the behaviour of complex, spatially distributed systems [vN66]. Cellular Automata constitute a *dynamic, discrete space, discrete time* formalism. Space in Cellular Automata is partitioned into discrete volume elements called *cells* and time progresses in discrete steps. Each cell can be in one of a finite number of states at any given time. The "physics" of this logical universe is *deterministic* and *local. Deterministic* means that once a local physics and an initial state of a Cellular Automaton has been chosen, its future evolution is *uniquely* determined. *Local* means that the state of a cell at time $t+1$ is determined only by its own state and the states of neighbouring cells at the previous time t. The *operational semantics* of a CA as prescribed in a *simulation procedure* and implemented in a CA solver dictates that values are updated *synchronously*: all new values are calculated simultaneously.

The local physics is typically determined by an *explicit mapping* from all possible local states of a predefined neighbourhood template (*e.g.*, the cells bordering on a cell, including the cell itself), to the state of that cell after the next time-step. For example, for a 2-state $(0,1)$, 1-D Cellular Automaton, with a neighbourhood template that includes a cell and its immediate neighbours to the left and right, there will be 2^3 possible neighbourhood states $\{000,\ldots,111\}$. For each of these, we must prescribe whether a transition of the center cell state to a 1 or to a 0 will occur. For an 8-state nearest neighbour 2-D Cellular Automaton, there will be 8^5 possible neighbourhood states, and a choice of 8 states to map to for each of those.

2.4.1 The formalism

The Cellular Automata formalism *CA* fits the general structure of deterministic systems in classical systems theory [Wym67, ZPK00]:

$$CA \equiv \langle T, X, \Omega, S, \delta, Y, \lambda \rangle.$$

The formalism is specified by elaboration of the elements of the *CA* 7-tuple.

The *discrete* time base T:

$$T = \mathbb{N} \text{ (or isomorphic with } \mathbb{N}).$$

The set X:

$$X = \{TIME\_TICK\}.$$

The Cellular Automata formalism can easily be extended to non-trivial inputs (see further comments on extensions).

The set of all input segments ω:

$$\Omega.$$

An input segment ω may be restricted to a domain ($\subseteq T$) such as $[n, n+1]$:

$$\begin{aligned} \omega &: T \to X, \\ \omega_{]n,n+1]} &:]n, n+1] \to X. \end{aligned}$$

The state set S is the product of all the *finite* state sets V_i (also called *cell value sets*) of the individual cells. C is the *cell index set.*

$$S = \times_{i \in C} V_i.$$

In the usual case of Cellular Automata realized on a D-dimensional grid, C consists of D-tuples of indices from an index set I:

$$C = I^D.$$

In the standard Cellular Automata formalism, the cell space is assumed *homogeneous*: all cell value sets are identical:

$$\forall i \in C, V_i = V.$$

The cell value function v maps a cell index i onto its value $v(i)$:

$$v : C \to V.$$

The total transition function δ is constructed from the transition functions δ_i for each of the cells.

$$\begin{array}{rccc} \delta & : & \Omega \times S & \to & S, \\ & & (\omega_{]n,n+1]}, \times_{i \in C} v(i)) & \to & \times_{i \in C} \delta_i(i). \end{array}$$

The *uniformity* of Cellular Automata requires the δ_i to be based on a *single* local transition function δ_l for all cells i:

$$\forall i \in C, \delta_i(i) = \delta_l(v(\sigma_{NT}(i))),$$

where the various operators and quantities are explained below.

A neigbourhood template NT, a vector of *finite* size ξ containing *offsets* from an "origin", encodes the relative positions of neigbouring cells influencing the future state of a cell. Usually, the *nearest (adjacent)* neighbours (including the cell itself) are used. For one-dimensional Cellular Automata, a cell is connected to r local neighbours (cells) on either side, where r is a parameter referred to as the *radius* (thus, there are $2r+1$ cells in each neighbourhood). For two-dimensional Cellular Automata, two types of neighbourhoods are usually considered:

- The *von Neumann* neighbourhood of radius r

  $$NT = \{(k,l) \in C | \ |k| + |l| \leq r\}.$$

  For $r = 1$, this yields 5-cells, consisting of a cell along with its four immediate nondiagonal neighbours.
- The *Moore* neighbourhood of radius r

  $$NT = \{(k,l) \in C | \ |k| \leq r \text{ and } |l| \leq r\}.$$

  For $r = 1$, this yields 9-cells, consisting of the cell along with its eight surrounding neighbours.

The *local* nature of the Cellular Automata models lies in the fact that *only* near neigbours influence the behaviour of a state, *not all* cells as the general form of δ allows. From a modelling point of view, physical arguments in disciplines ranging from physics to biology and artificial life support this assumption [Wol86]. From a simulation point of view, efficient solvers for the Cellular Automata formalism can be constructed which take advantage of locality (see below). Note how the $NT[j]$ are offsets between C elements which are again elements of C (with an appropriate C such as $\mathbb{N}^D$).

$$NT \in C^{\xi}.$$

The function σ_{NT} *shifts* the neighbourhood template NT to be centered over i:

$$\begin{array}{rccc} \sigma_{NT} & : & C & \to & C^{\xi}, \\ & & i & \to & \tau \quad \text{where } \forall j \in \{1, \ldots, \xi\} : \tau[j] = i + NT[j]. \end{array}$$

For all possible combinations of size ξ of cell values, the local transition function δ_l prescribes the transition to a new value:

$$\begin{array}{rccc} \delta_l & : & V^{\xi} & \to & V, \\ & & [v_1, \ldots, v_{\xi}] & \to & v_{new}. \end{array}$$

Thanks to the aforementioned uniformity of Cellular Automata, the same δ_l is used for each element of the cell space. The number of possible combinations of cell values is $\#V^{\xi}$ and the number of distinct results of δ_l is $\#V$ (# is the cardinality function).

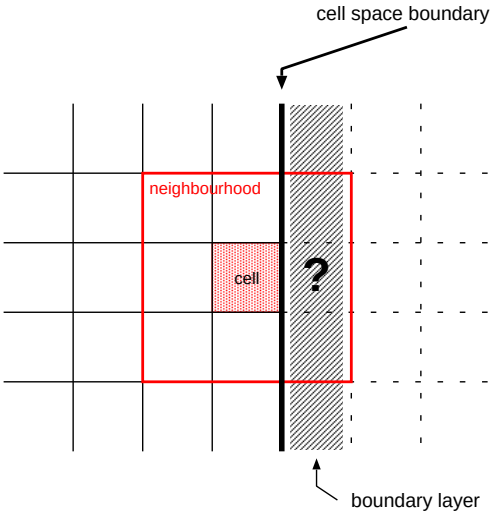

Figure 2.17: Boundary Conditions for a finite cell space

In the above, δ was constructed for an elementary time advance $n \rightarrow n+1$. The transitivity requirement for deterministic system models:

$$\forall t_x \in [t_i, t_f]; s_i \in S,$$

$$\delta(t_i, \omega_{[t_i,t_f]}, s_i) = \delta(\omega_{[t_x,t_f]}, \delta(\omega_{[t_i,t_x]}, s_i))$$

is satisfied by *construction* (*i.e.,* the definition, combined with iteration over n).

It is possible to "observe" the Cellular Automaton by projecting the total state S onto an output set Y by means of an output function λ

$$\lambda : S \rightarrow Y,$$

where Y commonly has a structure similar to that of S.

We shall now discuss the *structure* of the cell space. Usually, a D-dimensional *grid*, centered around the origin, is used. Most common choices are $D \in \{1,2,3\}$. In one dimension, a linear array of cells is the only possible geometry for the grid. In two dimensions, there are three choices:

- **triangular grid**: has a small number of nearest neigbours but is hard to visualise on square grid oriented computers.
- **square grid**: is easy to visualise, but (computationally) *anisotropic* (*i.e.,* a wave propagates faster along the primary axes than along the diagonals).
- **hexagonal grid**: has the lowest anisotropy of all grids but computer visualisation is is hard to implement.

Arbitrary cell space structures are possible (and corresponding cell shapes when visualising), though not practical.

Although the above mathematical formalism is perfectly valid, it can not be simulated *in practice*. For simulation to become possible, the cell space needs to be *finite*. In particular, the cell index set C must be finite. L, the length of the grid becomes finite, leading to a cell space of L^D cells.

When a finite cell space is used, the application of the transition function at the edges poses a problem as values are needed outside the cell space. As shown in Figure 2.17 for a 2-D cell space, *boundary conditions* need to be specified in the form of cell values *outside* the cell space. Two common approaches –also used in the specification and solution of partial differential equations– are

- **explicit** boundary conditions. Extra cells outside the perimeter of the cell space hold boundary values (*i.e.,* C and v are extended). The amount of extra border cells is determined by the size of (the perime-

ter of) the cell space as well as by the size (and shape) of the neighbourhood template. Boundary conditions may be time varying.

- **periodic** boundary conditions. The cell space is assumed to "wrap around": the cells at opposite ends of the grid act as each other's neighbours. In 1-D, this results in a (2-D) circle, in 2-D, in a (3-D) torus. By construction, the boundary conditions are time-varying.

2.4.2 Implementation of a CA solver

The Solver Structure

Algorithm 2 is the backbone of any cellular automata solver.

Algorithm 2 CA Simulation procedure

$\forall i \in C$: initialise $v(i)$	{Initialise Cell Space}
if explicit boundary conditions **then**	
$\forall i \in$ boundary(C): initialise $v(i)$	{Boundary extension of $v()$}
end if	
if periodic boundary conditions **then**	
$\forall i \in C \cup$ boundary(C): $v(i) \leftarrow v(i \bmod L)$	{Modulo extension of $v()$; assume $0 \ldots L-1$ indexing}
end if	
for $n := n_s$ to n_f **do**	
$\forall i \in C$: $v_{new}(i) \leftarrow \delta_l(v(\sigma_{NT}(i)))$	{One-step state transition computation}
$v \leftarrow v_{new}$	{Switch value buffers}
$n \leftarrow n+1$	{Time Advancement}
end for	

As the definition requires synchronous calculation, whereby new values only depend on old values (and not on new values) of neighbouring cells, a second value function v_{new} is needed to hold copies of the previous value. Note how a value functions is usually efficiently implemented as a lookup in a value *array*.

Improving Solver Performance

The performance of a Cellular Automaton solver is obviously related to an appropriate choice of data structures and algorithms. There are four main techniques [ROE99] for improving solver performance:

1. Lookup table:

   generally, a cell takes on a value v_{new} which is computed on the basis of information in the cell's neighbourhood. One may attempt to pack the neighbourhood value information bitwise into an integer *neigh* which can subsequently be used as an *index* into a *lookup table*. The *lookup* table *encodes* the local transition function δ_l:

   $$v_{new}(i) = lookup[neigh].$$

   The *lookup* values are pre-computed from a δ_l specification before simulating the Cellular Automaton. The *lookup* vector may also serve as an efficient means of model storage.

2. Neighbourhood shifting:

   in stepping through the cells, one repeatedly computes a cell's *neigh*, then computes the *neigh* of the next cell, and so on. Because the neighbourhoods overlap, a lot of the information in the next cell's *neigh* is the same as in the old cell's. With an appropriate representation, it is possible to *left shift* out the old info and *OR* in the new info.

3. Pointer swap:

   to run a CA, one needs two buffers, one for the current cell space (v at t), and one for the updated cell space (v_{new} at $t+1$). After the update, the updated cell space should *not* be *copied* into the current one

(though a naïve implementation of line 10 in Algorithm 2 would do so). Assuming the value functions v and v_{new} are encoded as arrays, swapping pointers is $O(L^D)$ faster.

4. Assembly language "inner loop":

   when processing a 2-D Cellular Automaton of VGA size, the cell space has approximately $300,000$ cells. This implies *neigh* will be assembled and *lookup* applied about $300,000$ times per time-step (one screen). This means the *inner loop* must be as efficient as possible. In particular, coding this in assembly language, reducing the total number of clock cycles of the instructions in the inner loop, can lead to a significant performance gain.

5. Sparse *vs.* dense configurations:

   if the rule set is known to lead to *sparse* configurations, as in the case of the Game of Life with a small initial pattern, one can use *sparse matrix techniques.* That is, one can just compute in the vicinity of occupied cells. Generally, these techniques do not compile as efficiently as a full matrix method, because there is more indirect addressing and branching. However, one can include both a sparse and full matrix method in the same program, and switch when the *cross-over density* is reached.

6. Periodic boundary conditions:

   there are two basic methods for handling periodic boundary conditions efficiently:

   (a) Coding for fast modulo arithmetic.
   The brute force method of doing modulo arithmetic on index variable `i` for a range of $0 \ldots R-1$ in `C` is

   ```
   (i + offset) % R
   ```

   On some architectures (*e.g.*, some Sun Sparcstations) it is actually faster to do

   ```
   register int tmp = i + offset;
   (tmp >= R) ? tmp - R : tmp
   ```

   if `offset` is positive and similarly if it is negative.
   If `R` is a power of 2, better performance can be obtained by means of

   ```
   (i + offset) & R
   ```

   when offset is positive and

   ```
   (i + offset + R) & R
   ```

   when offset is negative.

   (b) Using a larger array and copying the boundary cells between iterations

2.4.3 Examples

The [CAw99] at the Santa Fe Institute hosts a plethora of Cellular Automata examples. The site is mainly devoted to the study of Artificial Life, one of the prominent uses of Cellular Automata. Artificial Life research tries to explain and reproduce, ab-initio, all physical and biological phenomena.

Simple 2-state, 1-D Cellular Automaton of length 4

Figure 2.18 demonstrates the simulation procedure for a simple 2-state ($\{0,1\}$), 1-D Cellular Automaton of length 4 with periodic Boundary Conditions and initial condition 1101. The local transition function (a 1-D version of the Game of Life) is

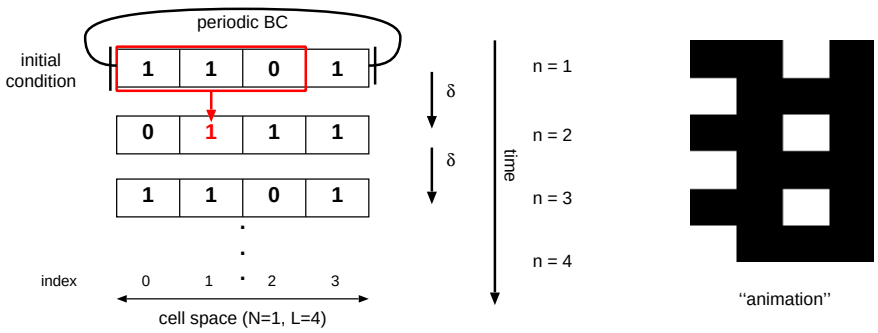

Figure 2.18: 2-state, 1-D Cellular Automaton of length 4

```
# 2 dimensional game of life

2 dimensions of 0..1

sum := [ 0,  1] + [ 1,  1] + [ 1,  0] +
       [-1,  1] + [-1,  0] + [-1, -1] +
       [ 0, -1] + [ 1, -1]

cell := 1 when (sum = 2 & cell = 1) | sum = 3
     := 0 otherwise
```

Figure 2.19: "cellang" specification of Conway's game of Life

$$\delta_l: \quad \begin{array}{ll} 000 \to 0 & 100 \to 0 \\ 001 \to 0 & 101 \to 1 \\ 010 \to 0 & 110 \to 1 \\ 011 \to 1 & 111 \to 0 \end{array}$$

For a 1-D Cellular Automaton, "animation" of the cell space can be visualised by colour coding the cell values (here: 0 by white and 1 by black) and by mapping t onto the vertical axis which leads to the 2-D image in Figure 2.18.

The Game of Life

Developed by Cambridge mathematician John Conway and popularized by Martin Gardner in his Mathematical Games column in Scientific American in 1970 [Gar70], the game of Life is one of the simplest examples of a Cellular Automaton. Each cell is either alive (1) or dead (0). To determine its status for the next time step, each cell counts the number of neighbouring cells which are alive. If the cell is alive and has 2 or 3 alive neighbours, then the cell is alive during the next time step. With fewer alive neighbours, a living cell dies of loneliness, with more, it dies of overcrowding. Many interesting patterns and behaviours have been investigated over the years. An example of a high-level *cellang* [Eck98] specification (*i.e.*, δ_l is written implicitly) of the Cellular Automata model is given in Figure 2.19. In combination with boundary conditions and an initial condition, this specification allows for model solving.

2.4.4 Formalism extensions

The Cellular Automata formalism can easily be extended in different ways:

1. Addition of *inputs*. The formalism as presented above is *autonomous*: there are no (non-trivial) inputs into the system. An intuitively appealing way of adding inputs is to associate an input with each cell.

The input set X will thus have a structure similar to the state set S.

2. The requirement of having the same cell value set for each cell can be relaxed to obtain *heterogeneous* Cellular Automata whereby not necessarily $\forall i \in C : V_i = V$. The homogeneous case can always be emulated by constructing V as the union of all individual cell value sets:

$$V = \bigcup_{i \in C} V_i.$$

3. The requirement of having the same local transition function δ_l for each cell can be relaxed to obtain *non-uniform* Cellular Automata. Obviously, in that case, it is no longer possible to use the performance-enhancing techniques descibed above.

4. As is demonstrated in Figure 2.19, a modelling language may allow for high-level representations. Agents are a typical example of such a high-level construct. Here, δ_l is no longer specified explicitly.

The "grid of cells" idea can also be used for continuous models. In particular, the local dynamics of cells can be described by System Dynamics models. Obviously, simulation will be done by first transforming the model to a flat DAE model and subsequently solving that continuous model.

2.4.5 Mapping the Cellular Automata formalism onto DEVS

Cellular Automata are a simple form of discrete-event models, of DEVS in particular. Describing a Cellular Automaton as a simple atomic DEVS is thus straightforward. Thanks to the general nature of DEVS, all extensions of the Cellular Automata formalism mentioned before can also be mapped onto DEVS.

It is more rewarding however to map a CA onto a coupled model, whereby every CA cell's dynamics is represented as an atomic model and the dependency between one cell and its neighbourhood is represented by the coupled model's coupling information. In what follows, a CA is mapped onto a coupled P-DEVS as this mapping is more elegant than that onto the original DEVS. In addition, the resultant parallel DEVS holds more potential for parallel implementation. The coupled parallel DEVS representation presented here, corresponds to the Cellular Automaton specification in section 2.4.1:

$$P-DEVS-CA = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle.$$

As the Cellular Automaton in section 2.4.1 did not incorporate external input,

$$X_{self} = \emptyset.$$

Typical output of the Cellular Automaton consists of all the cell values

$$Y_{self} = \times_{i \in D} V.$$

Components in the coupled parallel DEVS model correspond to CA cells. Indexing uses the CA index set:

$$D = C.$$

The $\{M_i\}$ are atomic P-DEVS components, described in more detail later.

The set of influencees of a component/cell i is constructed by means of the CA's neighbourhood template NT. The NT contains influencer rather than influencee information. Thus, the offset information needs to be mirrored with respect to the origin (*i.e.*, its inverse with respect to addition of offsets needs to be calculated) to obtain the influencees of component i:

$$I_i = \{ j \in C | \, j = i - offset, \; offset \in \{NT[k] | k = 1, \ldots, \xi\} \setminus \{0\}\}$$

As DEVS does not allow $i \in I_i$, the offset 0, if present in NT, is not included. A state transition of a CA cell usually does depend on the old state of the cell itself. This is encoded in the atomic P-DEVS (confluent)

transition function rather than by means of an external self-coupling. Note how $j = i - offset \in C$ may need to be relaxed depending on the particular boundary conditions. Cells outside C may need to be considered (and initialized).

As there is no external input, the input-to-input translation $Z_{self,i}$ is not needed.

The i, j output-to-input translation converts the output of a cell (*i.e.*, the cell's value) into a tuple containing that value and the offset between the two cells:

$$\begin{array}{rcccc} Z_{i,j} & : & Y_i & \rightarrow & X_j \\ & & s_i & \rightarrow & (s_i, i-j). \end{array}$$

The output-to-output translation translates the output of each cell into a tuple with the output in the position corresponding to the cell's index:

$$\begin{array}{rcccc} Z_{i,self} & : & Y_i & \rightarrow & Y_{self} \\ & & s_i & \rightarrow & (\ldots, s_i, \ldots). \end{array}$$

Individual cells are mapped onto atomic P-DEVS components:

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, \delta_{confl,i}, Y_i, \lambda_i \rangle, \forall i \in D.$$

Values of the components/cells are those from the cell value set

$$S_i = V.$$

The time advance is set to the same arbitrary non-zero value Δ for all cells to allow for synchronous operation:

$$ta_i(s_i) = \Delta.$$

The internal transition function does not modify the component's state:

$$\delta_{int}(s_i) = s_i.$$

The output function sends out a set containing only the cell value:

$$\lambda(s_i) = \{s_i\}, \text{ where } s_i \in Y_i = V.$$

The external transition function is not used as there is no global external input into the CA. Due to the synchronous operation, whereby internal transitions and external inputs will always collide, *only* the confluent transition function is used.

$$\delta_{conf,i}(s_i, e_i, x_i^b) = \delta_l(v_i),$$

where v_i is a vector with the same dimensions as the neighbourhood template NT with values

$$\begin{array}{rcl} v_i[\eta] & = & s_i, \text{ for the } \eta \text{ for which } NT[\eta] = 0; \\ v_i[proj_{offset}(x)] & = & proj_{value}(x), \ \forall x \in x_i^b. \end{array}$$

Messages communicate values of neighbouring cells, as well as offsets from the current cell:

$$X_i^b = \{(v, offset) | v \in V, offset \in C\}.$$

As in the case of the mapping from Event Scheduling to atomic DEVS, the above construction is not a proof. A proof of a generalized mapping onto variable-structure P-DEVS to allow for the representation of infinite CAs will be a continuation of the current work.

As a demonstration of the above approach, Murato coded the Game of Life CA in his `a-DEVS-0.2` [Nut99] parallel DEVS implementation.

2.5 The Differential and Algebraic Equation formalisms

For many problems, the result of the *modelling process* is a set of ordinary differential equations, often accompanied by algebraic constraint equations, thus forming a set of Differential and Algebraic Equations (*DAE*) [EBB+99]. The initial values of the state variables need to be specified with the model. Initial values are only required at simulation time when each initial value will lead to a unique solution of the DAEs. This implies that the DAE is mathematically formulated as an *Initial Value Problem*. The Initial Value DAE is used for simulation or other analysis activities such as controller design (possibly after linearisation), parameter estimation, or performance measure optimisation.

Currently, Boundary Value Problems are only supported (within the WEST++ modelling and simulation environment described in the next chapter) by constructing an appropriate *shooting* problem, an Initial Value Problem where some unknown initial values are varied during a series of simulation experiments in an attempt to satisfy the boundary values. The reason for not supporting Boundary Value Problems directly is the inability to guarantee the existence of unique solutions as well as the (resulting) lack of generally applicable solvers.

Hybrid DAEs may have *discontinuities* or the *structure* of a Hybrid DAE may change at certain points in time. *Events* are used to stop continuous integration at discontinuities of a hybrid DAE. After applying the discontinuous change (possibly changing state variable values and/or model equations), the integration is restarted. As described by Park and Barton [PB96, Bar00], Hybrid DAE problems can be treated as a sequence of continuous DAE problems separated by events (time-events if pre-scheduled at a known time or state-events if dependent on a condition over the state variables). In the following, we will describe the symbolic transformations of (implicit) DAEs to (explicit) forms which can be solved more efficiently by numerical solvers. Discontinuities (and structural model change) are an orthogonal issue and can be ignored without loss of generality.

In the following, different levels of Differential Algebraic Equation models will be presented in a bottom-up fashion. In this approach, it will be shown how each higher level form can be transformed into the lower level one. Chaining these transformations will allow transformation from the highest, most re-usable level to the lowest level. At this lowest level, the numerical solution is much more efficient, but re-usability is low. By introducing formalism transformations, both goals (re-use as well as efficient numerical simulation) can be satisfied.

2.5.1 Differential Algebraic Equations: Causal Sequence

The formalism

At the lowest level, we start from an Ordinary Differential Equation (ODE) with $x(t)$, $u(t)$, f and p vectors (of matching dimensions) or scalars

$$\frac{d^n x}{dt^n} = f(\frac{d^{n-1}x}{dt^{n-1}}, \ldots, x, u, t, p).$$

x are the state variables, u the input functions, t the independent time, and p, the parameters.

The above is posed as an initial value problem over an interval $[a,b]$ with inital conditions

$$\begin{aligned} x(t=a) &= x_0 \\ \tfrac{dx}{dt}(t=a) &= x_0^{(1)} \\ &\vdots \\ \tfrac{d^{n-1}x}{dt^{n-1}}(t=a) &= x_0^{(n-1)} \end{aligned}$$

The above higher order equation can always be transformed into a set of first order Ordinary Differential

Equations by introducing auxiliary variables corresponding to the derivatives of x.

$$\begin{cases} x & = & x_0 \\ \frac{dx}{dt} & = & x_1 \\ \frac{dx_1}{dt} & = & x_2 \\ & \dots & \\ \frac{dx_{n-1}}{dt} & = & x_n = f(x_{n-1}, x_{n-2}, \dots, x_1, x_0, u, t) \end{cases}$$

This is necessary to match the capabilities of most available numerical solvers. From now on, we will only consider first order ODEs.

Solving Ordinary Differential Equations

With a discretisation of the interval $[a,b]$ in N equidistant intervals Δt

$$t_j = a + j\Delta t, \;\; \Delta t = \frac{b-a}{N},$$

a Taylor series expansion can be written (if the solution $x(t)$ has continuous derivatives upto order $r+1$ over $[a,b]$, there exist θ_j).

$$\begin{aligned} x(t_j + \Delta t) = x(t_j) \quad & + \tfrac{\Delta t}{1!}\, x^{(1)}(t_j) + \tfrac{\Delta t^2}{2!}\, x^{(2)}(t_j) + \ldots + \tfrac{\Delta t^r}{r!}\, x^{(r)}(t_j) \\ & + \tfrac{\Delta t^{r+1}}{(r+1)!}\, x^{(r+1)}(t_j + \theta_j \Delta t), \qquad 0 < \theta_j < 1;\; j = 0, 1, \ldots, N-1 \end{aligned}$$

The first order derivative $x^{(1)}$ is given by the Right Hand Side (RHS) of the differential equation. In case this RHS is an analytical function, symbolic derivation may provide expressions for higher derivative terms. Different numerical integration methods for Ordinary Differential Equations correspond to different approximations for these higher order derivatives.

Integration methods are employed based on various quality metrics such as accuracy, stability, speed, memory consumption, and the ability to operate in a real-time environment. The methods are characterized by

- the order of approximation r,
- single-step or multi-step (when multiple old values are needed to compute a next value; this requires a one-step start-up method),
- the number of intermediate evaluations (between t_j and t_{j+1}),
- the symmetry of the method (only evaluation in t_j or t_{j+1} or balanced),
- fixed *vs.* adaptive step size (the stepsize is halved or doubled based on an error estimate),
- explicit *vs.* implicit (using future values, for stiff problems).

For more details on numerical simulation (without discontinuities), we refer to [PTVF92] and the WEST++ experimentation environment.

Figure 2.20 depicts the structure of a hybrid simulation kernel. On the right hand side, the integration with adaptive stepsize and state-event location is shown. On the left hand side, event-scheduling is shown. The simulator alternates between continuous simulation and discrete event handling. For more details, we refer to [Van00, JLZS00, MB01].

Continuous System Simulation Languages (CSSLs)

In 1967, the Continuous System Simulation Language standard (CSSL) was proposed [SAF+67]. It provides a standard for the representation of ODEs and algebraic equations as used in simulation. The standard is still in use to date in simulation languages such as CSSL-IV, ACSL [ACS95], and ADSIM/RTS [VV99].

CSSL satisfies the following requirements:

- An easy model description allowing both equations and “function block” forms (by means of macros).

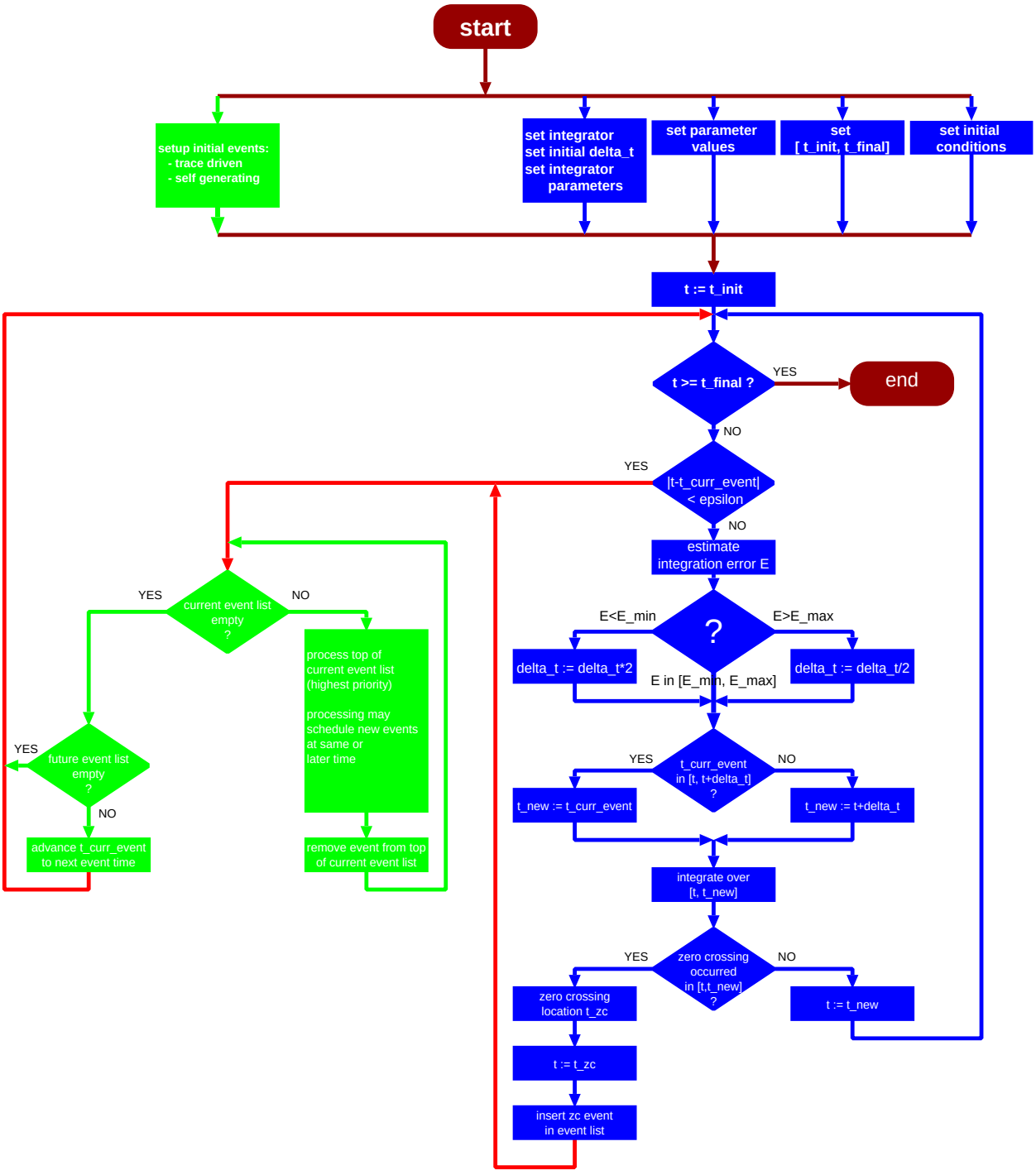

Figure 2.20: Hybrid simulation kernel structure

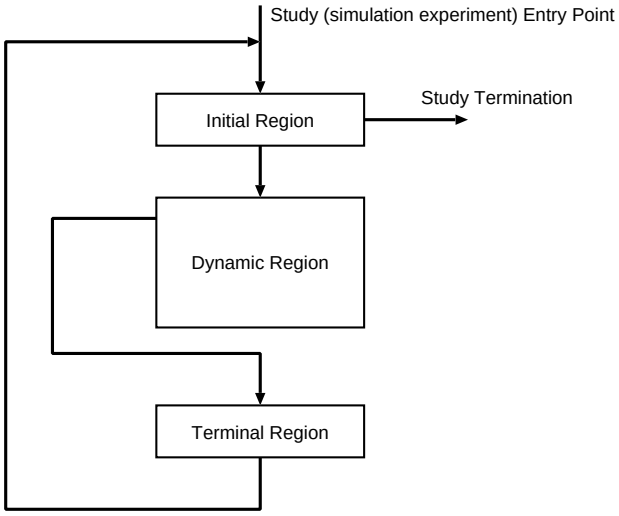

Figure 2.21: CSSL simulation study

- Integrator control:
  - selecting an integrator,
  - (initial) step size selection,
  - error control,
  - variable initialization,
  - parameter setting,
- Documentation of model and experiments.
- Structured: model is separated from experiment.

Figure 2.21 shows the structure of a CSSL simulation study. From a model description as shown below, the `INITIAL` section is executed exactly once. Then, the differential equations in the `DYNAMIC` section are solved using appropriate numerical solvers. This is an iterative process. Finally, the `TERMINAL` section is executed exactly once.

```
INITIAL
  X  = X0
 DX  = DX0
DYNAMIC
 DERIVATIVE
  DX' = F-B*X-A*DX
   X' = DX
TERMINAL
 END_X = X
```

As shown in Figure 2.22, the `INITIAL` section allows interactive intervention by a user (by means of a command interpreter). After executing user specified `INITIAL` code (such as constant and parameter calculations), the integrators are initialised.

Figure 2.23 shows the structure of the `DYNAMIC` region. One part, containing only algebraic equations, takes care of input/output. The integration subregion calls appropriate solver(s) (integrators) to solve the ODEs specified in the `DERIVATIVE` section(s).

Some CSSL compilers are capable of “sorting” algebraic equations. This procedure will be described in the next section.

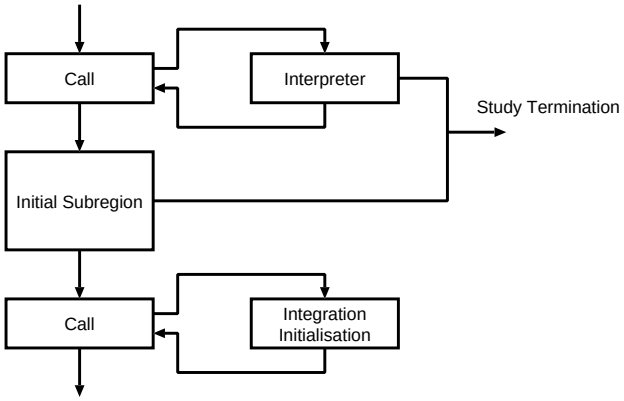

Figure 2.22: CSSL initial region

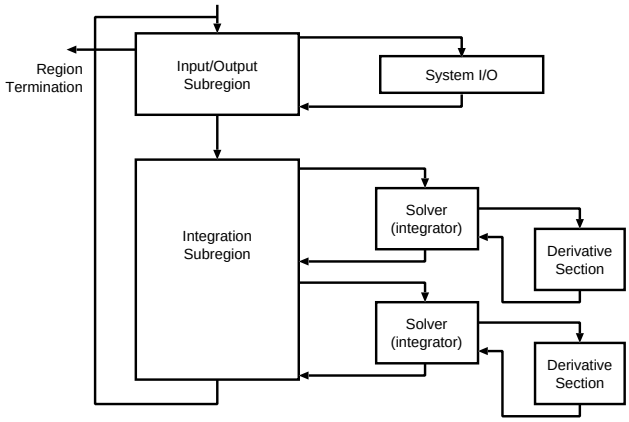

Figure 2.23: CSSL dynamic region

Neutral solver-level model representation

One problem of the CSSL standard is that it defines a language structure rather than an Application Programmer's Interface (API). This makes interfacing with a simulator non-standard. A more modular approach is to define an interface (API) between model solvers and models. Also, an interface (API) between a (scripted) experimentation environment and solver and model must be defined. The structure of such an arhcitecture is depicted in Figure 2.24. There are several advantages to such an approach:

1. It allows for independent plug-and-play of solvers and models.

2. The internals of both solvers and models are not specified. The implementors have total freedom, even in the implementation language (Fortran, C, C++). In WEST++, the binding is C with Fortran conventions (column major order) for matrix memory layout.

3. As the interface is defined at a binary (link) level, reverse-engineering of models is difficult.

In WEST++, the link-level representation used for models is named MSL-EXEC (EXECution level rather than USER level). MSL-EXEC is defined as C++ classes and contains both symbolic and computational model information. To access model information, the solver needs access to the standard methods ComputeOutput(), ComputeInitial(), ComputeTerminal(), and ComputeState() which correspond to the equivalent parts of the CSSL model specification. Communication of data between a solver and a model must be efficient. Hence the use of `#define` to *alias* vector elements to symbolic names. Communication is done by passing (references to) vectors of parameters, input variables, output variables, derived state variables, and

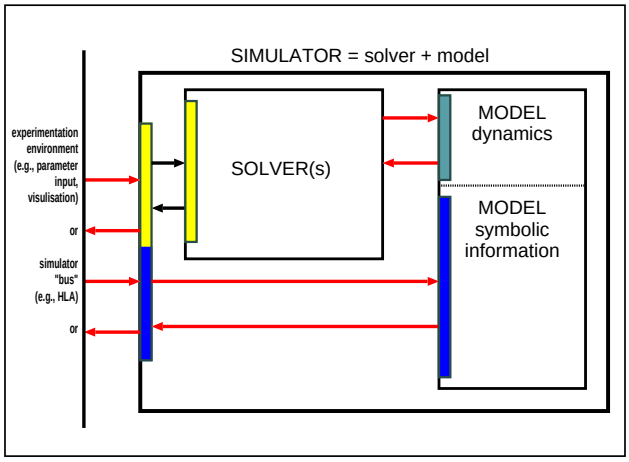

Figure 2.24: Model-solver architecture

derivatives of those. The MSL-EXEC model representation of a simple model

$$\begin{cases} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -x \\ x\_out &= x \\ y\_out &= y \end{cases}$$

is given below.

```
#include <math.h>
#include <assert.h>
#include "MSLE.h"
#include "MSLExternal.h"
#include "MSLU.h"
#include "Circle.h"

#define _t_ IndepVarValues[0]
#define _x_out_ OutputVarValues[0]
#define _y_out_ OutputVarValues[1]
#define _x_ DerStateVarValues[0]
#define _y_ DerStateVarValues[1]
#define _D_x_ Derivatives[0]
#define _D_y_ Derivatives[1]

CircleClass :: CircleClass(StringType name_arg)
{
  set_name(name_arg);
  set_description("Circle test.");
  set_class_name("CircleClass");

  set_no_indep_vars(1);
  set_indep_var(0, new MSLEIndepVarClass("t", "s"));

  set_no_output_vars(2);
  set_output_var(0, new MSLEOutputVarClass("x_out", "", 0));
  set_output_var(1, new MSLEOutputVarClass("y_out", "", 0));

  set_no_der_state_vars(2);
  set_der_state_var(0, new MSLEDerStateVarClass("x", "", 0.1));
  set_der_state_var(1, new MSLEDerStateVarClass("y", "", 0.1));

  set_no_indep_var_values(1);
```

```
  GetIndepVar(0)->LinkValue(this, MSLE_INDEP_VAR, 0);

  set_no_output_var_values(2);
  GetOutputVar(0)->LinkValue(this, MSLE_OUTPUT_VAR, 0);
  GetOutputVar(1)->LinkValue(this, MSLE_OUTPUT_VAR, 1);

  set_no_der_state_var_values(2);
  GetDerStateVar(0)->LinkValue(this, MSLE_DER_STATE_VAR, 0);
  GetDerStateVar(1)->LinkValue(this, MSLE_DER_STATE_VAR, 1);
  GetDerStateVar(0)->LinkInitialValue(this, 0);
  GetDerStateVar(1)->LinkInitialValue(this, 1);
  GetDerStateVar(0)->LinkDerivative(this, 0);
  GetDerStateVar(1)->LinkDerivative(this, 1);

  Reset();
}

void CircleClass :: ComputeOutput(void)
{
  _x_out_ = _x_;
  _y_out_ = _y_;
}

void CircleClass :: ComputeInitial(void)
{
}

void CircleClass :: ComputeState(void)
{
  _D_x_ = _y_;
  _D_y_ = -_x_;
}

void CircleClass :: ComputeTerminal(void)
{
}

#undef _t_
#undef _x_out_
#undef _y_out_
#undef _x_
#undef _y_
```

It is noted that in MSL-EXEC specifications, the *order* in which algebraic equations are written matters as imperative programming languages such as C++ have sequential semantics. Hence the name “Differential Algebraic Equations: Causal Sequence”: all equations have a single variable on the Left Hand Side (LHS) (computationally causal) and the order of equations matters (sequence).

2.5.2 Differential Algebraic Equations: Causal Set

The formalism

The need to order equations mentioned in the previous chapter only applies to algebraic equations. As numerical integrators calculate new values based on old (previous time-step) values, the order of evaluation of Right Hand Sides of derivative equations does not matter. Thus, from here on, we will only discuss the algebraic equations part of our DAEs.

To illustrate the problem when coding mathematical *sets* of equations in a language such as C++ with *sequence* semantics, consider the following set of equations, with $u()$ a function call (input, considered

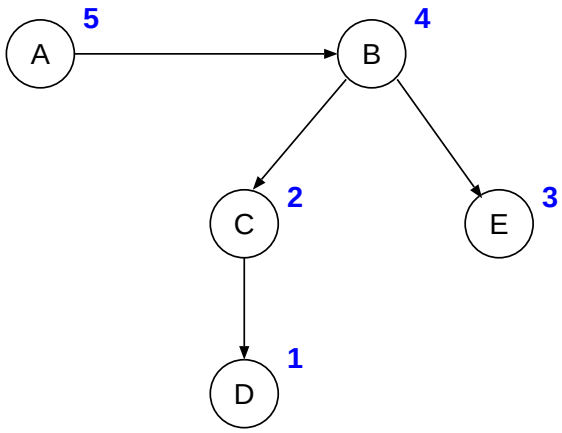

Figure 2.25: Sorting: Depth First Search, post-order numbering

known):

$$\begin{cases} a &= b^2+3 \\ b &= sin(c \times e) \\ c &= \sqrt{d-0.5} \\ d &= \pi/2 \\ e &= u() \end{cases}$$

When coded as a sequence in MSL-EXEC, uninitialized variables will be given a 0-value which leads to erroneous results (and even an exception in this case):

$$\begin{bmatrix} a &= 3 \\ b &= 0 \\ c &= \sqrt{-0.5} \ \text{(exception in} \mathbb{R}) \\ d &= \pi/2 \\ e &= u() \end{bmatrix}$$

If equations are re-arranged however, it is possible to compute the correct solution of the set of equations by means of a sequence:

$$\begin{bmatrix} d &= \pi/2 \\ e &= u() \\ c &= \sqrt{d-0.5} \\ b &= sin(c \times e) \\ a &= b^2+3 \end{bmatrix}$$

Equations must be *sorted* in the reverse order of their dependencies. Obviously, traversing the resulting sequence of equations is much faster than the numerical solution of the original implicit set of equations. Also, in the latter case, a meaningful initial guess for the unknowns must be found.

Mapping onto DAE Causal Sequence: sorting

To sort the equations it suffices to build a dependency graph and perform a Depth First Search with post-order numbering on this graph as shown in Figure 2.25. The numbers indicate the order in which equations need to be written. The Depth First Search (DFS) algorithm is given in Algorithm 3.

Algorithm 3 DFS sorting

```
dfsCounter ← 1                                   {initialize global DFS number}
for all v ∈ V do
  dfsNr[v] ← 0                                   {initialize vertex DFS number}
end for
for all v ∈ V do
  if dfsNr[v] = 0 {not yet visited} then
    DFS(v)                                       {DFS starting from v}
  end if
end for
DFS(v ∈ V) ≡                                     {DFS definition}
if dfsNr[v] = 0 {not yet visited} then
  dfsCounter ← dfsCounter + 1
  for all w ∈ children(v) do
    DFS(w)                                       {DFS starting from child w}
  end for
  dfsNr[v] ← dfsCounter                          {post-order numbering}
end if
```

Mapping onto DAE Causal Sequence: cycle detection

In some cases, sorting is not possible due to a *dependency cycle*, *strong component* or *algebraic loop*. The example below

$$\begin{cases} x &= y+16 \\ y &= -x-z \\ z &= 5 \end{cases}$$

can *never* be sorted due to a dependency cycle between x and y.

Once detected, an algebraic loop may be solved as an implicit set of algebraic equations.

$$\left[\begin{array}{rcl} z = 5 & & \\ \begin{cases} x-y &=& -6 \\ x+y &=& -z \end{cases} & & \end{array}\right.$$

An implicit set of n equations in n unknowns may be

- non-linear, solved using
  - a symbolic solution using Gröbner bases [DST93], or
  - a numerical solution using a solver such as Broyden's method (implemented in WEST++) for finding roots of systems of non-linear equations [PTVF92].
- linear, solved using:
  - an analytical solution using Cramer's rule, or
  - a numerical solution in case the analytical solution grows too large.

In the example, this leads to

$$x = \frac{\begin{vmatrix} -6 & -1 \\ -z & 1 \end{vmatrix}}{\begin{vmatrix} 1 & -1 \\ 1 & 1 \end{vmatrix}} = \frac{-6-z}{2} \; ; \; y = \frac{\begin{vmatrix} 1 & -6 \\ 1 & -z \end{vmatrix}}{\begin{vmatrix} 1 & -1 \\ 1 & 1 \end{vmatrix}} = \frac{6-z}{2}$$

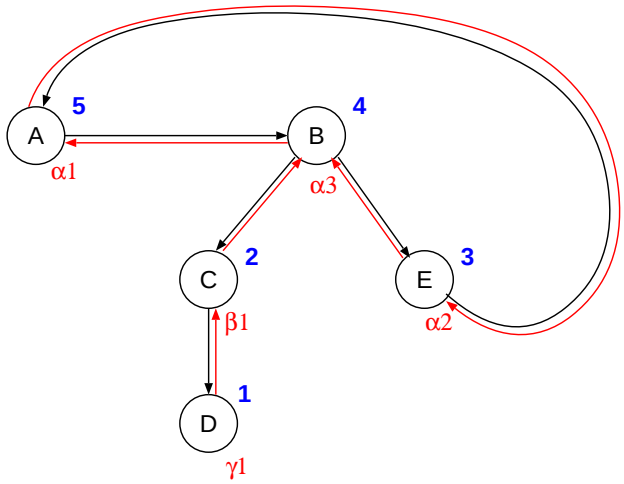

Figure 2.26: Algebraic loop (dependency cycle) detection

and finally

$$\begin{bmatrix} z &= 5 \\ x &= \frac{-6-z}{2} \\ y &= \frac{6-z}{2} \end{bmatrix}$$

The question remains how to detect and extract algebraic loops (cycles in the dependecy graph). A simple loop detection algorithm such as:

1. Build dependency matrix D
2. Calculate transitive closure D^*
3. If *True* on diagonal of D^*, a loop exists

is not usable. Even with Warshall's algorithm [Sed92, Wir89], the complexity of this is still $O(n^3)$ and we don't know immediately which nodes are involved in the loop(s).

Tarjan's $O(n+m)$ (n is the number of graph vertices, m is the number of graph edges) Loop Detection algorithm [Baa88] provides an efficient solution:

1. Complete Depth First Search (DFS) on G
   (possibly multiple DFS trees), postorder numbering as for sorting.
2. Reverse edges in the annotated G yielding G_R.
3. Depth First Search on G_R starting with the highest numbered v. The set of vertices in each DFS tree is a strong component. Remove the strong component from G_R and repeat until G_R has been removed completely. In case of absence of loops, the sets of vertices found will all be singletons.

before contains an algebraic loop:

$$\begin{cases} a &= b^2+3 \\ b &= sin(c \times e) \\ c &= \sqrt{d-0.5} \\ d &= \pi/2 \\ e &= a^2+u() \end{cases}$$

In Figure 2.26, the loop detection procedure is shown. G_R is denoted by the α edges.

This results in sorted equations with an isolated algebraic loop which needs to be solved implicitly:

$$\left[\begin{array}{rcl} d & = & \pi/2 \\ c & = & \sqrt{d-0.5} \\ \left\{\begin{array}{rcl} b & = & sin(c\times e) \\ a & = & b^2+3 \\ e & = & a^2+u() \end{array}\right. \end{array}\right] ; \left[\begin{array}{rcl} d & = & \pi/2 \\ c & = & \sqrt{d-4.5} \\ \left\{\begin{array}{rcl} b - sin(c\times e) & = & 0 \\ a - b^2 & = & -3 \\ a^2 - e + u() & = & 0 \end{array}\right. \end{array}\right]$$

Constants, parameters and output equations

In simulation models, it is often meaningful to specify the *variability* of identifiers. Typically, three levels of variability are identified:

1. Constant: the value never changes. Wherever the identifier occurs, it may be replaced by its value. Any variable which is an algebraic function of only constants can also be reduced to a constant. The latter statement can be applied recursively. Substituting all constant values is called *constant propagation* in compiler theory.

2. Parameter: the value is set at the beginning of a simulation but remains constant during a single simulation run. Whether the literal value of the parameter is substituted in equations depends on the simulator implementation (*i.e.*, its ability for symbolic processing at run-time).

3. Variable: the value is set to an "initial condition" at the beginning of a simulation run and may subsequently change over the whole integration domain. Variables occurring in $\frac{dx}{dt}$ form are called *derived state* variables. The equations are solved (integrated) for these variables. All other variables are *algebraic* variables. Different types of algebraic variables can be identified:

   - *input* variables ($u(t)$) can be considered "known" from the point of view of the equation solver as at each point in time, their value is given externally (interpolated from file or from a function generator –an algebraic model in its own right).
   - *output* variables ($y(t)$) are not in any way (via intermediate expressions and variables) needed in the Right-Hand-Side (RHS) of derived state variable equations.
   - algebraic *state* variables are all other algebraic variables. They are needed (as intermediate help variables) to compute the RHSs of $\frac{dx}{dt}$ equations.

Using the dependency graph described before, it is possible to identify constant and parameter equations and place them in the computeInitial() section of an MSL-EXEC model. Similarly, output equations can easily be extracted and placed in the computeOutput() section. computeOutput() is only evaluated when output is requested by a simulation user. computeState() on the other hand is called as frequently as needed for numerical stability and accuracy.

2.5.3 Differential Algebraic Equations: Non-causal Set

Mapping onto DAE Causal Set: causality assignment

From the point of view of lucidity, and re-usability in different causal contexts, a non-causal, implicit set of equations is desirable when modelling physical systems. However, to be able to solve for the various unknowns in the set of equations, it is far more preferable to have a causal representation. It is possible in many cases to transform a non-casual representation into a causal one. Consider the implicit set of equations

$$\left\{\begin{array}{rcll} x+y+z & = & 0 & \text{Equation 1} \\ x+3z+u^2 & = & 0 & \text{Equation 2} \\ z-u-16 & = & 0 & \text{Equation 3} \\ u-5 & = & 0 & \text{Equation 4.} \end{array}\right.$$

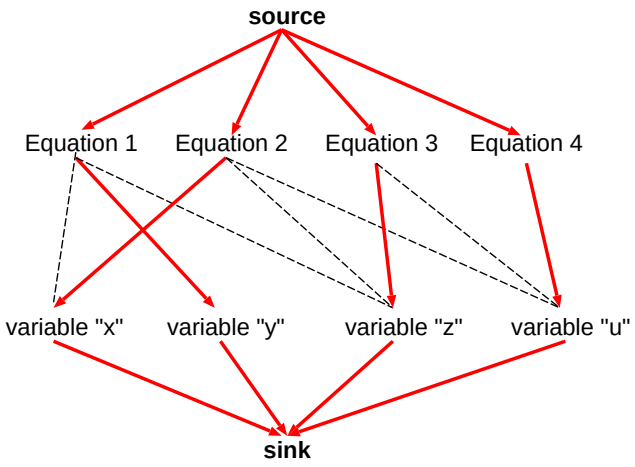

Figure 2.27: Causality assignment: network flow

In order to solve this set of equations on a computer, a matching of variables and equations is required, that is, we must identify which equation can be used to solve for what variable. This can be accomplished by turning equations into nodes and dependencies into edges in a bipartite graph. The problem of matching equations with variables is thus reduced to a *maximum cardinality matching* problem on the bipartite graph.

The problem can then be solved elegantly by turning it into a network flow problem. This is achieved by adding a source and sink node to the bipartite graph resulting in a directed graph as shown in Figure 2.27. All flow capacities of the network are set to 1.

By maximizing the flow from a source to a sink, the causality assignment is carried out. We obtain the correspondence between a variable and the equation used to solve for it:

$$\begin{cases} \underline{y} &= -x-z \\ \underline{x} &= -3z-u^2 \\ \underline{z} &= u+16 \\ \underline{u} &= 5 \end{cases}$$

By assigning *weights* to the equation-variable edges it is possible to take into account preferences for certain causal realisations. In the equation

$$x+y^2-3=0$$

for example, the preferred causality (a term from Bond Graph theory) is to calculate x from y rather than the opposite. The latter is harder to invert than the former [SBS94].

We describe the network flow problem in the following.

In our discussion of network flows, we follow the treatment by Tarjan [Tar83]. Network flow problems are a class of optimization problems with a wide variety of applications. Here, we are concerned with the *maximum flow problem* on a network, and in particular, the algorithm of Dinic [Din70] to find such a maximum flow. We first introduce some important concepts in the theory of network flows, and briefly sketch Ford and Fulkerson's method [FF62] of finding a maximum flow before moving on to Dinic's algorithm.

Flows, cuts and augmenting paths

Let $G=[V,E]$ be a directed graph made up of the set of vertices $V \equiv \{v\}$ and the set of edges $E \equiv \{e\}$. We identify two special vertices, the source s and the sink t. The number of vertices in G is n and the number of edges is m. With every edge we associate a positive *capacity* $cap(v,w)$, and set $cap(v,w)=0$ if $[v,w]$ is not an edge. We then define a *flow* f on G, which is a real-valued function on vertex pairs having the three properties:

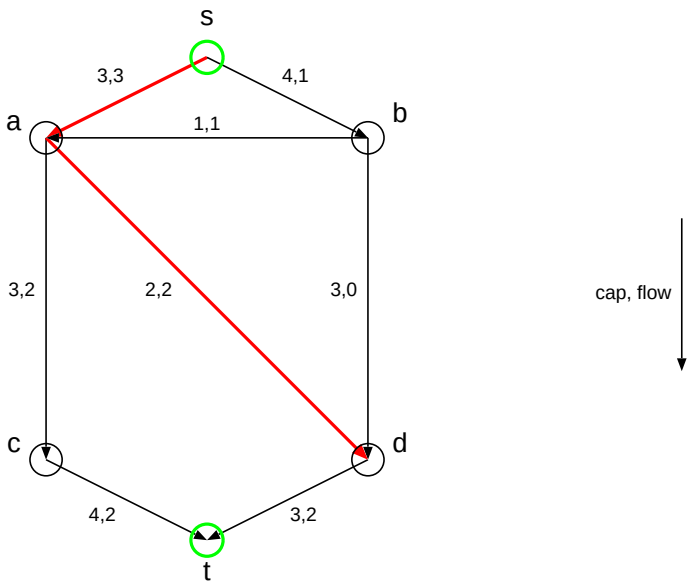

Figure 2.28: Directed graph G with flow f, source s and sink t

- *Skew symmetry*: $f(v,w) = -f(w,v)$.
  Also, if $f(v,w) > 0$, then there is a flow from v to w.
- *Capacity constraint*: $f(v,w) \leq cap(v,w)$.
  A flow is said to *saturate* the edge $[v,w]$ if the equality $f(v,w) = cap(v,w)$ holds.
- *Flow conservation*: for every vertex v excluding the source s and sink t, the net incoming flow must equal the net outgoing flow: $\sum_{w \in V} f(v,w) = 0$.

Figure 2.28 gives an example graph with flow annotations: the first number on an edge is its capacity, the second its flow. The edges $[s,a]$ and $[a,d]$ are saturated. Note how this example is more general than what is needed for bipartite graph matching.

The net flow out of the source, $\sum_{v \in V} f(s,v)$, is called the *value* of the flow f and is denoted by $|f|$. A flow of maximum value is called a *maximum flow*, and thus the *maximum flow problem* is that of finding such a maximum flow. Many attempts have been made to tackle this problem theoretically, and algorithms of increasing speed have been created over the years. See Table 2.1 for a historical overview.

Originally developed by Ford and Fulkerson [FF62], the theory of network flows has its roots in linear programming [Law76]. We review some basic results of Ford and Fulkerson, followed by a discussion of the algorithm of Dinic [Din70].

An important concept in the theory of network flows is that of a *cut*. We define a cut $X, \overline{X}$ to be a partition of the vertex set V into two parts X and $\overline{X} = V - X$, such that X contains the source s and $\overline{X}$ contains the sink t. The *capacity* of a cut $X, \overline{X}$ is

$$cap(X,\overline{X}) = \sum_{v \in X, w \in \overline{X}} cap(v,w).$$

A cut of minimum capacity is known as a *minimum cut*. If f is a flow and $X, \overline{X}$ is a cut, the *flow across the cut* is

$$f(X,\overline{X}) = \sum_{v \in X, w \in \overline{X}} f(v,w).$$

It can be shown that for any flow f, the flow across any cut $X, \overline{X}$ equals the flow value $|f|$. That is,

$$f(X,\overline{X}) = \sum_{v \in X, w \in \overline{X}} f(v,w) = |f|.$$

date	**discoverer(s)**	**running time**
1956	Ford and Fulkerson	——
1969	Edmonds and Karp	$O(nm^2)$
1970	Dinic	$O(n^2 m)$
1974	Karzanov	$O(n^3)$
1978	Malhotra, *et. al.*	$O(n^3)$
1977	Cherkasky	$O(n^2 m^{1/2})$
1978	Galil	$O(n^{5/3} m^{2/3})$
1979	Galil and Naamad; Shiloach	$O(nm(\log n)^2)$
1980	Sleator and Tarjan	$O(nm \log n)$

Table 2.1: History of maximum flow algorithms

By the capacity constraint, the flow across any cut cannot exceed the capacity of the cut. Thus the value of a maximum flow cannot be greater than the capacity of a minimum cut. The "*max-flow min-cut theorem*" states that in fact these two quantities are equal. We shall restate the theorem below after introducing the concepts of *residual capacity* and *augmenting path.*

The *residual capacity* for a flow f in a network is given by a function on vertex pairs, and is the difference in the capacity of the edge connecting the two vertices and the flow across the edge:

$$res(v,w) = cap(v,w) - f(v,w). \tag{2.1}$$

We can push up to $res(v,w)$ additional units of flow from v to w by increasing the flow $f(v,w)$ and correspondingly decreasing $f(w,v)$. We can construct the *residual graph* R for a flow f, which is the graph with vertex set V including the source s and sink t, and an edge $[v,w]$ of capacity $res(v,w)$, such that this capacity is positive: $res(v,w) > 0$. Figure 2.29 shows the residual graph for the flow of Figure 2.28.

An *augmenting path* for f is defined as a path p from s to t in R. The *residual capacity* of this path, denoted by $res(p)$, is the minimum value of $res(v,w)$ for $[v,w]$ an edge of p. The value of the flow f can be increased by any amount Δ up to $res(p)$ by increasing the flow on every edge of p by Δ. We must keep in mind that whenever we change $f(v,w)$ we must change $f(w,v)$ by a corresponding amount to maintain skew symmetry. An augmenting path $[s,b,d,a,c,t]$ for the residual graph in Figure 2.29 is shown in Figure 2.30. If f is any flow, and f^* a maximum flow on a graph G, and if R is the residual graph for f, it can be shown that the value of a maximum flow on R is given by the difference $|f^*|$ - $|f|$. We now restate the max-flow min-cut theorem as follows:

THEOREM: The following conditions are equivalent:

1. f is a maximum flow;
2. there is no augmenting path for f;
3. $|f| = cap(X, \overline{X})$ for some cut $X, \overline{X}$.

The max-flow min-cut theorem above gives a way to construct a maximum flow by *iterative improvement.* This is the *augmenting path method* of Ford and Fulkerson: begin with a flow of zero on all edges in the graph (called the *zero flow*), and repeat the following step until a flow *without* an augmenting path is obtained:

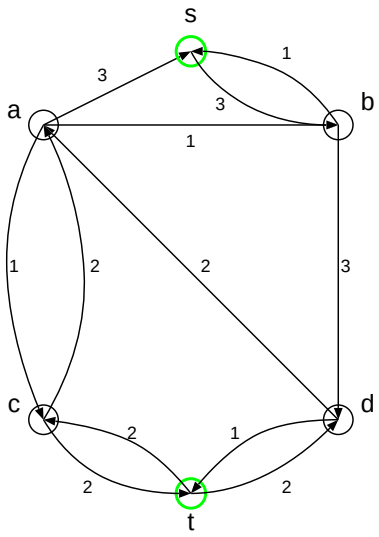

Figure 2.29: Residual graph

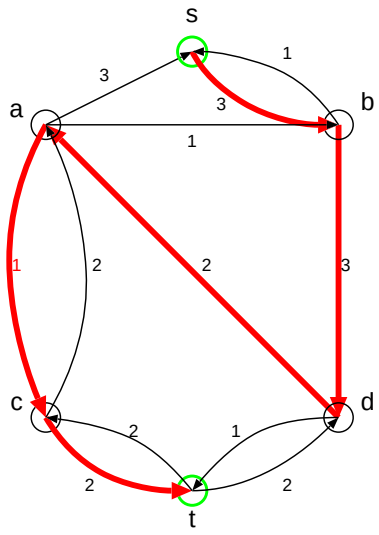

Figure 2.30: Augmenting path

AUGMENTING STEP (Ford and Fulkerson):

1. Find an augmenting path p for the current flow.

2. Increase the value of the flow by pushing $res(p)$ units of flow along p.

Knowing a maximum flow, we can compute a minimum cut in $O(m)$ time. If the edge capacities happen to be integers, the augmenting path method increases the flow value by at least one with each augmentation, and thus computes a maximum flow f^* in at most $|f^*|$ augmenting steps. Also, $f^*(v,w)$ is an integer for every v, w. A flow with such a property is called an *integral flow*. We are led to the "integrality theorem" that if all capacities are integers, there is an integral maximum flow.

However, if the capacities are large integers, the value of a maximum flow may be large, and the augmenting path method may iterate over several augmentations. Indeed, if the capacities are irrational the method may not halt, and although successive flow values converge they need not converge to the value of a maximum flow [FF62]. Thus it is obvious that for the method to be efficient we must select the augmenting paths carefully. It can be proved that starting from the zero flow, there is a way to construct a maximum flow in at most m steps, each of which increases the flow along a single path in the original graph (by an amount that is not necessarily maximum). This involves iterating over a *pathfinding step*, defined below.

Consider a maximum flow f^* on the graph G, and let G^* be the subgraph of G induced by the edges $[v,w]$ such that $f^*(v,w) > 0$. $i = 1$ initially, and the following *pathfinding step* is repeated until the sink t is not reachable from s in G^*:

PATHFINDING STEP:

1. Find a path p_i from s to t in G^*.

2. Let Δ_i be the minimum of $f^*(v,w)$ for $[v,w]$ an edge of p_i. For every edge $[v,w]$ on p_i, decrease $f^*(v,w)$ by Δ_i and delete $[v,w]$ from G^* if its flow is now zero.

3. Increment i by one.

Each pathfinding step deletes at least one edge from G^*; thus this algorithm halts after at most m steps, having reduced f^* to a flow of value zero. However there may still be cycles of flow. Starting with the zero flow and successively pushing Δ_1 units of flow along p_1, Δ_2 units of flow along p_2, ... thus produces a maximum flow in at most m steps.

Edmonds and Karp [EK72] proposed the *maximum capacity augmentation* method as the most natural way to select augmenting paths, wherein augmentation is always carried out along a path of maximum residual capacity. It can be proved that maximum capacity augmentation produces successive flow values that converge to the value of a maximum flow in $O(m \log c)$ augmenting steps, where c is the maximum edge capacity.

Finding a maximum capacity augmenting path is a version of the "bottleneck path problem", if we take lengths equal to the negative of the capacities. Such a path can be found using a suitably modified version of Dijkstra's algorithm. This method takes $O(m \log_{(2+m/n)} n)$ time to find an augmenting path, and the total time to find a maximum flow is $O(m^2 (\log_{(2+m/n)} (\log c))$ if the capacities are integers. This bound is polynomial in n, m and the number of bits needed to represent the capacities, but is still not fully satisfactory. We would like a bound which is polynomial in just n and m.

We can obtain an algorithm with such a bound by choosing augmenting paths by a different method, also proposed by Edmonds and Karp [EK72]: always choose a shortest augmenting path, where we measure the length of a path by the number of edges it contains. This method is most efficient if we augment along paths of the same length simultaneously, as suggested by Dinic [Din70], who independently arrived at his result. We discuss Dinic's method further below.

Dinic's algorithm and augmenting by blocking flows

In order to understand Dinic's algorithm we need two new concepts. A flow f is a *blocking flow* if every path from the source s to the sink t contains a saturated edge. Therefore it is not possible to increase the value of a blocking flow by pushing additional flow along any path in G. It may be possible, however, to increase the flow by rerouting – that is, decreasing the flow on some edges and increasing it on others. Let R be the residual graph for a flow f. The length of the shortest path from s to any vertex v in R is the *level* of v. The *level graph* L for f is the subgraph of R containing only the vertices reachable from s, and only the edges $[v,w]$ such that $level\ (w) = level\ (v) + 1$. L contains every shortest augmenting path and can be constructed in $O(m)$ time by *breadth-first search*.

Dinic's algorithm consists of beginning with the zero flow and repeating the *blocking step* below until the sink t is not in the level graph for the current flow. In Figure 2.31), A: shows the input graph, B: shows the first level graph with blocking flow. Levels of vertices are in parentheses. C: shows the second level graph with blocking flow and D: the third level graph with blocking flow. E: is the final flow. A minimum cut is {s,a,b,d}, {c,t}.

BLOCKING STEP (Dinic):

1. Find a blocking flow f' on the level graph for the current flow f.
2. Replace f by the flow $f + f'$ defined by:

$$(f + f')(v,w) = f(v,w) + f'(v,w). \tag{2.2}$$

The performance of Dinic's algorithm is governed by the theorem that the algorithm halts after at most $n-1$ blocking steps.

In the next section we describe Dinic's method of finding a blocking flow.

Finding blocking flows

Let G be an acyclic network on which it is required to find a blocking flow. There are different ways to find such a flow, each leading to a maximum flow algorithm. However the simplest way to find a blocking flow is Dinic's method: we find a path from the source s to the sink t, push enough flow along it to saturate an edge, delete all newly saturated edges, and repeat this procedure until t is not reachable from s. We use depth-first search to find each path. The method is defined more formally below. We begin with the zero flow, go to *Initialize*, and proceed as indicated. p is a path along which flow can be pushed from s to the current vertex v:

- *Initialize*: Let $p = [s]$ and $v = s$. Go to *Advance*.
- *Advance*: If there is no edge out of v, go to *Retreat*. Otherwise, let $[v,w]$ be an edge out of v. Replace p by $p\&[w]$ and v by w. If $w \neq t$ repeat *Advance*; if $w = t$ go to *Augment*.
- *Augment*: Let Δ be the minimum of $(cap(v,w) - f(v,w))$ for $[v,w]$ an edge of p. Add Δ to the flow of every edge on p, delete from G all newly saturated edges, and go to *Initialize*.
- *Retreat*: If $v = s$ halt. Otherwise, let $[u,v]$ be the last edge on p. Delete v from p and $[u,v]$ from G, replace v by u, and go to *Advance*.

It can be proved that Dinic's algorithm above correctly finds a blocking flow in $O(nm)$ time, and a maximum flow in $O(n^2 m)$ time. It can also be proved that on a unit network, Dinic's algorithm finds a blocking flow in $O(m)$ time, and a maximum flow in $O(n^{1/2} m)$ time. In a unit network, all edge capacities are integers, and each vertex v other than the source and the sink has either a single entering edge of capacity one, or a single outgoing edge of capacity one. On a network whose edge capacities are all one, Dinic's algorithm finds a maximum flow in $O(\min\{n^{2/3} m, m^{3/2}\})$ time [ET75]. This is exactly the case we are confronted with.

On general networks Dinic's blocking flow method saturates only one edge at a time in the worst case, taking up $O(n)$ time for each edge saturated. On dense graphs there are faster methods that, in effect, saturate one vertex at a time and have an $O(n^2)$ running time.

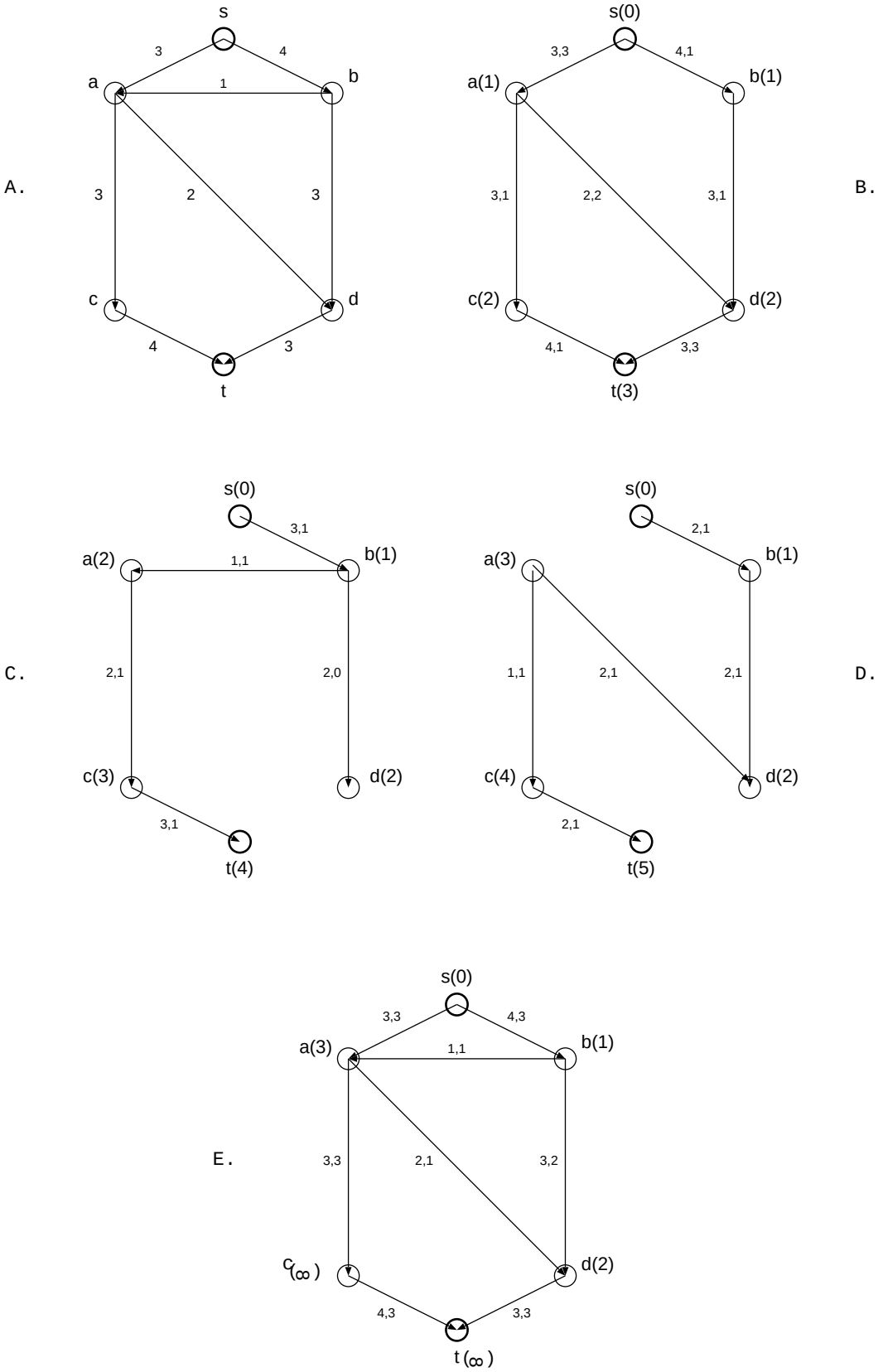

Figure 2.31: Dinic's maximum flow algorithm applied

How does Dinic's algorithm compare to other approaches ? Elmquist in his PhD thesis [Elm78] was the first to apply causality assignment in what is currently the Dymola tool (`www.dynasim.com`). Based (probably, not explicitly mentioned) on the Kuhn-Munkres algorithm. More recently, in his PhD thesis, Sahlin [Sah96] also uses the Kuhn-Munkres algorithm (also known as the Hungarian method) from operations research. In his PhD thesis, Broenink [Bro90] uses a causality assignment procedure specific to Bond Graphs. The approach we have presented cleanly maps the problem onto a graph problem. Furthermore, we claim Dinic's is the most efficient algorithm for causality assignment problems. For any n (corresponding to the number of variables and equations in the original set of equations), its performance ($O(\min\{n^{2/3}m, m^{3/2}\})$) exceeds that of the Hungarian algorithm ($O(nmlog_{(2+m/n)}n)$). In Table 2.1 we find more efficient algorithms. These do use more elaborate data structures however which makes practical implementation more cumbersome. Furthermore, Dinic's algorithm is most efficient on sparse networks and when edge capacities are all one. For large n the performance even exceeds that of the recent Sleator and Tarjan algorithm ($O(nm\log n)$). This, as

$$n^{2/3}m < nm\log n$$

for large n.

Once the dependency graph has been built, network flow analysis, cycle detection, sorting, as well as isolation of constant/parameter and output expressions are elegantly performed. In particular, the network flow graph can be re-used for sorting and cycle detection of the equation nodes are ignored.

2.6 The Transfer Function formalism

The Transfer Function formalism is commonly used in control engineering. Transfer Function model representations are the basis for a plethora of stability analysis and controller design techniques. In the following, we show how one may transform at will between state-space and Transfer Function models. This transformation was automated using the MuPAD [F$^+$96] computer algebra tool.

2.6.1 Formalism representations and transformations

The state-space mathematical representation of a linear system is

$$\begin{cases} \frac{dx}{dt} = Ax + Bu \\ y = Cx + Du \end{cases}$$

Starting from the matrices A, B, C, D, by Laplace rules, we obtain the transfer function representation of the system (I is the identity matrix):

$$H(s) \quad = \quad C(sI - A)^{-1}B + D.$$

We notice that for a MIMO (Multiple Input, Multiple Output) system, $H(s)$ is a rational matrix of dimension $p \times m$. Let us consider an element $h(s) = H[i, j]$; $h(s)$ is the transfer function between the j-th input and the i-th output. The external stability (*i.e.*, I/O stability) of the open loop linearized model can be checked by finding the poles of the rational function $h(s)$ through factorization. Once poles and zeros of $h(s)$ are available, we can write the *pole-zero* form:

$$h(s) \quad = \quad k\frac{(s - z_1)\ldots(s - z_m)}{(s - p_1)\ldots(s - p_n)}.$$

$H(s)$ in the above form implies that, with the Laplace transformation of the input $U(s)$, the Laplace transformation of the output can be obtained by means of a simple multiplication:

$$Y(s) = H(s)U(s).$$

From $h(s)$ we can also obtain the frequency response of the system:

$$H(\omega) = H(s = j\omega).$$

If a transfer function of a linear SISO (Single Input, Single Output) model

$$h(s) = \frac{b_r s^r + b_{r-1} s^{r-1} + \cdots + b_1 s + b_0}{a_n s^n + a_{n-1} s^{n-1} + \cdots + a_1 s + a_0}$$

(with $n \geq r$) is available, it can be useful to get a state space representation of the system. This is called a "realization" of h(s). We point out that this transformation is not unique. We choose the *canonical reachable* realization:

$$\begin{cases} \frac{dx}{dt} = Ax + Bu \\ y = Cx + Du \end{cases}$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 \\ -\tilde{a}_0 & -\tilde{a}_1 & \cdots & \cdots & -\tilde{a}_{n-1} \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$$C = \begin{bmatrix} \hat{b}_0 & \hat{b}_1 & \cdots & \hat{b}_{n-1} \end{bmatrix}$$

$$D = \begin{bmatrix} \tilde{b}_n \end{bmatrix}$$

with $\tilde{a}_i = a_i/a_n$, $\tilde{b}_i = b_i/a_n$, $\hat{b}_i = \tilde{b}_i - \tilde{b}_n \tilde{a}_i$.
We observe that $[b_n \ldots b_{r+1}] = [0 \ldots 0]$.

2.6.2 Example

If we consider the following transfer function

$$h(s) = \frac{6.01s + 1.18}{0.64s^2 + 1.8s + 0.36},$$

the gain $h(0)$ is 3.28 and we can rewrite $h(s)$ in the pole-zero form

$$h(s) = 9.39 \frac{s + 0.2}{(s + 0.22)(s + 2.6)}.$$

The canonical reachable realization of h(s) is

$$\begin{cases} \frac{dx}{dt} = Ax + Bu \\ y = Cx + Du \end{cases}$$

where

$$A = \begin{bmatrix} 0 & 1 \\ -0.5625 & -2.8125 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

$$C = \begin{bmatrix} 1.84375 & 9.3906 \end{bmatrix}, \quad D = \begin{bmatrix} 0 \end{bmatrix}.$$

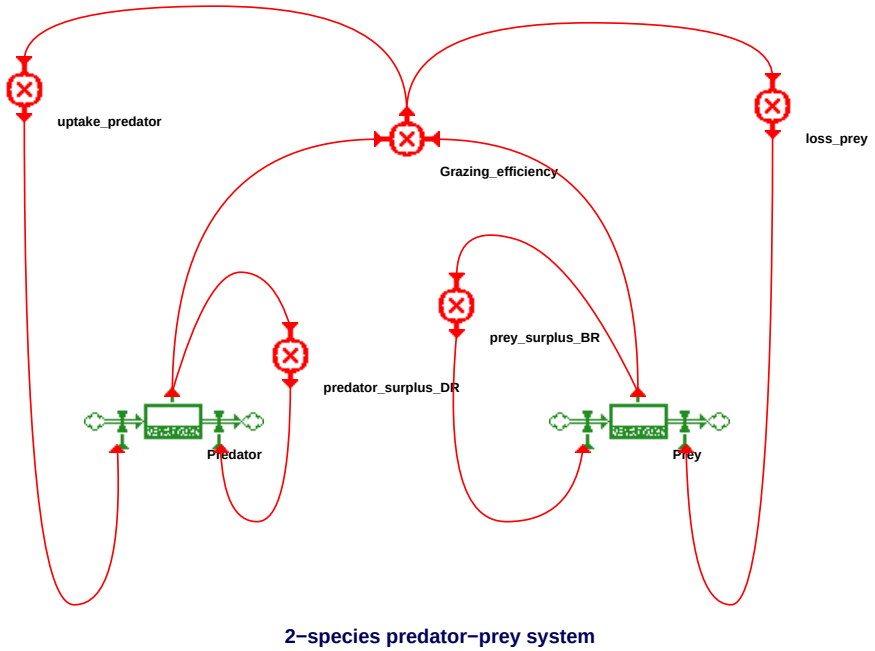

Figure 2.32: Predator-prey, System Dynamics Formalism

2.6.3 Flattening networks of Transfer Function models

As with state-space models (Linear Ordinary Differential and Algebraic Equation formalism), building blocks may be combined to form a network. Actually, the Transfer Function building blocks contain algebraic equations (in s). Thus, coupling semantics is the same as for state-space models: connections are replaced by algebraic equalities.

2.7 The System Dynamics formalism

The Forrester System Dynamics [For68] formalism is a graphical formalism, based on the differential equation formalism. The formalism is particularly geared toward expressing the dynamics of populations. It is used in diverse field such as economics [For61], biology [Cel91], software engineering, and urban planning. The Forrester System Dynamics formalism describes the variation of material like quantities (MLQs) or levels. The variation is determined by birth rates (*BR*) and death rates (*DR*). *BR* and *DR* are graphically represented as valves to the left and right respectively of boxes denoting the levels. Levels may influence each other by influencing each other's *BR* and *DR*. Figure 2.32 shows a typical interaction between a predator and a prey (modelled using the WEST++ modelling and simulation environment [VCV98]). The (product) interaction between predator and prey populations influences the predator's birth rate and the prey's death rate.

The System Dynamics semantics is given by mapping each of the level/*BR*/*DR* combinations to an Ordinary Differential Equation

$$\frac{d\ level}{dt} = BR - DR.$$

The operations such as product and sum are mapped onto the appropriate algebraic equations and couplings are mapped onto algebraic equalities. Simulation in WEST++ of the model in Figure 2.32 produces the trajectories in Figure 2.33. The system, described by Lotke-Volterra equations exhibits an oscillating, stable

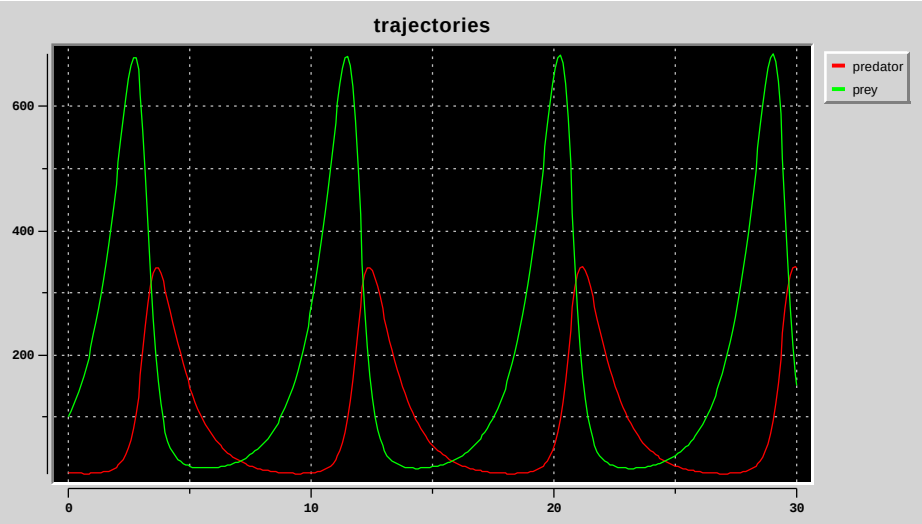

Figure 2.33: Predator-prey system behaviour

behaviour.

2.7.1 Formalism mapping through representation in MSL-USER

Transformation between the System Dynamics formalism and the ODE formalism may be done by constructing appropriate System Dynamics building blocks in the MSL-USER language. For each System Dynamics block, a class is constructed containing the appropriate differential and algebraic equations. The introduction of interfaces (ports) makes coupling possible. The coupling semantics of System Dynamics, substituting connections by algebraic equalities, is identical to the coupling semantics of algebraic and differential equation models and is thus taken care of by the MSL-USER compiler within the DAE formalism. This is shown below for a few relevant building blocks.

```
// Use generic model base
#include "generic.msl"

// don't specify quantity, unit, ...
CLASS SDTerminal SPECIALISES PhysicalQuantityType;

CLASS ConstantClass
  (* class = "constant"; category = "" *)
  "Constant:
   Produces at its output 'out', the value of the parameter 'c'"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
      OBJ out (* terminal = "out" *) "out" :
        SDTerminal := {: causality <- "COUT" :};
    };
    parameters <-
    {
      OBJ c "c" : SDTerminal := {: value <- 0; :};
    };
```

```
    independent <-
    {
      OBJ t "t": Time;
    };
    equations <-
    {
      interface.out = parameters.c;
    };
  :};

CLASS PopulationClass
  (* class = "levelNrate"; category = "" *)
  "System Dynamics Population:
   Produces at its output 'level', the solution
   of the differential equation
    d level/d t = birth_rate - death_rate"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
      OBJ birth_rate (* terminal = "birth_rate" *) "birth_rate" :
        SDTerminal := {: causality <- "CIN"; value <- 0 :};
      OBJ death_rate (* terminal = "death_rate" *) "death_rate" :
        SDTerminal := {: causality <- "CIN"; value <- 0 :};
      OBJ level (* terminal = "level" *) "population level" :
        SDTerminal := {: causality <- "COUT" :};
    };
    independent <-
    {
      OBJ t "t": Time;
    };
    state <-
    {
      OBJ population "population level": SDTerminal;
      OBJ pop_change_rate "population level change rate": SDTerminal;
    };
    equations <-
    {
      DERIV(state.population, [independent.t]) = state.pop_change_rate;
      state.pop_change_rate = interface.birth_rate - interface.death_rate;
      interface.level = state.population;
    };
  :};
#endif
```

To ensure correct use of System Dynamics blocks in a graphical modelling environment, *constraints* must be imposed. In WEST++ this is done in the form of a Tcl script of which relevant parts are shown below. In this case, only constraints on the allowed number of connections at a terminal are given.

```
#*
#* Description: Hierarchical Graph Editor / Library /
#*              System Dynamics.
#*
#* In this library the following needs to be defined:
#*
#*   Variables:
#*
#*     HGE_colors(<type>)
#*       Colors for all types used in the node and terminal classes.
#*     HGE_bitmap_library_path
#*       Path of the library of bitmaps used in the node and terminal classes.
#*     HGE_node_classes
```

```
#*       List of node classes. Each node is described by a list which consists
#*       of a node name, a description, a bitmap name, a type and a list of
#*       terminals. Each terminal is again a list which consists of a terminal
#*       name, a type, a terminal class, an integer indicating how many edges
#*       can be connected to the terminal, the orientation of the bitmap and
#*       the coordinates with respect to the node's hotspot.
#*     HGE_terminal_classes(<type>,<orientation>)
#*       Bitmap names for all type-orientation pairs used in the node classes.

#* Includes ************************************************************

source "$westpp_data_path/me/HGE/generic/generic.HGE.lib.tcl";

#*** Colors ************************************************************

set HGE_colors(generic) blue;
set HGE_colors(physical) forestgreen;
set HGE_colors(data) red;

#*** Node classes *****************************************************

set HGE_node_classes \
{ \
  { \
    constant \
    "Constant" \
    constant \
    data \
    { \
      { out data output 100 right 17 0 } \
    } \
  } \
  { \
    levelNrate \
    "Population determined by
     BirthRate and DeathRate" \
    levelNrate \
    physical \
    { \
      { birth_rate data input 1 bottom -34 16 } \
      { death_rate data input 1 bottom 28 16 } \
      { level data output 100 top -1 -18 } \
    } \
  } \
  { \
    product \
    "Product of two Inputs
     and a Parameter" \
    product \
    data \
    { \
      { out data output 100 top 0 -17 } \
      { in_1 data input 1 left -17 0 } \
      { in_2 data input 1 right 17 0 } \
    } \
  } \
};
#*** Terminal classes *************************************************

set HGE_terminal_classes(generic,left) square;
set HGE_terminal_classes(generic,right) square;
set HGE_terminal_classes(generic,top) square;
set HGE_terminal_classes(generic,bottom) square;
```

```
set HGE_terminal_classes(input,left) right_arrow;
set HGE_terminal_classes(input,right) left_arrow;
set HGE_terminal_classes(input,top) down_arrow;
set HGE_terminal_classes(input,bottom) up_arrow;
set HGE_terminal_classes(output,left) left_arrow;
set HGE_terminal_classes(output,right) right_arrow;
set HGE_terminal_classes(output,top) up_arrow;
set HGE_terminal_classes(output,bottom) down_arrow;
set HGE_terminal_classes(bidirectional,left) circle;
set HGE_terminal_classes(bidirectional,right) circle;
set HGE_terminal_classes(bidirectional,top) circle;
set HGE_terminal_classes(bidirectional,bottom) circle;

#**********************************************************************
```

This completely specifies the graphical representation and behaviour of System Dynamics models in the WEST++ Hierarchical Graphical Editor (see also next chapter).

2.8 The Partial Differential Equations formalism

Partial Differential Equations (PDEs) increase the number of independent variables (one, actually) appearing in Ordinary Differential Equations. Often, these variables concern spatial distribution, hence the name "distributed parameter systems" as opposed to "lumped parameter systems". Numerical techniques exist to solve classes of PDEs (usually divided into parabolic, hyperbolic and elliptic [Far93]). Corresponding numerical codes have been developed and are highly succesful. The approach presented here is to *symbolically* distrectize a class of PDEs. Discretization of a Partial Differential Equation results in a set of Differential Algebraic Equations which may in some cases be represented in explicit form as Ordinary Differential Equations and Algebraic Equations. Symbolic discretization was chosen for the following reasons:

1. Once in DAE form, the symbolic transformations described before may be invoked. This can result in a drastic reduction of model complexity (and consequently in simulation time).

2. The numerical techniques available to solve DAEs and above all ODEs are more powerful than those for solving PDEs. In particular, higher performance, better error control and better stability can be achieved.

3. Once transformed to DAE or ODE form, a PDE model can be perfectly integrated with DAE and ODE models thanks to closure under coupling of these formalisms. Global optimizations can be carried out taking into account the whole system of equations. In the case of the WEST++ tool, PDE to DAE transformation makes it possible to integrate PDE models seamlessly without modifying the basic simulator. This demonstrates the power of formalism transformation for multi-formalism modelling.

In WEST++, the symbolic discretization is implemented as a model compiler. Its working is described in the last chapter.

2.8.1 Introduction

In this section we discuss how to transform the PDE used to model a 1-D clarifier [VVKR97] (which forms part of a waste water treatment plant) into a differential algebraic equation (DAE). We use the PDE to look at two test case conditions for the clarifier: continuous and batch sedimentation [A.V98a].

We use z to represent the coordinate along the length of the 1-D clarifier. The top and bottom ends of the clarifier are located at $z = 0$ and $z = L$ respectively. As discussed in detail in [A.V98a], the PDEs for both test cases have the following general form:

$$\frac{\partial X(z,t)}{\partial t} = -\frac{\partial F(X(z,t))}{\partial z} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}. \tag{2.3}$$

Here $X(z,t)$ is the space- and time- dependent concentration of suspended solids. $F(X(z,t))$ is the total convective flux, which is different for the two test cases, and will be discussed further below.

The required DAE for each test case is obtained from the PDE above by discretizing all the functions of z, and the partial derivatives w.r.t z, but retaining the continuous forms of the time derivative and functions of time. Thus the spatial derivatives are replaced by algebraic expressions, and the PDE therefore becomes an ordinary differential equation in time. We implement the spatial discretization using the method of orthogonal collocation on finite elements [Oh95, MV72, SJ85].

2.8.2 The Orthogonal Collocation method

In the following we give a brief summary of the salient features of the orthogonal collocation method. A detailed description can be found in [Oh95, GSMC85]. This method is one of the *weighted residual methods* used to solve PDEs. Consider a function $\phi(z,t)$ defined over a certain spatial domain Ω, which depends on the two independent variables (z,t). Let the space- and- time- evolution of $\phi(z,t)$ be governed by a PDE :

$$\mathcal{D}\,\phi(z,t) = 0, \tag{2.4}$$

where $\mathcal{D}$ is a differential operator. The PDE is supplemented by boundary conditions for $\phi(z,t)$ on the boundary of the domain Ω, and initial conditions giving the value of $\phi(z,t)$ at time $t = 0$.

In a weighted residual method, we seek an approximate solution $\tilde{\phi}(z,t)$ of Eq.(2.4), which can be expressed as a sum of known polynomial functions depending only on z, with unknown coefficients depending only on t:

$$\tilde{\phi}(z,t) = \sum_{k=0}^{n} a_k(t)\,\theta_k(z). \tag{2.5}$$

The functions $\{\theta_k(z)\}$ are known, and the coefficients $\{a_k(t)\}$ have to be determined so that the solution satisfies the PDE and the boundary and initial conditions in an 'optimal' sense [Oh95].

We first form the *residual* of $\mathcal{D}$, $R_n(z,t)$. It is given by:

$$R_n(z,t) = \mathcal{D}\,\tilde{\phi}(z,t). \tag{2.6}$$

$R_n(z,t)$ will be a continuous function of z and t. The approximate solution is then found by demanding that the *average* value of $R_n(z,t)$ (over the domain Ω) should be zero. We perform this average using some weighting function $W(z)$. That is:

$$\int_{\Omega} W(z)\,R_n(z,t)\,dz = 0. \tag{2.7}$$

We can evaluate the average instead by using a discrete set of m points:

$$\sum_{i=1}^{m} W(z_i)\,R_n(z_i,t) = 0. \tag{2.8}$$

Now in the *collocation method*, we simply choose the weights $\{W(z_i)\}$ to be Dirac delta functions at a particular set of points $\{z_j\}$, $j = 1..m$, called *collocation points*. From Eq.(2.8) above we have:

$$\begin{aligned} \textstyle\sum_i^m \delta(z_i - z_j)\,R_n(z_i,t) &= 0, \quad j = 1,\ldots,m; \\ \Rightarrow R_n(z_j,t) &= 0, \quad j = 1,\ldots,m. \end{aligned} \tag{2.9}$$

In the *orthogonal collocation method*, which is usually employed, the collocation points $\{z_j\}$ are chosen as the roots (zeroes) of one member of a set of orthogonal polynomials, usually the *Jacobi polynomials* [AS65, AW95]. Let the domain of interest be given by the interval $[z_0, z_{n+1}]$. Using the n zeroes of a Jacobi polynomial $G_n(\alpha+1,\beta+1,z)$ located in $[z_0, z_{n+1}]$, together with the two end-points of the domain - z_0 and

z_{n+1}, the approximate solution is constructed using *Lagrange interpolation* [AS65, AW95]. (See further subsections below for notes on Jacobi polynomials and Lagrange interpolation.) That is,

$$\tilde{\phi}(z,t) = \sum_{k=0}^{n} \phi(z_k,t)\,\ell_k(z)\,. \tag{2.10}$$

Comparing Eqs.(2.5) and (2.10) above, we note that the unknown coefficients $\{a_k(t)\}$ are just the values of the solution ϕ evaluated at the collocation points (the zeroes of the Jacobi polynomial and the end-points), ($\{\phi(z_k,t)\}$, $k = 0,\ldots,n+1$), and the polynomials $\{\theta_k(z)\}$ are just *Lagrange polynomials.*

A weighted residual method such as the orthogonal collocation method described above can be applied over the entire spatial domain Ω, and increased accuracy can be obtained by increasing the order n of the polynomials used. However, a better strategy is to use a *finite element* method. In such a method, the domain of interest is first divided into many smaller sub-domains or 'elements'. Then a weighted residual method is applied within each element. Usually low-order approximations of the residuals are sufficient. Additional constraints are imposed by demanding the continuity of the approximate solution, and one or more of its derivatives, (or that of other quantities), at the boundaries between elements [Oh95].

Lagrange Interpolation

For simplicity, we shall consider Lagrange interpolation in one dimension [Oh95]. The method can be easily extended to multiple dimensions. Consider a set of points $\{x_0,x_1,x_2,\ldots,x_n\}$ located on the X axis. Suppose data corresponding to a certain quantity f are provided at these points; that is, we know the values $\{f(x_i)\}$ at these $n+1$ points. Using these values, we must find an approximation for the function $f(x)$ over the interval $[x_0,x_n]$. The simplest thing to do is to approximate $f(x)$ by a polynomial curve passing through the values $\{f(x_i)\}$. This can be done easily using the *Lagrange interpolation formula* to construct the interpolating polynomial using the $\{x_i\}$ as 'nodes'. We thus have:

$$f(x) \simeq \sum_{i=0}^{n} \ell_i(x)\, f(x_i). \tag{2.11}$$

Here $\{\ell_i(x)\}$ are the *Lagrange polynomials*, of degree n (since we know $(n+1)$ constants, we can determine a polynomial of degree n). These polynomials are orthonormal. That is,

$$\ell_i(x_j) = \delta_{ij}\,. \tag{2.12}$$

As an example, we shall find the interpolating polynomial $f(x)$, given the two nodes $\{x_0,x_1\}$, and the values of $f(x)$ at these points. In this case, $f(x)$ will be a straight line connecting the two points. Using the interpolation formula (Eq.(2.11)), we have:

$$f(x) = \ell_0(x)\, f(x_0) + \ell_1(x)\, f(x_1). \tag{2.13}$$

For any x located between x_0 and x_1, we can use the property of a straight line that it has constant slope:

$$\frac{f(x_1)-f(x)}{(x_1-x)} = \frac{f(x_1)-f(x_0)}{(x_1-x_0)}. \tag{2.14}$$

Rearranging terms gives us the Lagrange interpolation formula:

$$f(x) = \frac{(x-x_1)}{(x_0-x_1)}\, f(x_0) + \frac{(x-x_0)}{(x_1-x_0)}\, f(x_1). \tag{2.15}$$

From the above equation, we see that the Lagrange polynomials are given by:

$$\begin{aligned} \ell_0(x) &= \frac{(x-x_1)}{(x_0-x_1)} \\ \ell_1(x) &= \frac{(x-x_0)}{(x_1-x_0)} \end{aligned} \tag{2.16}$$

It can be verified that these are orthonormal.

Extending the discussion above to $(n+1)$ points, we can write a general expression for the Lagrange polynomials as follows:

$$\ell_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(x-x_j)}{(x_i-x_j)}. \tag{2.17}$$

We can rewrite $\ell_i(x)$ in a different form by first defining a polynomial $p(x)$ as:

$$\begin{aligned} p(x) &= \prod_{j=0}^{n}(x-x_j); \\ p^{(1)}(x_j) &= \left.\frac{dp}{dx}\right|_{x=x_j}. \end{aligned} \tag{2.18}$$

Therefore,

$$\ell_i(x) = \frac{p(x)}{(x-x_i)\,p^{(1)}(x_j)}. \tag{2.19}$$

By rearranging Eq.(2.19) and after some algebra, we can obtain explicit expressions for the first and second derivatives $\ell_i^{(1)}(x)$ and $\ell_i^{(2)}(x)$ [MV72], evaluated at a particular point $x = x_j$, in terms of $p(x)$ and its derivatives. There are two cases to consider: $i = j$, and $i \neq j$. We thus have [MV72]:

1. $i = j$:

We obtain a general expression for the m-th derivative of $\ell_i(x)$, $\ell_i^{(m)}(x)$ (evaluated at x_j) only in terms of $p(x)$ and its derivatives, when $i = j$:

$$\ell_i^{(m)}(x_i) = \frac{1}{m+1}\,\frac{p^{(m+1)}(x_i)}{p^{(1)}(x_i)}. \tag{2.20}$$

The first and second derivatives are therefore given by:

$$\begin{aligned} \ell_i^{(1)}(x_i) &= \frac{1}{2}\,\frac{p^{(2)}(x_i)}{p^{(1)}(x_i)}; \\ \ell_i^{(2)}(x_i) &= \frac{1}{3}\,\frac{p^{(3)}(x_i)}{p^{(1)}(x_i)}. \end{aligned} \tag{2.21}$$

2. $i \neq j$:

In this case, it is not possible to obtain a general expression for $\ell_i^{(m)}(x)$ in terms of only $p(x)$ and its derivatives. We arrive at the following expressions for the first and second derivatives:

$$\ell_i^{(1)}(x_j) = \frac{1}{(x_j-x_i)}\,\frac{p^{(1)}(x_j)}{p^{(1)}(x_i)}; \tag{2.22}$$

and

$$\begin{aligned} \ell_i^{(2)}(x_j) &= \frac{1}{(x_j-x_i)}\left(\frac{p^{(2)}(x_j)}{p^{(1)}(x_i)} - 2\,\ell_i^{(1)}(x_j)\right) \\ &= 2\,\ell_i^{(1)}(x_j)\left(\ell_j^{(1)}(x_j) - \frac{1}{(x_j-x_i)}\right). \end{aligned} \tag{2.23}$$

Jacobi Polynomials

There are two kinds of Jacobi polynomials [AS65, AW95], $P_n(\alpha,\beta,x)$ and $G_n(\alpha+\beta+1,\beta+1,x)$, where $(\alpha,\beta>-1)$. The polynomials $P_n(\alpha,\beta,x)$ form a complete orthogonal system in the interval $[-1,1]$, while $G_n(\alpha+\beta+1,\beta+1,x)$ are orthogonal over $[0,1]$. We shall be concerned only with the latter. They are defined by:

$$G_n(\alpha+\beta+1,\beta+1,x)=\frac{\Gamma(\beta+1+n)}{\Gamma(\alpha+\beta+1+2n)}\sum_{m=0}^{n}(-1)^m\left(\begin{array}{c}n\\m\end{array}\right)\frac{\Gamma(\alpha+\beta+1+2n-m)}{\Gamma(\beta+1+n-m)}x^{n-m}. \tag{2.24}$$

They are orthogonal w.r.t the weight function:

$$w(x)=(1-x)^{\alpha}x^{\beta}. \tag{2.25}$$

That is,

$$\int_0^1 w(x)\,G_n(\alpha+\beta+1,\beta+1,x)\,G_m(\alpha+\beta+1,\beta+1,x)\,dx=C_n\,\delta_{nm}. \tag{2.26}$$

The normalization constant C_n is given by:

$$C_n=\frac{n!\,\Gamma(n+\beta+1)\,\Gamma(n+\alpha+1)\,\Gamma(n+\alpha+\beta+1)}{(2n+\alpha+\beta+1)\,\Gamma^2(2n+\alpha+\beta+1)}. \tag{2.27}$$

For a given pair (α,β), let us write

$$G_n(\alpha+\beta+1,\beta+1,x)\equiv G_n(x). \tag{2.28}$$

The $\{G_n(x)\}$ satisfy the recurrence relation:

$$G_{n+1}(x)=(x-a_n)\,G_n(x)-b_n\,G_{n-1}(x), \tag{2.29}$$

for $n=0,1,2\ldots$. The coefficients are given by:

$$\begin{aligned}a_n &= \frac{2n(n+\alpha+\beta+1)+(\beta+1)(\alpha+\beta)}{(2n+\alpha+\beta)(2n+\alpha+\beta+2)}\\ b_n &= \frac{n(n+\alpha)(n+\beta)(n+\alpha+\beta)}{(2n+\alpha+\beta-1)(2n+\alpha+\beta)^2(2n+\alpha+\beta+1)}.\end{aligned} \tag{2.30}$$

Also, $G_0(x)=1$, and by definition, $G_{-1}(x)=0$. We can compute other polynomials for higher values of n by simply using the recurrence relation successively. The values of α and β determine the location of the roots of the Jacobi polynomial within the interval $[0,1]$ [SJ85, Kop97]. Setting $(\alpha=\beta=0)$ is the most common choice.

Zeroes of Jacobi Polynomials

In order to implement the collocation procedure, we have to find the n zeroes of a Jacobi polynomial $G_n(x)$ numerically. We proceed in the following way. We first write out the recurrence relation in Eq.(2.29) explicitly for the first few values of n:

$$\begin{aligned}G_1(x) &= (x-a_0)\,G_0(x)-b_0\,G_{-1}(x)\\ &= (x-a_0).1-0\\ &= (x-a_0);\\ \\ G_2(x) &= (x-a_1)\,G_1(x)-b_1\,G_0(x)\\ &= (x-a_1)\,(x-a_0)\,G_0(x)-b_1\,G_0(x)\\ &= (x-a_1)\,(x-a_0).1-b_1.1\\ &= (x-a_1)\,(x-a_0);\\ \\ G_3(x) &= (x-a_2)\,G_2(x)-b_2\,G_1(x)\\ &= (x-a_2)\,[\,(x-a_1)\,(x-a_0)\,G_0(x)-b_1\,G_0(x)\,]-b_2\,(x-a_0)\\ &= (x-a_2)\,[\,(x-a_1)\,(x-a_0)-b_1\,]-b_2\,(x-a_0),\end{aligned} \tag{2.31}$$

and so on for $n = 4, 5, \ldots$. From the above expressions, we notice that the polynomials $G_n(x)$ can be written as the *determinants* of corresponding square matrices of order *n* which have the form:

$$\begin{aligned} G_1(x) &= det\,[\,(x-a_0)\,]\,; \\ \\ G_2(x) &= det\begin{bmatrix} (x-a_0) & -\sqrt{b_1} \\ -\sqrt{b_1} & (x-a_1) \end{bmatrix}; \\ \\ G_3(x) &= det\begin{bmatrix} (x-a_0) & -\sqrt{b_1} & 0 \\ -\sqrt{b_1} & (x-a_1) & -\sqrt{b_2} \\ 0 & -\sqrt{b_2} & (x-a_2) \end{bmatrix}. \end{aligned} \tag{2.32}$$

Finding the zeroes of $\{G_1(x), G_2(x), G_3(x), \ldots\}$, is just equivalent to setting the determinants above to zero. However, notice that doing this yields the characteristic (or eigenvalue) equations for the symmetric tridiagonal matrices whose diagonals have the coefficients $\{a_i\}$ and the subdiagonals are made up of the coefficients $\{\sqrt{b_i}\}$. That is, for a given *n*, the zeroes of $G_n(x)$ are given by the eigenvalues of a symmetric tridiagonal matrix of dimension $[\,n \times n\,]$, whose diagonal is the *n* - dimensional vector *Diag*:

$$Diag = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}, \tag{2.33}$$

and the subdiagonals are given by the $(n-1)$ - dimensional vector *Sub*:

$$Sub = \begin{bmatrix} \sqrt{b_1} \\ \sqrt{b_2} \\ \vdots \\ \sqrt{b_{n-1}} \end{bmatrix}. \tag{2.34}$$

2.8.3 The PDE for continuous sedimentation

For continuous sedimentation, the PDE, Eq.(2.3), is given by:

$$\frac{\partial X(z,t)}{\partial t} = -\frac{\partial F(X(z,t))}{\partial z} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}. \tag{2.35}$$

where the convective flux F is given by the sum of the gravitational settling flux and the bulk flow. We also consider two different cases which determine the form of the bulk flows: if the effluent is drawn out from the top of the clarifier (using a pump), then the effluent flow appears explicitly in the bulk flow. If there is no pump, then the effluent is an overflow, and is a consequence of the other flows. We consider these two cases separately.

Effluent pumped out

We write the convective flux in this case in the following form:

$$F(X(z,t),z,t) = G(X(z,t))$$

$$+X(z,t)\left[\frac{Q_u(t)}{A}\theta(z-z_f)-\frac{Q_e(t)}{A}\theta(z_f-z)\right]$$
$$-X_f(t)\,\frac{Q_f(t)}{A}\,\theta(z-z_f)\,. \tag{2.36}$$

Here, $G(X(z,t))$ is the gravitational settling flux, and is given by

$$G(X(z,t)) = X(z,t)\,v_s(X(z,t)), \tag{2.37}$$

where $v_s(X(z,t))$ is the Vesilind settling velocity function having the exponential form

$$v_s(X(z,t)) = v_o e^{-nX(z,t)}, \tag{2.38}$$

with v_o and n being constant parameters [VVKR97]. Also, in (2.36), $Q_e(t)/A$ is the upward bulk velocity (the effluent), and $Q_u(t)/A$ the downward bulk velocity (the underflow). A is the area of cross subsection of the clarifier tank. $Q_f(t)/A$ and $X_f(t)/A$ are the source velocity and concentration respectively. Notice that writing the flux in this form means that we consider that there is a net flow of liquid downwards below the inlet, which is a sum of the underflow and source fluxes, and a net flow upward above the inlet, consisting of the effluent. Fluxes are positive or negative, depending on whether they are going out of the clarifier or coming into it.

The *Heaviside theta function,* or the *unit step function,* appearing above, is defined as follows:

$$\theta(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}\,. \tag{2.39}$$

Its derivative is given by the Dirac delta function:

$$\theta'(z) \equiv \frac{d\theta(z)}{dz} = \delta(z)\,. \tag{2.40}$$

$\theta(z)$ also has the property:

$$\theta(z) = 1 - \theta(-z)\,. \tag{2.41}$$

We shall use these properties later when we transform the PDE to a DAE.

Effluent overflow

In the case where the effluent simply overflows, the convective flux reduces to the simple form:

$$\begin{aligned} F(X(z,t),z,t) &= G(X(z,t)) \\ &\quad +X(z,t)\,\frac{Q_u(t)}{A} - X_f(t)\,\frac{Q_f(t)}{A}\,\theta(z-z_f)\,. \end{aligned} \tag{2.42}$$

This just means that the bulk flow consists of a net flow downwards everywhere (with the velocity of the underflow), and a source flux coming in at the location of the inlet, at $z = z_f$.

Initial and Boundary conditions

As the initial condition for the concentration, we assume a given concentration profile. As a particular example, we first take the simplest case of a constant concentration profile:

$$\begin{aligned} X(z,t=0) &= X_0(z) \\ &= X_0, \end{aligned} \tag{2.43}$$

where X_0 is a constant for all z.

From the discussion in [A.V98a], we are led to the following boundary conditions for the clarifier:

$$\begin{aligned} \left.\frac{\partial X(z,t)}{\partial z}\right|_{z=0} &= 0 \\ \left.\frac{\partial X(z,t)}{\partial z}\right|_{z=L} &= 0. \end{aligned} \tag{2.44}$$

These boundary conditions hold for both cases of the effluent flow discussed above.

2.8.4 The PDE for batch sedimentation

For the batch sedimentation case, the convective flux comprises only the gravitational settling, since there are no source or bulk flows present. That is, $F(X(z,t)) \equiv G(X(z,t))$. Therefore, the PDE simply becomes:

$$\frac{\partial X(z,t)}{\partial t} = -\frac{\partial G(X(z,t))}{\partial z} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}. \tag{2.45}$$

Initial and Boundary conditions

We use the same initial condition as that for the continuous sedimentation case. That is,

$$\begin{aligned} X(z,t=0) &= X_0(z) \\ &= X_0\,. \end{aligned} \tag{2.46}$$

We have derived the following boundary conditions for the batch sedimentation case in [A.V98a]:

$$\begin{aligned} G(X(0,t)) - D_0 \left.\frac{\partial X(z,t)}{\partial z}\right|_{z=0} &= 0 \\ G(X(L,t)) - D_0 \left.\frac{\partial X(z,t)}{\partial z}\right|_{z=L} &= 0. \end{aligned} \tag{2.47}$$

2.8.5 Discretization of the PDE

We will first set up the discretization and collocation points [Kop97] for the general form of the PDE given in Eq.(2.3), and then apply the procedure for the two test case conditions separately.

We first use a normalized dimensionless variable h to replace the z- coordinate in the PDE of Eq.(2.3), defined by:

$$h = z/L, \tag{2.48}$$

so that as z varies over $[0,L]$, h correspondingly varies over $[0,1]$.

Therefore Eq.(2.3) becomes:

$$\frac{\partial X(Lh,t)}{\partial t} = -\frac{1}{L}\frac{\partial F(X(Lh,t))}{\partial h} + \frac{D_0}{L^2}\frac{\partial^2 X(Lh,t)}{\partial h^2}. \tag{2.49}$$

We divide the domain [0,1] into N_E elements or intervals, by defining (N_E+1) nodes:
$\{H_1 \equiv 0, H_2, H_3, \ldots, H_{N_E+1} \equiv 1\}$. The width of the i-th element is thus given by:

$$\Delta_i = H_{i+1} - H_i, \quad i = 1, \ldots, N_E. \tag{2.50}$$

We now need to define a further internal 'reduced' variable on each element, such that every element is again rescaled to $[0,1]$. Thus for the i-th element $(i = 1, \ldots N_E)$, we define this reduced variable $\xi^{(i)}$ as follows:

$$\xi^{(i)} = \frac{h - H_i}{\Delta_i}, \tag{2.51}$$

so that $\xi^{(i)}$ varies over $[0,1]$ as h varies over $[H_i, H_{i+1}]$.

We use the values of the quantity X at the end-points of every interval i, and those at $N(i)$ *interior collocation points*, to approximate X over this interval using the Lagrange interpolation formula as in Eq.(2.11). These interior collocation points are chosen as the zeroes of orthogonal Jacobi polynomials, as discussed above. We thus have a total of $(N(i)+2)$ coefficients, obtained by evaluating the concentration X at the $N(i)+2$

points $\{\xi_0^{(i)} \equiv 0, \xi_1^{(i)}, \ldots, \xi_{N(i)}^{(i)}, \xi_{N(i)+1}^{(i)} \equiv 1\}$, which can determine the polynomial approximation $X^{(i)}$ of X over this interval (of degree $N(i)+1$). We first write:

$$X^{(i)}(Lh,t) = X^{(i)}(L(\xi^{(i)}\Delta_i + H_i), t) = X^{(i)}(\xi^{(i)}, t) \tag{2.52}$$

$X^{(i)}(\xi^{(i)},t)$ is thus given by:

$$X^{(i)}(\xi^{(i)},t) \simeq \sum_{k=0}^{N(i)+1} \ell_k^{(i)}(\xi^{(i)})\, X^{(i)}(\xi_k^{(i)},t). \tag{2.53}$$

Here the $\{\ell_k^{(i)}(\xi^{(i)})\}$ are Lagrange polynomials of degree $(N(i)+1)$. The total number of unknowns, $\{X^{(i)}(\xi_k^{(i)},t)\}$, is therefore equal to the total number of collocation points, N_T, which is the sum (over all intervals) of all the collocation points in every interval - including the interior ones and the end-points of the interval. That is,

$$N_T = \sum_{i=1}^{N_E} (N(i)+2). \tag{2.54}$$

The total number of interior collocation points, N_C, is given by:

$$N_C = \sum_{i=1}^{N_E} N(i). \tag{2.55}$$

Therefore we have:

$$N_T = N_C + 2N_E. \tag{2.56}$$

In order to determine the N_T unknowns uniquely, we need at least N_T equations in the unknowns. We can account for these as follows: there will be N_C DAEs resulting from the discretization of the PDE at the N_C interior collocation points. In addition, there are two equations in the two collocation points at the domain boundaries ($z = 0, L$), given by the boundary conditions in Eq.(2.47). Also, there are two equations arising from the continuity of the concentration $X(z,t)$ and its first derivative $\partial X/\partial z$, at each of the $(N_E - 1)$ nodes $\{H_2, \ldots, H_{N_E}\}$. Thus the total number of equations we have at our disposal is $N_C + 2 + 2(N_E - 1) \equiv N_T$. In addition, there are N_C initial conditions available for solving the DAE at the N_C interior collocation points.

Using the expansion (2.53), we can define the derivatives of X as well. Thus,

$$\left.\frac{\partial X^{(i)}(\xi^{(i)},t)}{\partial \xi^{(i)}}\right|_{\xi^{(i)}=\xi_j^{(i)}} = \sum_{k=0}^{N(i)+1} \left.\frac{\partial \ell_k^{(i)}(\xi^{(i)})}{\partial \xi^{(i)}}\right|_{\xi^{(i)}=\xi_j^{(i)}} X^{(i)}(\xi_k^{(i)},t). \tag{2.57}$$

Similarly,

$$\left.\frac{\partial^2 X^{(i)}(\xi^{(i)},t)}{\partial {\xi^{(i)}}^2}\right|_{\xi^{(i)}=\xi_j^{(i)}} = \sum_{k=0}^{N(i)+1} \left.\frac{\partial^2 \ell_k^{(i)}(\xi^{(i)})}{\partial {\xi^{(i)}}^2}\right|_{\xi^{(i)}=\xi_j^{(i)}} X^{(i)}(\xi_k^{(i)},t). \tag{2.58}$$

We set

$$\begin{aligned}
X^{(i)}(\xi_j^{(i)},t) &= X_j^{(i)}(t);\\
\left.\frac{\partial \ell_k^{(i)}(\xi^{(i)})}{\partial \xi^{(i)}}\right|_{\xi^{(i)}=\xi_j^{(i)}} &= A_{j,k}^{(i)};\\
\left.\frac{\partial^2 \ell_k^{(i)}(\xi^{(i)})}{\partial {\xi^{(i)}}^2}\right|_{\xi^{(i)}=\xi_j^{(i)}} &= B_{j,k}^{(i)};\\
i &= 1,\ldots,N_E,\\
j &= 0,\ldots,N(i)+1,\\
k &= 0,\ldots,N(i)+1.
\end{aligned} \tag{2.59}$$

Therefore we can rewrite Eqns.(2.57) and (2.58) as:

$$\left.\frac{\partial X^{(i)}(\xi^{(i)},t)}{\partial \xi^{(i)}}\right|_{\xi^{(i)}=\xi_j^{(i)}} = \sum_{k=0}^{N(i)+1} A_{j,k}^{(i)} X_k^{(i)}(t)$$
$$\left.\frac{\partial^2 X^{(i)}(\xi^{(i)},t)}{\partial {\xi^{(i)}}^2}\right|_{\xi^{(i)}=\xi_j^{(i)}} = \sum_{k=0}^{N(i)+1} B_{j,k}^{(i)} X_k^{(i)}(t) \qquad (2.60)$$

Note that the square matrices $A^{(i)}$ and $B^{(i)}$ both have the dimensions $(N(i)+2)\times(N(i)+2)$.

Using the discretized version of the concentration and its derivatives as indicated above, we finally arrive at the required DAEs for the concentration X at each interior collocation point. These DAEs are supplemented by the discretized versions of the initial condition at the interior collocation points, the boundary conditions at the domain boundaries, and the continuity conditions for the concentration and its first derivative at the (interior) element boundaries. We now consider the two test cases separately.

2.8.6 The DAE for continuous sedimentation

We use the explicit form of the derivative of the flux F in Eq.(2.35). The derivative of the gravitational flux is given by:

$$\frac{\partial G(X(z,t))}{\partial z} = (1-nX(z,t))\, v_0 e^{-nX(z,t)} \frac{\partial X(z,t)}{\partial z}$$
$$= (1-nX(z,t))\, \frac{G(X(z,t))}{X(z,t)} \frac{\partial X(z,t)}{\partial z}. \qquad (2.61)$$

We again consider the two cases for the effluent flow:

Effluent pumped out

We use the properties of $\theta(z)$ - Eqns.(2.40) and (2.41), to simplify the other terms appearing in the flux in Eq.(2.36). We also use the relation [A.V98a]:

$$\frac{Q_f(t)}{A} = \frac{Q_u(t)}{A} + \frac{Q_e(t)}{A} \qquad (2.62)$$

to eliminate $Q_e(t)/A$, since the effluent flux is an output of the calculation, and is not known beforehand.

After some algebra, we arrive at the following expression for the derivative of the flux :

$$\frac{\partial F(X(z,t))}{\partial z} = \left[(1-nX(z,t))\, v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A}\,\theta(z_f-z)\right]\frac{\partial X(z,t)}{\partial z}$$
$$+ (X(z,t) - X_f(t))\,\frac{Q_f(t)}{A}\,\delta(z-z_f)\,. \qquad (2.63)$$

Therefore the PDE for continuous sedimentation with the effluent pumped out becomes:

$$\frac{\partial X(z,t)}{\partial t} = -\left[(1-nX(z,t))\, v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A}\,\theta(z_f-z)\right]\frac{\partial X(z,t)}{\partial z}$$
$$- (X(z,t) - X_f(t))\,\frac{Q_f(t)}{A}\,\delta(z-z_f)$$
$$+ D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}\,. \qquad (2.64)$$

For the convenient numerical implementation of the above PDE as a DAE, we use the defining numerical values of $\theta(z-z_f)$. We shall also represent the Dirac delta function by a rectangular pulse of unit height and width 2σ, centred at $z=z_f$. σ is assumed to be small.

Therefore the PDE in Eq.(2.64) will consist of three parts:
1. for $\{\, 0 \le z \le z_f - \sigma \}$:

$$\frac{\partial X(z,t)}{\partial t} = -\left[(1 - nX(z,t))\ v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A} \right] \frac{\partial X(z,t)}{\partial z} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}; \tag{2.65}$$

2. for $\{\, z_f - \sigma < z < z_f + \sigma \}$:

$$\frac{\partial X(z,t)}{\partial t} = -\left[(1 - nX(z,t))\ v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A} \right] \frac{\partial X(z,t)}{\partial z} - (X(z,t) - X_f(t)) \frac{Q_f(t)}{A} \frac{1}{2\sigma} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}; \tag{2.66}$$

3. for $\{\, z_f + \sigma \le z \le L \}$:

$$\frac{\partial X(z,t)}{\partial t} = -\left[(1 - nX(z,t))\ v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A} \right] \frac{\partial X(z,t)}{\partial z} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}. \tag{2.67}$$

Using Eqns.(2.53)-(2.60), we can now write the DAE for continuous sedimentation. Let us specify the finite elements such that the unit pulse (representing the delta function) spans element N_D. Therefore, the N_D-th element is of width 2σ. We set $\sigma = \eta L$, where η has a small value. The DAE is written for all the N_C interior collocation points. Again, the DAE will be written in three parts: 1. for $\{\, t = 0;\ i = 1, \ldots N_D - 1;\ j = 1, \ldots N(i) \}$

$$\frac{dX_j^{(i)}(t)}{dt} = -\frac{1}{L\Delta_i}\left[\left(1 - nX_j^{(i)}(t)\right)\ v_0\, e^{-nX_j^{(i)}(t)} + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A} \right] \sum_{k=0}^{N(i)+1} A_{j,k}^{(i)} X_k^{(i)}(t) + \frac{D_0}{L^2\Delta_i^2} \sum_{k=0}^{N(i)+1} B_{j,k}^{(i)} X_k^{(i)}(t); \tag{2.68}$$

We can group the summands on the RHS of the above equation to obtain:

$$\frac{dX_j^{(i)}(t)}{dt} = \sum_{k=0}^{N(i)+1} \left(-\frac{1}{L\Delta_i}\left[\left(1 - nX_j^{(i)}(t)\right)\ v_0\, e^{-nX_j^{(i)}(t)} + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A} \right] A_{j,k}^{(i)} + \frac{D_0}{L^2\Delta_i^2} B_{j,k}^{(i)} \right) X_k^{(i)}(t). \tag{2.69}$$

Similarly, we have:
2. for $\{\, t = 0;\ i = N_D;\ j = 1, \ldots N(N_D) \}$

$$\frac{dX_j^{(i)}(t)}{dt} = \sum_{k=0}^{N(i)+1} \left(-\frac{1}{L\Delta_i}\left[\left(1 - nX_j^{(i)}(t)\right)\ v_0\, e^{-nX_j^{(i)}(t)} + \frac{Q_u(t)}{A} \right] A_{j,k}^{(i)} + \frac{D_0}{L^2\Delta_i^2} B_{j,k}^{(i)} \right) X_k^{(i)}(t) - \left(X_j^{(i)}(t) - X_f(t) \right) \frac{Q_f(t)}{A} \frac{1}{2\eta L}. \tag{2.70}$$

3. for $\{\, t = 0;\ i = N_D + 1, \ldots N_E;\ j = 1, \ldots N(i) \}$

$$\frac{dX_j^{(i)}(t)}{dt} = \sum_{k=0}^{N(i)+1} \left(-\frac{1}{L\Delta_i}\left[\left(1 - nX_j^{(i)}(t)\right)\ v_0\, e^{-nX_j^{(i)}(t)} + \frac{Q_u(t)}{A} \right] A_{j,k}^{(i)} + \frac{D_0}{L^2\Delta_i^2} B_{j,k}^{(i)} \right) X_k^{(i)}(t). \tag{2.71}$$

The initial condition, Eq.(2.43), is implemented as follows, again for the N_C interior collocation points:

$$\begin{aligned} X_j^{(i)}(t=0) &= X_{j0}^{(i)} = X_0\,; \\ i &= 1,\ldots,N_E\,; \\ j &= 1,\ldots,N(i)\,. \end{aligned} \tag{2.72}$$

The two boundary conditions, Eqns.(2.44), are implemented at the collocation points corresponding to $z=0$ and $z=L$:

$$\begin{aligned} \frac{D_0}{L\Delta_1}\sum_{k=0}^{N(1)+1} A_{0,k}^{(1)} X_k^{(1)}(t) &= 0\,; \\ \frac{D_0}{L\Delta_{N_E}}\sum_{k=0}^{N(N_E)+1} A_{N(N_E)+1,k}^{(N_E)} X_k^{(N_E)}(t) &= 0\,. \end{aligned} \tag{2.73}$$

We now have to implement the element boundary conditions. As discussed earlier, we assume that both the concentration X and its first derivative $\partial X/\partial z$ are continuous at all the interior nodes which are the boundaries between neighbouring elements. This is implemented as follows: for $(i=1,\ldots,N_E-1)$,

$$\begin{aligned} X_0^{(i+1)}(t) &= X_{N(i)+1}^{(i)}(t); \\ \frac{1}{L\Delta_{i+1}}\sum_{k=0}^{N(i+1)+1} A_{0,k}^{(i+1)} X_k^{(i+1)}(t) &= \frac{1}{L\Delta_i}\sum_{k=0}^{N(i)+1} A_{N(i)+1,k}^{(i)} X_k^{(i)}(t). \end{aligned} \tag{2.74}$$

Effluent overflow

The discussion leading to the three DAEs is similar to that in the previous subsubsection, including the initial and boundary conditions. The PDE in this case will be given by:

$$\begin{aligned} \frac{\partial X(z,t)}{\partial t} &= -\left[\left(1-nX(z,t)\right)\,v_0\,e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + X_f(t)\,\frac{Q_f(t)}{A}\,\delta(z-z_f) \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}\,. \end{aligned} \tag{2.75}$$

Again using the representation of the delta function by a unit pulse, we have the three parts of the PDE:
1. for $\{\,0 \le z \le z_f-\sigma\}$:

$$\begin{aligned} \frac{\partial X(z,t)}{\partial t} &= -\left[\left(1-nX(z,t)\right)\,v_0\,e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}\,; \end{aligned} \tag{2.76}$$

2. for $\{\,z_f-\sigma < z < z_f+\sigma\}$:

$$\begin{aligned} \frac{\partial X(z,t)}{\partial t} &= -\left[\left(1-nX(z,t)\right)\,v_0\,e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + X_f(t)\,\frac{Q_f(t)}{A}\,\frac{1}{2\sigma} \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}\,; \end{aligned} \tag{2.77}$$

3. for $\{ z_f + \sigma \leq z \leq L \}$:

$$\begin{aligned} \frac{\partial X(z,t)}{\partial t} &= -\left[(1 - nX(z,t))\ v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A} \right] \frac{\partial X(z,t)}{\partial z} \\ &\quad + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}. \end{aligned} \tag{2.78}$$

Note that the first and the third PDEs are actually identical. The corresponding DAEs will thus be given by:

$$\frac{dX_j^{(i)}(t)}{dt} = \sum_{k=0}^{N(i)+1} \left(-\frac{1}{L\Delta_i} \left[\left(1 - nX_j^{(i)}(t) \right)\ v_0\, e^{-nX_j^{(i)}(t)} + \frac{Q_u(t)}{A} \right] A_{j,k}^{(i)} + \frac{D_0}{L^2\Delta_i^2} B_{j,k}^{(i)} \right) X_k^{(i)}(t). \tag{2.79}$$

Similarly, we have:

2. for $\{ t = 0;\ i = N_D;\ j = 1, \ldots N(N_D) \}$

$$\begin{aligned} \frac{dX_j^{(i)}(t)}{dt} &= \sum_{k=0}^{N(i)+1} \left(-\frac{1}{L\Delta_i} \left[\left(1 - nX_j^{(i)}(t) \right)\ v_0\, e^{-nX_j^{(i)}(t)} + \frac{Q_u(t)}{A} \right] A_{j,k}^{(i)} + \frac{D_0}{L^2\Delta_i^2} B_{j,k}^{(i)} \right) X_k^{(i)}(t) \\ &\quad - \left(X_j^{(i)}(t) - X_f(t) \right) \frac{Q_f(t)}{A} \frac{1}{2\eta L}. \end{aligned} \tag{2.80}$$

3. for $\{ t = 0;\ i = N_D + 1, \ldots N_E;\ j = 1, \ldots N(i) \}$

$$\frac{dX_j^{(i)}(t)}{dt} = \sum_{k=0}^{N(i)+1} \left(-\frac{1}{L\Delta_i} \left[\left(1 - nX_j^{(i)}(t) \right)\ v_0\, e^{-nX_j^{(i)}(t)} + \frac{Q_u(t)}{A} \right] A_{j,k}^{(i)} + \frac{D_0}{L^2\Delta_i^2} B_{j,k}^{(i)} \right) X_k^{(i)}(t). \tag{2.81}$$

Again, note that the first and third DAEs above are the same. The initial and boundary conditions remain identical to the case of the effluent being pumped out.

2.8.7 The DAE for batch sedimentation

Using the explicit form of $(\partial G/\partial z)$ derived in Eq.(2.61), the PDE for batch sedimentation has the form:

$$\frac{\partial X(z,t)}{\partial t} = -\left((1 - nX(z,t))\, v_0\, e^{-nX(z,t)} \right) \frac{\partial X(z,t)}{\partial z} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}. \tag{2.82}$$

Using Eqns.(2.53)-(2.60), we write the DAE for batch sedimentation just as we did for the continuous case in the previous subsection. Again, the DAE is written for all the N_C interior collocation points.

$$\frac{dX_j^{(i)}(t)}{dt} = -\frac{1}{L\Delta_i} \left(\left(1 - nX_j^{(i)}(t) \right)\ v_0\, e^{-nX_j^{(i)}(t)} \right) \sum_{k=0}^{N(i)+1} A_{j,k}^{(i)} X_k^{(i)}(t) + \frac{D_0}{L^2\Delta_i^2} \sum_{k=0}^{N(i)+1} B_{j,k}^{(i)} X_k^{(i)}(t), \tag{2.83}$$

for:

$$\begin{aligned} t &> 0; \\ i &= 1, \ldots, N_E; \\ j &= 1, \ldots, N(i). \end{aligned} \tag{2.84}$$

We can group the summands on the RHS of Eq.(2.83) as before, to obtain, finally:

$$\frac{dX_j^{(i)}(t)}{dt} = \sum_{k=0}^{N(i)+1} \left[-\frac{1}{L\Delta_i} \left(\left(1 - nX_j^{(i)}(t) \right)\ v_0\, e^{-nX_j^{(i)}(t)} \right) A_{j,k}^{(i)} + \frac{D_0}{L^2\Delta_i^2} B_{j,k}^{(i)} \right] X_k^{(i)}(t). \tag{2.85}$$

The initial condition for the batch case, Eq.(2.46), is implemented as follows, again for the N_C interior collocation points:

$$\begin{aligned} X_j^{(i)}(t=0) &= X_{j0}^{(i)} = X_0\,; \\ i &= 1,\ldots,N_E\,; \\ j &= 1,\ldots,N(i)\,. \end{aligned} \tag{2.86}$$

The two corresponding boundary conditions for the batch case, Eqns.(2.47), are again implemented at the collocation points corresponding to $z=0$ and $z=L$:

$$\begin{aligned} \frac{D_0}{L\Delta_1}\sum_{k=0}^{N(1)+1} A_{0,k}^{(1)} X_k^{(1)}(t) &= G(X_0^{(1)}(t)) \\ &= X_0^{(1)}(t)\,v_0\,e^{-nX_0^{(1)}(t)}\,; \\ \frac{D_0}{L\Delta_{N_E}}\sum_{k=0}^{N(N_E)+1} A_{N(N_E)+1,k}^{(N_E)} X_k^{(N_E)}(t) &= G(X_{N(N_E)+1}^{(N_E)}(t)) \\ &= X_{N(N_E)+1}^{(N_E)}(t)\,v_0\,e^{-nX_{N(N_E)+1}^{(N_E)}(t)}\,. \end{aligned} \tag{2.87}$$

The element boundary conditions are implemented as follows, just as for the continuous case: for ($i = 1,\ldots,N_E-1$),

$$\begin{aligned} X_0^{(i+1)}(t) &= X_{N(i)+1}^{(i)}(t); \\ \frac{1}{L\Delta_{i+1}}\sum_{k=0}^{N(i+1)+1} A_{0,k}^{(i+1)} X_k^{(i+1)}(t) &= \frac{1}{L\Delta_i}\sum_{k=0}^{N(i)+1} A_{N(i)+1,k}^{(i)} X_k^{(i)}(t). \end{aligned} \tag{2.88}$$

2.8.8 Examples

We shall first illustrate the conversion of the PDE for batch sedimentation, Eq.(2.45), to a DAE using the orthogonal collocation method on finite elements. Here we shall discuss the method in detail. Following this we present a brief illustration of the same method applied to the continuous sedimentation case.

An example for batch sedimentation

The PDE to be discretized is the following (using the dimensionless variable h):

$$\frac{\partial X(Lh,t)}{\partial t} = -\frac{1}{L}\frac{\partial G(X(Lh,t))}{\partial h} + \frac{D_0}{L^2}\frac{\partial^2 X(Lh,t)}{\partial h^2}. \tag{2.89}$$

We divide the domain of interest $[0,1]$ into *two* elements, by defining three nodes $\{H_1=0, H_2=0.4, H_3=1\}$. On the first element, we locate one interior collocation point, and two on the second. We set $\alpha=\beta=0$. Thus we have:

$$\begin{aligned} \alpha &= 0; \\ \beta &= 0; \\ N_E &= 2; \\ i &= 1,2 \\ \Delta_1 &= (H_2-H_1) = 0.4; \\ \Delta_2 &= (H_3-H_2) = 0.6; \\ N(1) &= 1; \\ N(2) &= 2; \\ N_C &= 3; \\ N_T &= 7; \end{aligned} \tag{2.90}$$

Now the appropriate Jacobi polynomials are $(G_1(0,0,x),\, G_2(0,0,x))$. These are given by:

$$\begin{aligned} G_1(0,0,x) &= x - 1/2; \\ G_2(0,0,x) &= x^2 - x + 1/6; \end{aligned} \tag{2.91}$$

Their roots are given by: (0.5), and $(0.21132,\, 0.78868)$ respectively. Thus the collocation points $\{\xi_j^{(i)}\}$ are given by:

$$\begin{aligned} \xi_0^{(1)} &= 0.0; \\ \xi_1^{(1)} &= 0.5; \\ \xi_2^{(1)} &= 1.0; \\ \xi_0^{(2)} &= 0.0; \\ \xi_1^{(2)} &= 0.21132; \\ \xi_2^{(2)} &= 0.78868; \\ \xi_3^{(2)} &= 1.0; \end{aligned} \tag{2.92}$$

The polynomial approximation to the concentration X over each element will then have the form in Eq.(2.53):

$$\begin{aligned} X^{(1)}(\xi^{(1)},t) &= \sum_{k=0}^{2} \ell_k^{(1)}(\xi^{(1)})\, X^{(1)}(\xi_k^{(1)},t); \\ X^{(2)}(\xi^{(2)},t) &= \sum_{k=0}^{3} \ell_k^{(2)}(\xi^{(2)})\, X^{(2)}(\xi_k^{(2)},t). \end{aligned} \tag{2.93}$$

The polynomials $\{p_i(x)\}$ of Eq.(2.18) used to define the required Lagrange polynomials and their derivatives over each interval are given by:

$$p_i(\xi^{(i)}) = \prod_{j=0}^{n} (\xi^{(i)} - \xi_j^{(i)}), \tag{2.94}$$

where $p_i(\xi^{(i)})$ indicates the polynomial on the i-th element. Thus we have:

$$\begin{aligned} p_1(\xi^{(1)}) &= (\xi^{(1)} - \xi_0^{(1)})\,(\xi^{(1)} - \xi_1^{(1)})\,(\xi^{(1)} - \xi_2^{(1)}) \\ &\equiv (\xi^{(1)} - 0.0)\,(\xi^{(1)} - 0.5)\,(\xi^{(1)} - 1.0). \end{aligned} \tag{2.95}$$

and

$$\begin{aligned} p_2(\xi^{(2)}) &= (\xi^{(2)} - \xi_0^{(2)})\,(\xi^{(2)} - \xi_1^{(2)})\,(\xi^{(2)} - \xi_2^{(2)})\,(\xi^{(2)} - \xi_3^{(2)}) \\ &\equiv (\xi^{(2)} - 0.0)\,(\xi^{(2)} - 0.21132)\,(\xi^{(2)} - 0.78868)\,(\xi^{(2)} - 1.0). \end{aligned} \tag{2.96}$$

From the above expressions for (p_1, p_2), the Lagrange polynomials, their derivatives, and hence the matrices $(A^{(1)}, B^{(1)})$ and $(A^{(2)}, B^{(2)})$ can be obtained, using Eqns.(2.19) - (2.21) and the definitions (2.59). We thus have [Oh95, MV72] (note that these matrices are the transposes of the ones in [Oh95]):

$$A^{(1)} \equiv \begin{bmatrix} -3.0 & 4.0 & -1.0 \\ -1.0 & 0.0 & 1.0 \\ 1.0 & -4.0 & 3.0 \end{bmatrix}; \tag{2.97}$$

$$B^{(1)} \equiv \begin{bmatrix} 4.0 & -8.0 & 4.0 \\ 4.0 & -8.0 & 4.0 \\ 4.0 & -8.0 & 4.0 \end{bmatrix}; \tag{2.98}$$

$$A^{(2)} \equiv \begin{bmatrix} -7.00000 & 8.19615 & -2.19615 & 1.00000 \\ -2.73205 & 1.73205 & 1.73205 & -0.73205 \\ 0.73205 & -1.73205 & -1.73205 & 2.73205 \\ -1.00000 & 2.19615 & -8.19615 & 7.00000 \end{bmatrix}; \tag{2.99}$$

$$B^{(2)} \equiv \begin{bmatrix} 24.00000 & -37.17691 & 25.17691 & -12.00000 \\ 16.39230 & -24.00000 & 12.00000 & -4.39230 \\ -4.39230 & 12.00000 & -24.00000 & 16.39230 \\ -12.00000 & 25.17691 & -37.17691 & 24.00000 \end{bmatrix}. \tag{2.100}$$

We can now write down explicitly the DAEs for the concentration at the $N_C = 3$ interior collocation points, Eq.(2.85). For the collocation point $\xi_1^{(1)}$ within the first element, we have:

$$\frac{dX_1^{(1)}(t)}{dt} = \sum_{k=0}^{2} \left[-\frac{1}{L\Delta_1} \left(\left(1 - nX_1^{(1)}(t) \right) \nu_0 \, e^{-nX_1^{(1)}(t)} \right) A_{1,k}^{(1)} + \frac{D_0}{L^2\Delta_1^2} B_{1,k}^{(1)} \right] X_k^{(1)}(t). \tag{2.101}$$

Expanding the sum above, we have:

$$\begin{aligned} \frac{dX_1^{(1)}(t)}{dt} &= \left[-\frac{1}{L\Delta_1} \left(\left(1 - nX_1^{(1)}(t) \right) \nu_0 \, e^{-nX_1^{(1)}(t)} \right) A_{1,0}^{(1)} + \frac{D_0}{L^2\Delta_1^2} B_{1,0}^{(1)} \right] X_0^{(1)}(t) \\ &+ \left[-\frac{1}{L\Delta_1} \left(\left(1 - nX_1^{(1)}(t) \right) \nu_0 \, e^{-nX_1^{(1)}(t)} \right) A_{1,1}^{(1)} + \frac{D_0}{L^2\Delta_1^2} B_{1,1}^{(1)} \right] X_1^{(1)}(t) \\ &+ \left[-\frac{1}{L\Delta_1} \left(\left(1 - nX_1^{(1)}(t) \right) \nu_0 \, e^{-nX_1^{(1)}(t)} \right) A_{1,2}^{(1)} + \frac{D_0}{L^2\Delta_1^2} B_{1,2}^{(1)} \right] X_2^{(1)}(t). \end{aligned} \tag{2.102}$$

Similarly at the two interior collocation points ($\xi_1^{(2)} = 0.21132$, $\xi_2^{(2)} = 0.78868$), we have the DAEs:

$$\begin{aligned} \frac{dX_1^{(2)}(t)}{dt} &= \sum_{k=0}^{3} \left[-\frac{1}{L\Delta_2} \left(\left(1 - nX_1^{(2)}(t) \right) \nu_0 \, e^{-nX_1^{(2)}(t)} \right) A_{1,k}^{(2)} + \frac{D_0}{L^2\Delta_2^2} B_{1,k}^{(2)} \right] X_k^{(2)}(t); \\ \frac{dX_2^{(2)}(t)}{dt} &= \sum_{k=0}^{3} \left[-\frac{1}{L\Delta_2} \left(\left(1 - nX_2^{(2)}(t) \right) \nu_0 \, e^{-nX_2^{(2)}(t)} \right) A_{2,k}^{(2)} + \frac{D_0}{L^2\Delta_2^2} B_{2,k}^{(2)} \right] X_k^{(2)}(t). \end{aligned} \tag{2.103}$$

Expanding the sums again, we have:

$$\begin{aligned} \frac{dX_1^{(2)}(t)}{dt} &= \left[-\frac{1}{L\Delta_2} \left(\left(1 - nX_1^{(2)}(t) \right) \nu_0 \, e^{-nX_1^{(2)}(t)} \right) A_{1,0}^{(2)} + \frac{D_0}{L^2\Delta_2^2} B_{1,0}^{(2)} \right] X_0^{(2)}(t) \\ &+ \left[-\frac{1}{L\Delta_2} \left(\left(1 - nX_1^{(2)}(t) \right) \nu_0 \, e^{-nX_1^{(2)}(t)} \right) A_{1,1}^{(2)} + \frac{D_0}{L^2\Delta_2^2} B_{1,1}^{(2)} \right] X_1^{(2)}(t) \\ &+ \left[-\frac{1}{L\Delta_2} \left(\left(1 - nX_1^{(2)}(t) \right) \nu_0 \, e^{-nX_1^{(2)}(t)} \right) A_{1,2}^{(2)} + \frac{D_0}{L^2\Delta_2^2} B_{1,2}^{(2)} \right] X_2^{(2)}(t) \\ &+ \left[-\frac{1}{L\Delta_2} \left(\left(1 - nX_1^{(2)}(t) \right) \nu_0 \, e^{-nX_1^{(2)}(t)} \right) A_{1,3}^{(2)} + \frac{D_0}{L^2\Delta_2^2} B_{1,3}^{(2)} \right] X_3^{(2)}(t). \end{aligned} \tag{2.104}$$

and:

$$\begin{aligned}
\frac{dX_2^{(2)}(t)}{dt} &= \left[-\frac{1}{L\Delta_2}\left(\left(1-nX_2^{(2)}(t)\right)\,v_0\,e^{-nX_2^{(2)}(t)}\right)A_{2,0}^{(2)}+\frac{D_0}{L^2\Delta_2^2}B_{2,0}^{(2)}\right]X_0^{(2)}(t) \\
&+\left[-\frac{1}{L\Delta_2}\left(\left(1-nX_2^{(2)}(t)\right)\,v_0\,e^{-nX_2^{(2)}(t)}\right)A_{2,1}^{(2)}+\frac{D_0}{L^2\Delta_2^2}B_{2,1}^{(2)}\right]X_1^{(2)}(t) \\
&+\left[-\frac{1}{L\Delta_2}\left(\left(1-nX_2^{(2)}(t)\right)\,v_0\,e^{-nX_2^{(2)}(t)}\right)A_{2,2}^{(2)}+\frac{D_0}{L^2\Delta_2^2}B_{2,2}^{(2)}\right]X_2^{(2)}(t) \\
&+\left[-\frac{1}{L\Delta_2}\left(\left(1-nX_2^{(2)}(t)\right)\,v_0\,e^{-nX_2^{(2)}(t)}\right)A_{2,3}^{(2)}+\frac{D_0}{L^2\Delta_2^2}B_{2,3}^{(2)}\right]X_3^{(2)}(t)\,. \qquad (2.105)
\end{aligned}$$

The three DAEs above (Eqns.(2.102)-(2.105)) must be solved together with the initial conditions:

$$\begin{aligned}
X_1^{(1)}(t=0) &= X_{10}^{(1)}=X_0\,; \\
X_1^{(2)}(t=0) &= X_{10}^{(2)}=X_0\,; \\
X_2^{(2)}(t=0) &= X_{20}^{(2)}=X_0\,. \qquad (2.106)
\end{aligned}$$

The two boundary conditions (at $z=0,L$) are now given by the equations:

$$\begin{aligned}
\frac{D_0}{L\Delta_1}\sum_{k=0}^{2}A_{0,k}^{(1)}X_k^{(1)}(t) &= G(X_0^{(1)}(t)) \\
&= X_0^{(1)}(t)\,v_0\,e^{-nX_0^{(1)}(t)}\,; \\
\frac{D_0}{L\Delta_2}\sum_{k=0}^{3}A_{3,k}^{(2)}X_k^{(2)}(t) &= G(X_3^{(2)}(t)) \\
&= X_3^{(2)}(t)\,v_0\,e^{-nX_3^{(2)}(t)}\,. \qquad (2.107)
\end{aligned}$$

Expanding them, we have:

$$\begin{aligned}
\frac{D_0}{L\Delta_1}\left(A_{0,0}^{(1)}X_0^{(1)}(t)+A_{0,1}^{(1)}X_1^{(1)}(t)+A_{0,2}^{(1)}X_2^{(1)}(t)\right) &= G(X_0^{(1)}(t)) \\
&= X_0^{(1)}(t)\,v_0\,e^{-nX_0^{(1)}(t)}\,; \\
\frac{D_0}{L\Delta_2}\left(A_{3,0}^{(2)}X_0^{(2)}(t)+A_{3,1}^{(2)}X_1^{(2)}(t)+A_{3,2}^{(2)}X_2^{(2)}(t)+A_{3,3}^{(2)}X_3^{(2)}(t)\right) &= G(X_3^{(2)}(t)) \\
&= X_3^{(2)}(t)\,v_0\,e^{-nX_3^{(2)}(t)}. \qquad (2.108)
\end{aligned}$$

Finally, we write the element boundary conditions. We have just one pair, since there is only one interior node (H_2):

$$\begin{aligned}
X_0^{(2)}(t) &= X_2^{(1)}(t)\,; \\
\frac{1}{L\Delta_2}\sum_{k=0}^{3}A_{0,k}^{(2)}X_k^{(2)}(t) &= \frac{1}{L\Delta_1}\sum_{k=0}^{2}A_{2,k}^{(1)}X_k^{(1)}(t)\,. \qquad (2.109)
\end{aligned}$$

Expanding the second of the two equations above, we have:

$$\begin{aligned}
&\frac{1}{L\Delta_2}\left(A_{0,0}^{(2)}X_0^{(2)}(t)+A_{0,1}^{(2)}X_1^{(2)}(t)+A_{0,2}^{(2)}X_2^{(2)}(t)+A_{0,3}^{(2)}X_3^{(2)}(t)\right)= \\
&\qquad\frac{1}{L\Delta_1}\left(A_{2,0}^{(1)}X_0^{(1)}(t)+A_{2,1}^{(1)}X_1^{(1)}(t)+A_{2,2}^{(1)}X_2^{(1)}(t)\right)\,. \qquad (2.110)
\end{aligned}$$

Thus the set of $N_T=7$ equations we have to solve for the seven unknown concentration values $\{X_0^{(1)}(t),\ldots X_3^{(2)}(t)\}$ are: the three DAEs - Eqns.(2.102), (2.104), (2.105) with the three initial conditions - Eqns.(2.106) ; the two boundary conditions given by Eqns.(2.108) and the two element continuity conditions, Eqns.(2.110).

Matrix formulation

One way to solve the DAEs - Eqns.(2.102-2.105), is to use matrices to represent the unknown concentrations at the nodes and interior collocation points The associated domain boundary conditions and the element boundary conditions can also be combined and written in matrix form. In addition, there are the collocation matrices A and B over each element. If we have only linear terms appearing in the DAEs, we can finally even write the entire set of DAEs as a matrix equation. In our case, though, we do have non-linear terms in all the DAEs. However, we can still employ matrices to eliminate the unknowns at the nodes, to yield a coupled set of DAEs involving only the unknowns at the interior collocation points. We illustrate this procedure below.

Let us first define two vectors W and U, which are the vectors of unknown concentrations at the internal collocation points and at the nodes respectively. That is,

$$W \equiv \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \equiv \begin{bmatrix} X_1^{(1)}(t) \\ X_1^{(2)}(t) \\ X_2^{(2)}(t) \end{bmatrix} \tag{2.111}$$

and

$$U \equiv \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \equiv \begin{bmatrix} X_0^{(1)}(t) \\ X_2^{(1)}(t) = X_0^{(2)}(t) \\ X_3^{(2)}(t) \end{bmatrix} . \tag{2.112}$$

We also define the vector of initial values at the interior collocation points:

$$W^0 \equiv \begin{bmatrix} w_1^0 \\ w_2^0 \\ w_3^0 \end{bmatrix} \equiv \begin{bmatrix} X_1^{(1)}(t=0) \\ X_1^{(2)}(t=0) \\ X_2^{(2)}(t=0) \end{bmatrix} \tag{2.113}$$

Note that in defining U, we have already used the first of Eqns.(2.110). In general, the vectors W and W^0 have the dimension $[N_C \times 1]$, and U is of dimension $[(N_E+1)\times 1]$.

Thus the DAEs are written for the $\{w_i\}$, the unknown concentrations at the interior collocation points:

$$\begin{aligned} \frac{dw_1}{dt} &= \left[-a_1 g(w_1) A_{1,1}^{(1)} + c_1 B_{1,1}^{(1)}\right] w_1 \\ &+ \left[-a_1 g(w_1) A_{1,0}^{(1)} + c_1 B_{1,0}^{(1)}\right] u_1 \\ &+ \left[-a_1 g(w_1) A_{1,2}^{(1)} + c_1 B_{1,2}^{(1)}\right] u_2 ; \end{aligned} \tag{2.114}$$

$$\begin{aligned} \frac{dw_2}{dt} &= \left[-a_2 g(w_2) A_{1,1}^{(2)} + c_2 B_{1,1}^{(2)}\right] w_2 \\ &+ \left[-a_2 g(w_2) A_{1,2}^{(2)} + c_2 B_{1,2}^{(2)}\right] w_3 \\ &+ \left[-a_2 g(w_2) A_{1,0}^{(2)} + c_2 B_{1,0}^{(2)}\right] u_2 \\ &+ \left[-a_2 g(w_2) A_{1,3}^{(2)} + c_2 B_{1,3}^{(2)}\right] u_3 ; \end{aligned} \tag{2.115}$$

and:

$$\begin{aligned} \frac{dw_3}{dt} &= \left[-a_2 g(w_3) A_{2,1}^{(2)} + c_2 B_{2,1}^{(2)}\right] w_2 \\ &+ \left[-a_2 g(w_3) A_{2,2}^{(2)} + c_2 B_{2,2}^{(2)}\right] w_3 \\ &+ \left[-a_2 g(w_3) A_{2,0}^{(2)} + c_2 B_{2,0}^{(2)}\right] u_2 \\ &+ \left[-a_2 g(w_3) A_{2,3}^{(2)} + c_2 B_{2,3}^{(2)}\right] u_3 ; \end{aligned} \tag{2.116}$$

In the equations above, we have used the convenient notation:

$$\begin{array}{rr} 1/L\Delta_1 = a_1\,; & 1/L\Delta_2 = a_2\,; \\ D_0/L\Delta_1 = b_1\,; & D_0/L\Delta_2 = b_2\,; \\ D_0/L^2\Delta_1^2 = c_1\,; & D_0/L^2\Delta_2^2 = c_2\,; \end{array} \tag{2.117}$$

and the function:

$$g(x) \;=\; \frac{(1-nx)}{x}\,G(x) = \,(1-nx)\,v_o\,e^{-nx}\,. \tag{2.118}$$

We see from Eqns.(2.114-2.116) above that their right hand sides contain terms involving both the $\{w_i\}$, and the $\{u_i\}$, which are the unknown concentrations at the element boundaries (nodes). The easiest way to solve the DAEs is to eliminate the $\{u_i\}$, by writing them in terms of the $\{w_i\}$. This can be done if we can explicitly relate the $\{u_i\}$ and the $\{w_i\}$. We can try to do this using the algebraic relations, Eqns.(2.108) and Eqns.(2.110), describing the boundary conditions and continuity at the elements.

We can write Eqns.(2.108) and the second of Eqns.(2.110) in terms of the $\{w_i\}$ and $\{u_i\}$. Re-ordering the equations, we have :

$$\begin{aligned} b_1 A_{0,1}^{(1)} w_1 &= \left(v_0 e^{-nu_1} - b_1 A_{0,0}^{(1)}\right) u_1 - b_1 A_{0,2}^{(1)} u_2\,; \\ -a_1 A_{2,1}^{(1)} w_1 + a_2 A_{0,1}^{(2)} w_2 + a_2 A_{0,2}^{(2)} w_3 &= a_1 A_{2,0}^{(1)} u_1 + (a_1 A_{2,2}^{(1)} - a_2 A_{0,0}^{(2)})\, u_2 - a_2 A_{0,3}^{(2)} u_3\,; \\ b_2 A_{3,1}^{(2)} w_2 + b_2 A_{3,2}^{(2)} w_3 &= -b_2 A_{3,0}^{(2)} u_2 + \left(v_0 e^{-nu_3} - b_2 A_{3,3}^{(2)}\right) u_3\,. \end{aligned} \tag{2.119}$$

We can simply write the above equations together in matrix form, if we linearize the exponential terms in the boundary conditions. That is, as a first approximation, we replace $(e^{-nu_1},\, e^{-nu_3})$ by 1, the first term in their Taylor expansion. We can then write the matrix equation:

$$PW = QU\,. \tag{2.120}$$

The matrices P and Q are given by:

$$P \equiv \left[\begin{array}{ccc} b_1 A_{0,1}^{(1)} & 0 & 0 \\ -a_1 A_{2,1}^{(1)} & a_2 A_{0,1}^{(2)} & a_2 A_{0,2}^{(2)} \\ 0 & b_2 A_{3,1}^{(2)} & b_2 A_{3,2}^{(2)} \end{array}\right]. \tag{2.121}$$

Note that P has the dimension $[\,(N_E+1)\times N_C\,]$. Now Q has the form:

$$Q \equiv \left[\begin{array}{ccc} (v_0 - b_1 A_{0,0}^{(1)}) & -b_1 A_{0,2}^{(1)} & 0 \\ a_1 A_{2,0}^{(1)} & (a_1 A_{2,2}^{(1)} - a_2 A_{0,0}^{(2)}) & -a_2 A_{0,3}^{(2)} \\ 0 & -b_2 A_{3,0}^{(2)} & (v_0 - b_2 A_{3,3}^{(2)}) \end{array}\right]. \tag{2.122}$$

Notice that Q is a square tridiagonal matrix of dimension $[\,(N_E+1)\times(N_E+1)\,]$.

We can solve for U from Eq.(2.120):

$$U = Q^{-1} PW\,; \tag{2.123}$$

or, defining a matrix $R \equiv Q^{-1}P$ we have:

$$U = RW\,. \tag{2.124}$$

This is the required equation relating U and W. We first obtain R, and can then substitute for the $\{u_i\}$ appearing in Eqns.(2.114-2.116) in terms of the $\{w_i\}$, thus yielding a coupled system of DAEs in the $\{w_i\}$. Solving these, together with the initial conditions given by Eq.(2.113), we obtain W. We then obtain U explicitly from Eq.(2.124). Thus W and U together constitute the full solution of the PDE.

An example for continuous sedimentation

To illustrate the continuous sedimentation case, we consider the case of the effluent being pumped out. The other case of the effluent overflow can be treated identically.

we divide the domain of interest $[0,1]$ into *three* elements, by defining four nodes $\{H_1 = 0, H_2 = 0.3, H_3 = 0.4, H_4 = 1.0\}$. We locate the rectangular unit pulse representing the delta function source on the second element. We locate one interior collocation point each on the first and second elements, and two on the third. The parameters of the problem are therfore:

$$\begin{aligned}
\alpha &= 0; \\
\beta &= 0; \\
N_E &= 3; \\
i &= 1, 2, 3 \\
\Delta_1 &= (H_2 - H_1) = 0.3; \\
\Delta_2 &= (H_3 - H_2) = 0.1 = 2\eta; \\
\Delta_3 &= (H_4 - H_3) = 0.6; \\
N(1) &= 1; \\
N(2) &= 1; \\
N(3) &= 2; \\
N_C &= 4; \\
N_T &= 10;
\end{aligned} \tag{2.125}$$

Now the appropriate Jacobi polynomials are, respectively for the three elements,
$(G_1(0,0,x), G_1(0,0,x), G_2(0,0,x))$. We have already evaluated these polynomials and their roots for the batch case.

The collocation points $\{\xi_j^{(i)}\}$ are given by:

$$\begin{aligned}
\xi_0^{(1)} &= 0.0; \\
\xi_1^{(1)} &= 0.5; \\
\xi_2^{(1)} &= 1.0; \\
\xi_0^{(2)} &= 0.0; \\
\xi_1^{(2)} &= 0.5; \\
\xi_2^{(2)} &= 1.0; \\
\xi_0^{(3)} &= 0.0; \\
\xi_1^{(3)} &= 0.21132; \\
\xi_2^{(3)} &= 0.78868; \\
\xi_3^{(3)} &= 1.0;
\end{aligned} \tag{2.126}$$

The corresponding Lagrange polynomials follow:

$$p_1(\xi^{(1)}) = (\xi^{(1)} - 0.0)\,(\xi^{(1)} - 0.5)\,(\xi^{(1)} - 1.0); \tag{2.127}$$

$$p_2(\xi^{(2)}) = (\xi^{(2)} - 0.0)\,(\xi^{(2)} - 0.5)\,(\xi^{(2)} - 1.0)\,; \tag{2.128}$$

and

$$p_3(\xi^{(3)}) = (\xi^{(3)} - 0.0)\,(\xi^{(3)} - 0.21132)\,(\xi^{(3)} - 0.78868)\,(\xi^{(3)} - 1.0)\,. \tag{2.129}$$

The collocation matrices are now given by:

$$A^{(1)} = A^{(2)} \equiv \begin{bmatrix} -3.0 & 4.0 & -1.0 \\ -1.0 & 0.0 & 1.0 \\ 1.0 & -4.0 & 3.0 \end{bmatrix}\,; \tag{2.130}$$

$$B^{(1)} = B^{(2)} \equiv \begin{bmatrix} 4.0 & -8.0 & 4.0 \\ 4.0 & -8.0 & 4.0 \\ 4.0 & -8.0 & 4.0 \end{bmatrix}\,; \tag{2.131}$$

$$A^{(3)} \equiv \begin{bmatrix} -7.00000 & 8.19615 & -2.19615 & 1.00000 \\ -2.73205 & 1.73205 & 1.73205 & -0.73205 \\ 0.73205 & -1.73205 & -1.73205 & 2.73205 \\ -1.00000 & 2.19615 & -8.19615 & 7.00000 \end{bmatrix}\,; \tag{2.132}$$

$$B^{(3)} \equiv \begin{bmatrix} 24.00000 & -37.17691 & 25.17691 & -12.00000 \\ 16.39230 & -24.00000 & 12.00000 & -4.39230 \\ -4.39230 & 12.00000 & -24.00000 & 16.39230 \\ -12.00000 & 25.17691 & -37.17691 & 24.00000 \end{bmatrix}\,. \tag{2.133}$$

We can now write down explicitly the DAEs for the concentration at the $N_C = 4$ interior collocation points, Eqns.(2.69,2.70, 2.71).

For the collocation point $\xi_1^{(1)}$ within the first element, we have:

$$\frac{dX_1^{(1)}(t)}{dt} = \sum_{k=0}^{2} \left[-\frac{1}{L\Delta_1} \left(\left(1 - nX_1^{(1)}(t)\right)\, v_0\, e^{-nX_1^{(1)}(t)} + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A} \right) A_{1,k}^{(1)} + \frac{D_0}{L^2\Delta_1^2} B_{1,k}^{(1)} \right] X_k^{(1)}(t). \tag{2.134}$$

For the collocation point $\xi_1^{(2)}$ within the second element, we have:

$$\begin{aligned} \frac{dX_1^{(2)}(t)}{dt} \;=\; & \sum_{k=0}^{2} \left[-\frac{1}{L\Delta_2} \left(\left(1 - nX_1^{(2)}(t)\right)\, v_0\, e^{-nX_1^{(2)}(t)} + \frac{Q_u(t)}{A} \right) A_{1,k}^{(2)} + \frac{D_0}{L^2\Delta_2^2} B_{1,k}^{(2)} \right] X_k^{(2)}(t) \\ & - \left(X_1^{(2)}(t) - X_f(t) \right) \frac{Q_f(t)}{A} \frac{1}{2\eta L}. \end{aligned} \tag{2.135}$$

At the two interior collocation points $(\xi_1^{(3)}, \xi_2^{(3)})$, we have the DAEs:

$$\frac{dX_1^{(3)}(t)}{dt} = \sum_{k=0}^{3} \left[-\frac{1}{L\Delta_3} \left(\left(1 - nX_1^{(3)}(t)\right)\, v_0\, e^{-nX_1^{(3)}(t)} + \frac{Q_u(t)}{A} \right) A_{1,k}^{(3)} + \frac{D_0}{L^2\Delta_3^2} B_{1,k}^{(3)} \right] X_k^{(3)}(t)\,; \tag{2.136}$$

$$\frac{dX_2^{(3)}(t)}{dt} = \sum_{k=0}^{3} \left[-\frac{1}{L\Delta_3} \left(\left(1 - nX_2^{(3)}(t)\right)\, v_0\, e^{-nX_2^{(3)}(t)} + \frac{Q_u(t)}{A} \right) A_{2,k}^{(3)} + \frac{D_0}{L^2\Delta_3^2} B_{2,k}^{(3)} \right] X_k^{(3)}(t)\,. \tag{2.137}$$

The four DAEs above (Eqns.(2.134)-(2.137)) must be solved together with the initial conditions:

$$\begin{aligned} X_1^{(1)}(t=0) &= X_{10}^{(1)} = X_0\,; \\ X_1^{(2)}(t=0) &= X_{10}^{(2)} = X_0\,; \\ X_1^{(3)}(t=0) &= X_{10}^{(3)} = X_0\,; \\ X_2^{(3)}(t=0) &= X_{20}^{(3)} = X_0\,. \end{aligned} \tag{2.138}$$

The two boundary conditions (at $z = 0, L$) are now given by the equations:

$$\begin{aligned} \frac{D_0}{L\Delta_1}\sum_{k=0}^{2} A_{0,k}^{(1)} X_k^{(1)}(t) &= 0\,; \\ \frac{D_0}{L\Delta_3}\sum_{k=0}^{3} A_{3,k}^{(3)} X_k^{(3)}(t) &= 0\,. \end{aligned} \tag{2.139}$$

Expanding them, we have:

$$\begin{aligned} \frac{D_0}{L\Delta_1}\left(A_{0,0}^{(1)}X_0^{(1)}(t) + A_{0,1}^{(1)}X_1^{(1)}(t) + A_{0,2}^{(1)}X_2^{(1)}(t)\right) &= 0\,; \\ \frac{D_0}{L\Delta_3}\left(A_{3,0}^{(3)}X_0^{(3)}(t) + A_{3,1}^{(3)}X_1^{(3)}(t) + A_{3,2}^{(3)}X_2^{(3)}(t) + A_{3,3}^{(3)}X_3^{(3)}(t)\right) &= 0\,. \end{aligned} \tag{2.140}$$

Finally, we write the element boundary conditions. We now have two pairs, since there are two interior nodes (H_2, H_3):

$$\begin{aligned} X_0^{(2)}(t) &= X_2^{(1)}(t)\,; \\ X_0^{(3)}(t) &= X_2^{(2)}(t)\,; \\ \frac{1}{L\Delta_2}\sum_{k=0}^{2} A_{0,k}^{(2)} X_k^{(2)}(t) &= \frac{1}{L\Delta_1}\sum_{k=0}^{2} A_{2,k}^{(1)} X_k^{(1)}(t) \\ \frac{1}{L\Delta_3}\sum_{k=0}^{3} A_{0,k}^{(3)} X_k^{(3)}(t) &= \frac{1}{L\Delta_2}\sum_{k=0}^{2} A_{2,k}^{(2)} X_k^{(2)}(t)\,. \end{aligned} \tag{2.141}$$

Expanding the last two of the equations above, we have:

$$\begin{aligned} &\frac{1}{L\Delta_2}\left(A_{0,0}^{(2)}X_0^{(2)}(t) + A_{0,1}^{(2)}X_1^{(2)}(t) + A_{0,2}^{(2)}X_2^{(2)}(t)\right) = \\ &\quad \frac{1}{L\Delta_1}\left(A_{2,0}^{(1)}X_0^{(1)}(t) + A_{2,1}^{(1)}X_1^{(1)}(t) + A_{2,2}^{(1)}X_2^{(1)}(t)\right)\,; \end{aligned}$$

and:

$$\begin{aligned} &\frac{1}{L\Delta_3}\left(A_{0,0}^{(3)}X_0^{(3)}(t) + A_{0,1}^{(3)}X_1^{(3)}(t) + A_{0,2}^{(3)}X_2^{(3)}(t) + A_{0,3}^{(3)}X_3^{(3)}(t)\right) = \\ &\quad \frac{1}{L\Delta_2}\left(A_{2,0}^{(2)}X_0^{(2)}(t) + A_{2,1}^{(2)}X_1^{(2)}(t) + A_{2,2}^{(2)}X_2^{(2)}(t)\right)\,. \end{aligned} \tag{2.142}$$

Thus for the continuous case, the set of $N_T = 10$ equations we have to solve for the ten unknown concentration values $\{X_0^{(1)}(t), \ldots X_3^{(3)}(t)\}$ are: the four DAEs - Eqns.(2.134)-(2.137) with the four initial conditions - Eqns.(2.138) ; the two boundary conditions given by Eqns.(2.140) and the four element continuity conditions, Eqns.(2.141).

Matrix formulation

We shall again solve the DAEs for the continous case, Eqns.(2.134-2.137) using matrices. The vectors of unknown concentrations at the interior collocation points and at the nodes, W and U are:

$$W \equiv \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \equiv \begin{bmatrix} X_1^{(1)}(t) \\ X_1^{(2)}(t) \\ X_1^{(3)}(t) \\ X_2^{(3)}(t) \end{bmatrix} \tag{2.143}$$

and

$$U \equiv \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \equiv \begin{bmatrix} X_0^{(1)}(t) \\ X_2^{(1)}(t) = X_0^{(2)}(t) \\ X_2^{(2)}(t) = X_0^{(3)}(t) \\ X_3^{(3)}(t) \end{bmatrix} . \tag{2.144}$$

The vector of initial values at the interior collocation points is:

$$W^0 \equiv \begin{bmatrix} w_1^0 \\ w_2^0 \\ w_3^0 \\ w_4^0 \end{bmatrix} \equiv \begin{bmatrix} X_1^{(1)}(t=0) \\ X_1^{(2)}(t=0) \\ X_1^{(3)}(t=0) \\ X_2^{(3)}(t=0) \end{bmatrix} \tag{2.145}$$

The DAEs are written for the $\{w_i\}$:

$$\begin{aligned} \frac{dw_1}{dt} \;=\; & \left[-a_1 \left(g(w_1) + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A} \right) A_{1,1}^{(1)} + c_1 B_{1,1}^{(1)} \right] w_1 \\ & + \left[-a_1 \left(g(w_1) + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A} \right) A_{1,0}^{(1)} + c_1 B_{1,0}^{(1)} \right] u_1 \\ & + \left[-a_1 \left(g(w_1) + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A} \right) A_{1,2}^{(1)} + c_1 B_{1,2}^{(1)} \right] u_2 ; \end{aligned} \tag{2.146}$$

$$\begin{aligned} \frac{dw_2}{dt} \;=\; & \left[-a_2 \left(g(w_2) + \frac{Q_u(t)}{A} \right) A_{1,1}^{(2)} + c_2 B_{1,1}^{(2)} \right] w_2 \\ & + \left[-a_2 \left(g(w_2) + \frac{Q_u(t)}{A} \right) A_{1,0}^{(2)} + c_2 B_{1,0}^{(2)} \right] u_2 \\ & + \left[-a_2 \left(g(w_2) + \frac{Q_u(t)}{A} \right) A_{1,2}^{(2)} + c_2 B_{1,2}^{(2)} \right] u_3 \\ & - (w_2 - X_f(t)) \frac{Q_f(t)}{2\eta L A} ; \end{aligned} \tag{2.147}$$

$$\begin{aligned} \frac{dw_3}{dt} \;=\; & \left[-a_3 \left(g(w_3) + \frac{Q_u(t)}{A} \right) A_{1,1}^{(3)} + c_3 B_{1,1}^{(3)} \right] w_3 \\ & + \left[-a_3 \left(g(w_3) + \frac{Q_u(t)}{A} \right) A_{1,2}^{(3)} + c_3 B_{1,2}^{(3)} \right] w_4 \\ & + \left[-a_3 \left(g(w_3) + \frac{Q_u(t)}{A} \right) A_{1,0}^{(3)} + c_3 B_{1,0}^{(3)} \right] u_3 \\ & + \left[-a_3 \left(g(w_3) + \frac{Q_u(t)}{A} \right) A_{1,3}^{(3)} + c_3 B_{1,3}^{(3)} \right] u_4 ; \end{aligned} \tag{2.148}$$

and:

$$\begin{aligned}\frac{dw_4}{dt} &= \left[-a_3\left(g(w_4)+\frac{Q_u(t)}{A}\right)A^{(3)}_{2,1}+c_3B^{(3)}_{2,1}\right]w_3\\ &+\left[-a_3\left(g(w_4)+\frac{Q_u(t)}{A}\right)A^{(2)}_{2,2}+c_3B^{(3)}_{2,2}\right]w_4\\ &+\left[-a_3\left(g(w_4)+\frac{Q_u(t)}{A}\right)A^{(3)}_{2,0}+c_3B^{(3)}_{2,0}\right]u_3\\ &+\left[-a_3\left(g(w_4)+\frac{Q_u(t)}{A}\right)A^{(3)}_{2,3}+c_3B^{(3)}_{2,3}\right]u_4\,; \end{aligned} \tag{2.149}$$

In the equations above, we have used the convenient notation:

$$\begin{array}{lll} 1/L\Delta_1=a_1\,; & 1/L\Delta_2=a_2 & 1/L\Delta_3=a_3\,;\\ D_0/L^2\Delta_1^2=c_1\,; & D_0/L^2\Delta_2^2=c_2 & D_0/L^2\Delta_3^2=c_3\,;\end{array} \tag{2.150}$$

and the function $g(x)$ is as defined before for the batch case:

$$g(x) = (1-nx)\,v_o\,e^{-nx}; \tag{2.151}$$

The algebraic equations resulting from the boundary conditions and the element continuity conditions are as follows:

$$\begin{aligned} A^{(1)}_{0,1}w_1 &= -A^{(1)}_{0,0}u_1-A^{(1)}_{0,2}u_2\,;\\ -a_1A^{(1)}_{2,1}w_1+a_2A^{(2)}_{0,1}w_2 &= a_1A^{(1)}_{2,0}u_1+(a_1A^{(1)}_{2,2}-a_2A^{(2)}_{0,0})\,u_2-a_2A^{(2)}_{0,2}u_3\,;\\ -a_2A^{(2)}_{2,1}w_2+a_3A^{(3)}_{0,1}w_3+a_3A^{(3)}_{0,2}w_4 &= a_2A^{(2)}_{2,0}u_2+(a_2A^{(2)}_{2,2}-a_3A^{(3)}_{0,0})\,u_3-a_3A^{(3)}_{0,3}u_4\,;\\ A^{(3)}_{3,1}w_3+A^{(3)}_{3,2}w_4 &= -A^{(3)}_{3,0}u_3-A^{(3)}_{3,3}u_4\,. \end{aligned} \tag{2.152}$$

The equations above are all linear. Thus the matrices P and Q are given by:

$$P\equiv\begin{bmatrix} A^{(1)}_{0,1} & 0 & 0 & 0\\ -a_1A^{(1)}_{2,1} & a_2A^{(2)}_{0,1} & 0 & 0\\ 0 & -a_2A^{(2)}_{2,1} & a_3A^{(3)}_{0,1} & a_3A^{(3)}_{0,2}\\ 0 & 0 & A^{(3)}_{3,1} & A^{(3)}_{3,2}\end{bmatrix}. \tag{2.153}$$

And Q has the form:

$$Q\equiv\begin{bmatrix} -A^{(1)}_{0,0} & -A^{(1)}_{0,2} & 0 & 0\\ a_1A^{(1)}_{2,0} & (a_1A^{(1)}_{2,2}-a_2A^{(2)}_{0,0}) & -a_2A^{(2)}_{0,2} & 0\\ 0 & a_2A^{(2)}_{2,0} & (a_2A^{(2)}_{2,2}-a_3A^{(3)}_{0,0}) & -a_3A^{(3)}_{0,3}\\ 0 & 0 & -A^{(3)}_{3,0} & -A^{(3)}_{3,3}\end{bmatrix}. \tag{2.154}$$

We can thus find the matrix $R\equiv Q^{-1}P$, and hence obtain the required coupled set of DAEs in the $\{w_i\}$. Solving these, and using the relation $U=RW$, we finally obtain the full solution of the original PDE for continuous sedimentation with effluent pumped out, Eqns.(2.36-2.38).

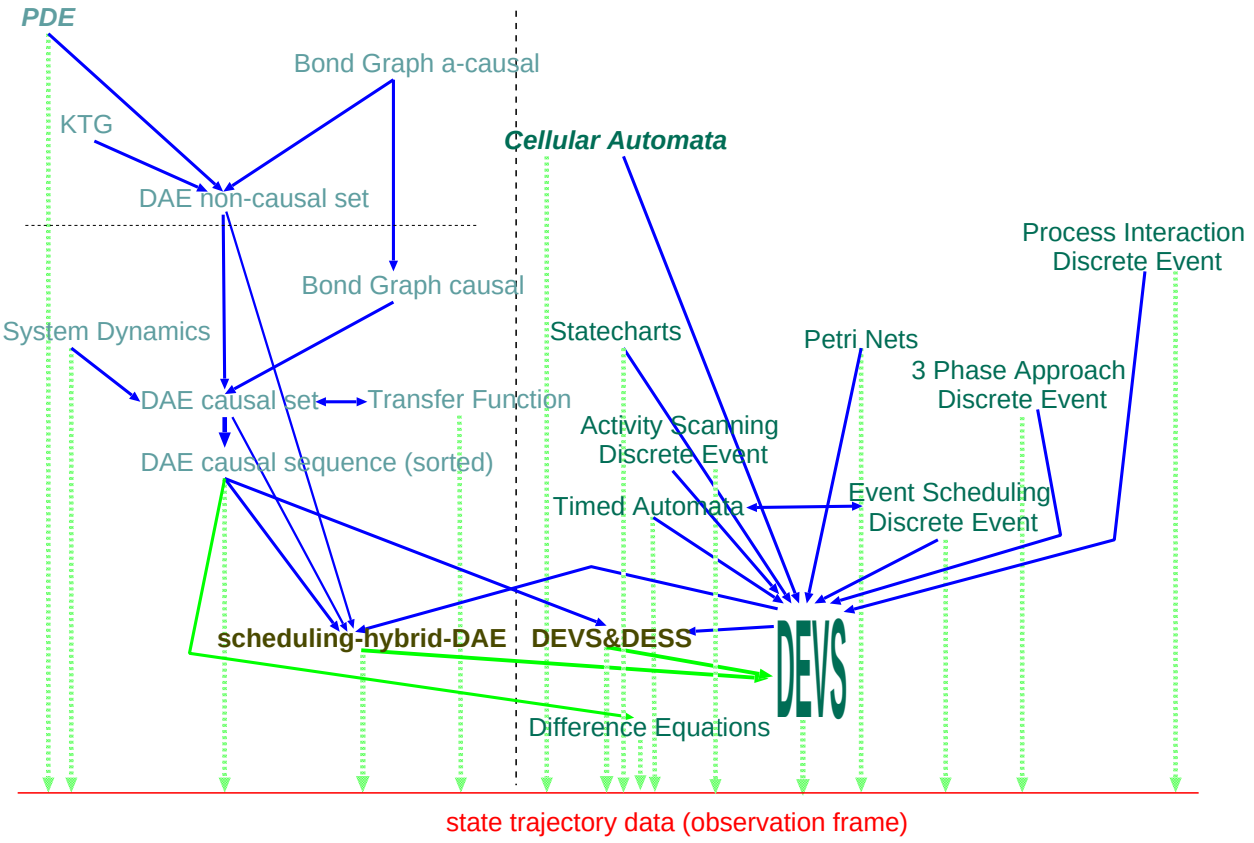

Figure 2.34: Formalism Transformation Graph

2.9 The Modular Network formalism

In this section we show how the semantics of multi-formalism networks may be given by transforming all components to a common formalism and then using the closure under coupling property of that formalism.

2.9.1 Formalism transformation

Based on the mathematical relationship between the System Dynamics and the ODE formalisms, translation of any model in the first formalism to a behaviourally equivalent model described in the second formalism is possible. In Figure 2.34, a part of "formalism space" is depicted again in the form of a Formalism Transformation Graph (FTG) introduced in the first chapter. The different formalisms are shown as nodes in the graph. The names of both the PDE and Cellular Automata formalisms are printed slanted, denoting the spatial distribution they incorporate. The vertical striped line in the middle denotes the distinction between continuous models (on the left) and discrete models (on the right). The well known Difference Equations formalism is often implicitly used in numerical simulators: ODEs are discretized by means of a suitable numerical scheme and the resulting difference equations are iteratively solved. Suitable refers to the nature of the equations as well as to the accuracy requirements. The arrows denote a homomorphic relationship "can be mapped onto", implemented as a symbolic transformation between formalisms. The vertical, dotted lines denote the existence of a *solver* or *simulation kernel* which is capable of simulating a model, thus generating a trajectory. A trajectory is really a model of the system in the data formalism (time/value tuples). In a denotational sense, traversing the graph makes *semantics* of models in formalisms explicit: the meaning of a model/formalism is given by mapping it onto some known formalism. This procedure can be applied iteratively to reach any desired (reachable) level. In an operational sense, a mapping describes how model interpretation can be achieved. If the "trajectory" formalism is the target of the mapping, model interpretation is model simulation. Though a multi-step mapping may seem cumbersome, it can be perfectly and

correctly performed by tools. The advantage of this approach is that the introduction of a new formalism only requires the description of the mapping onto the nearest formalism as well as the implementation of a translator to the latter formalism. It is often meaningful to introduce a new formalism for a specific application, encoding particular properties and constraints of the application. Often, translation involves some loss of information, though behaviour must obviously be conserved. This loss may be a blessing in disguise as it entails a reduction in complexity, leading to an increase in (simulation) performance. Usually, the aim of multi-step mapping is to eventually reach the trajectory level.

Another major use for formalism transformation is the *answering of particular questions* about the system. Some questions can only be answered in the context of a particular formalism. In case of a System Dynamics model for example, the visual inspection of the model can provide insight into influences. If the model is mapped onto a set of Algebraic and Ordinary Differential Equations, a dependency analysis may reveal algebraic dependency cycles not apparent at the System Dynamics level. At this same level, one may check whether parts of the model are linear. If so, these parts may be solved symbolically by means of computer algebra. Also, transformation to the Laplace domain (*i.e.,* to a Transfer Function form) opens avenues to a plethora of techniques for stability analysis. Finally, the transformation through numerical simulation to the data level allows for quantitative analysis of problems posed in initial value form. Note how the larger the number of intermediate formalisms, the higher the possibility for optimization along the way.

Above all, the traversal described above is the basis for the meaningful coupling of models described in different formalisms. This is discussed next.

2.9.2 Coupled model transformation

When we describe a structured model in the *network* or *coupled* formalism, we can only make meaningful assertions about its structure, its outside connections (its interface) and its components, not about its overall meaning or behaviour. Formally, a coupled model has the form

$$CM \equiv \langle id, interface, S, C \rangle$$

The model is identified by a unique identifier *id* (a name or reference). The $interface$ is a set of connectors or ports to the environment. Associated with the ports are allowed values as well as causality. Meaningful causalities are $\{in, out, inout\}$. The set S contains the sub-models (or at least their unique identifiers). The coupling information is contained in a graph structure C. For non-causal, continuous models, the graph is undirected. For causal models, the graph is directed. Obviously, a coupled model is only valid if types and causalities of connected ports are compatible. In certain cases, the graph may be annotated with extra information. In case of traditional discrete-event models, a tie-breaking function is usually required to select between simultaneous events [Zei84a].

If all sub-models are described in the *same* formalism F, it may be possible to replace the coupled model (at least conceptually) by one atomic model of type F. In this case, F is called *closed under coupling* (or under composition). The property often holds by construction. In case of Differential Algebraic Equations (DAEs), connections $\{connect(port_i, port_j)\}$ are replaced by algebraic $port_i = port_j$ coupling equations. Together with the sub-model equations, these form a DAE. In formalisms such as Bond Graphs, information about the physical nature of variables allows one to generate either the above type of equations in case of coupling of "across" variables (this corresponds to Kirchoff's voltage law in electricity) or an equation summing all connected values to zero for "through" variables (this corresponds to Kirchoff's current law in electricity). In discrete-event models, implementing closure involves the correct time-ordered *scheduling* of sub-model events. The most imminent event will always be processed first. The tie-breaking function is used to resolve conflicts due to simultaneous events (an artifact of the high level of abstraction).

If a coupled model consists of sub-models expressed in *different formalisms,* several approaches are possible:

- A *meta-formalism* can be used which subsumes the different formalisms of the sub-models. The different sub-models are thus described in a single formalism. The Hybrid DAE and DEVS&DESS [ZPK00] formalisms integrate continuous and discrete modelling constructs. Meaningful meta-formalisms which

truly add expressiveness as well as reduce complexity are rare. Bond Graphs are a good example of the integration of different domains (mechanical, electrical, hydraulic).

- Another approach is to *transform* the different sub-models to one *common formalism*. Which formalism to transform to depends on the questions asked. The closest common formalism for DAE and System Dynamics formalisms for example is the DAE formalism. By transforming a System Dynamics model to a set of DAEs, and using the closure property of the DAE formalism, it becomes possible to answer questions about the overall model.
- In the *co-simulation* approach, each of the sub-models is simulated with a formalism-specific simulator. Interaction due to coupling is resolved at the trajectory level. Compared to transformation to a common formalism before simulation, this approach, though appealing from a software engineering point of view (it is object-oriented) discards a lot of useful information. Questions can *only* be answered at the trajectory level. Furthermore, there are obvious speed and numerical accuracy problems for continuous formalisms in particular if one attempts to support non-causal models. The approach is meaningful mostly for discrete-event formalisms. In this realm, it is the basis of the DoD High Level Architecture (HLA) for simulator interoperability.

The transformation to a common formalism mentioned above proceeds as follows:

1. Start from a coupled multi-formalism model. Check consistency of this model (*e.g.*, whether causalites and types of connected ports match).
2. Cluster together all models described in the same formalism.
3. For each cluster, implement closure under coupling.
4. Look for the best common formalism in the Formalism Transformation Graph all the remaining different formalisms can be transformed to. In the worst case, this will be the trajectory level in which case the approach falls back to co-simulation. Which common formalism is best depends on a quality metric which can take into account transformation speed, potential for optimization, *etc.*
5. Transform all the sub-models to the common formalism.
6. Implement closure under coupling of the common formalism.

A side-effect of mapping onto a common formalism is the great potential for optimization of the flattened model, as well as the reduced number of (optimized) simulation kernels needed.

To describe which formalism transformations are possible, the Formalism Transformation Graph (FTG) mentioned above is used. A plethora of formalisms is depicted in Figure 2.34. Each of these has its own merits. Petri Nets are particularly suited for symbolic analysis (proof of dynamic properties) of concurrent systems. State Charts, an extension of Finite State Automata are a graphical formalism with a very appealing, intuitive semantics. In the UML, the State Chart formalism is used to specify the concurrent behaviour of software. Cellular Automata extend Finite State Automata with a (discretized) notion of space. As such, they are similar to Partial Differential Equations which add a spatial dimension to Ordinary Differential Equations. Apart from the formalism transformations described earlier, the central position of DEVS is striking. On the one hand, the expressiveness of DEVS makes many discrete-event formalisms DEVS-representable. Recently, it has been shown that continuous models can be quantized and described in the DEVS [ZL98] formalism. This mapping will be described further on. It allows one to meaningfully handle discrete/continuous multi-formalism models. Also, the potential for parallel implementation increases drastically [KSKP96].

2.9.3 Mapping the ODE formalism onto DEVS

Though still the subject of ongoing research, we briefly present the mapping of ODE models onto DEVS as this is deemed to be a novel way of bridging the gap between the “continuous” and the “discrete” realm.

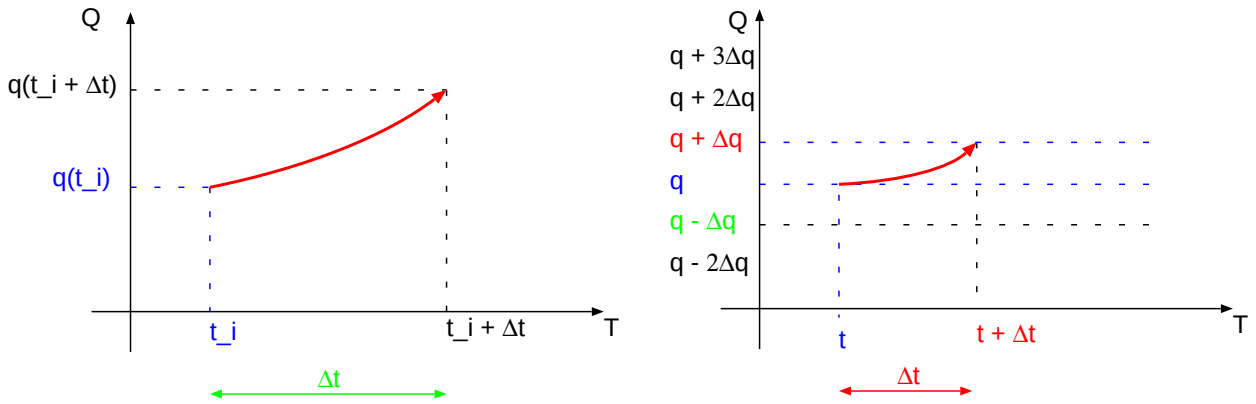

Figure 2.35: Time discretization *vs.* State-discretization

In the context of hybrid systems models, the formalism transformations in Figure 2.34 converge to a common denominator which unifies continuous and discrete constructs. Typically, this is some form of event-scheduling/state event locating/DAE formalism with its corresponding solver. A different approach, quantizing space rather than discretizing time, is presented here which maps continuous models, in particular algebraic and differential equations of the form

$$\begin{cases} \frac{dq}{dt} &= f(q,x,t), \quad q \in Q; \\ y(t) &= g(q,t), \quad y \in Y; \end{cases}$$

with $x(t) \in X$ a known input function, and initial conditions given by $q(0) = q_0$, onto Zeigler's DEVS formalism presented above. If the mapping is done appropriately, a discrete-event simulation of the DEVS model will yield a close approximation of the continuous model's continuous behaviour.

The normal approach of numerical mathematics is to approximate an ODE solution based on a Taylor expansion. Here, time is discretized, and subsequent state-variable approximations are calculated. Zeigler [ZL98] proposes to discretize the state variables and to calculate the corresponding approximate time-increases. The DEVS transition function (constructed from the ODE) will repeatedly go from one discretized state value to either the one just above or the one just below. The transition function will also calculate the time till the next discrete transition (possibly $+\infty$ if the derivative is zero). Both approaches are shown side by side in Figure 2.35. In mathematical terms, the model above is mapped onto a DEVS

$$atomicDEVS \equiv \langle \hat{S}, ta, \delta_{int}, \hat{X}, \delta_{ext}, \hat{Y}, \lambda \rangle.$$

$\hat{x} \in \hat{X}$, $\hat{q} \in \hat{Q}$ and $\hat{y} \in \hat{Y}$ are the quantized variables. The simple quantization used here is based on a grid of quanta (Δ_x, Δ_q, Δ_y). Note how each of the quanta are hypercubes. The quantized state set $\hat{S} = \{(\hat{q},\hat{x},t) | \hat{q} \in \hat{Q}, \hat{x} \in \hat{X}, t \in T\}$. A memory of input and absolute time is kept in the DEVS model. The internal transition function is

$$\delta_{int}((\hat{q},\hat{x},t)) = (\hat{q} + sgn(f(\hat{q},\hat{x},t))\Delta_q, \hat{x}, t + ta(\hat{q},\hat{x},t)).$$

The time advance function

$$ta((\hat{q},\hat{x},t)) = \left| \frac{\Delta_q}{f(\hat{q},\hat{x},t)} \right|$$

specifies after how much time the trajectory will leave the quantum hypercube. The external transition function describes how autonomous (integration) behaviour can be interrupted by an external input event (the input function exceeding a quantum boundary)

$$\delta_{ext}((\hat{q},\hat{x},t),e,\hat{x}') = (\hat{q},\hat{x}',t+e).$$

Note how ignoring the change in q (not $\hat{q}$) during e is a rough approximation and a better approach is to internally keep track of the non-quantized value q. In case an internal *and* external transition occur simultaneously, the confluent transition function of parallel DEVS describes how internal transition and external input are both taken into account:

$$\delta_{confl}((\hat{q},\hat{x},t),\hat{x}') = (\hat{q} + sgn(f(\hat{q},\hat{x},t))\Delta_q, \hat{x}', t).$$

Quantized output is obtained after application of the output function

$$\lambda((\hat{q},\hat{x},t)) = \lfloor g(\hat{q},t)/\Delta_y \rfloor.$$

Coupling of the thus obtained atomic DEVS models into a coupled DEVS provides a means for –possibly parallel– simulation of hierarchically coupled continuous models. To achieve maximum performance, equations should first be symbolically manipulated, and tightly coupled sets of equations must be clustered inside an atomic DEVS.

Summary

In this chapter, the structure of diverse formalisms was presented. These formalisms are nodes in the Formalism Transformation Graph introduced in the previous chapter.

The first formalism consists of trajectories. Simulators for specific higher level formalisms transform a model specification into a model specification at this Data level. The class of discrete event formalisms and the different world views in common use were presented in some detail. In particular, a rigourous description of the event scheduling world view was given. The DEVS formalism was introduced as a basis for the description and simulation of all discrete event (and even discrete) formalisms. The rigourous description of the event scheduling world view served as a basis for mapping that formalism onto the DEVS formalism. When spatial distribution of the state-space is introduced in the form of cells and the dynamics of the system (in the form of a transition function) is limited to interaction between neighbouring cells, one obtains the Cellular Automata formalism. This formalism was described and its mapping onto the DEVS formalism explained. In the continuous realm, Differential and Algebraic Equation (DAE) formalisms were presented. In particular, the use of and conversion between non-causal set, causal set, and causal sequence models was described. The first transformation is achieved by means of "causality assignment" based on Dinic's network flow algorithm applied to an equation-variable dependency graph. The second transformation, "sorting", is based on a depth first search of the dependency graph. The Transfer Function formalism, popular in control theory, and its transformation onto the Differential Equation formalism was presented next. Still in the continuous realm, Forrester's System Dynamics formalism is defined in terms of the Differential Equation formalism and is mapped onto it. It is shown how judicious representation in the modelling language MSL-USER alleviates the need for explicit transformation. Introduction of spatial distribution in continuous models leads to the Partial Differential Equation (PDE) formalism. For a limited class of one dimensional PDEs, it was shown how discretization by means of orthogonal collocation over finite elements allows transformation of PDEs to the DAE formalism (and to Ordinary Differential Equations in particular) to be performed automatically. This was demonstrated for the specific case of sedimentation in waste water treatment. To show how models in different formalisms can meaningfully be combined, the network formalism, coupling model components in a hierarchical fashion, was presented. In particular, a flattening algorithm for multi-formalism coupled models, was introduced, based on formalism transformation to a common formalism. Finally, to bridge the gap between continuous and discrete event models, a transformation between ODE models and DEVS models was introduced.

3

A Generic Modelling and Simulation Architecture

In this chapter, we start from the vision of a *Generic Modelling and Simulation Architecture* (GMSA). Though open simulation architectures have been proposed in the literature, the modelling aspect has been neglected. The GMSA addresses the need for an open architecture fom modelling as well as simulation.

The entities central to the modelling process are obviously models. To allow for the representation, and subsequent re-use and exchange of these models, a *modelling language* is needed. The design of the language MSL-USER is introduced in this chapter. MSL-USER is an ongoing effort which will evolve in the direction of meta modelling. The concept of *meta modelling* and its advantages over the use of a single super-language are presented here.

The ideas of this and previous chapters have been implemented in the WEST++ *interactive modelling and simulation environment.* Though the focus of WEST++ is on continuous formalisms, the design is general and allows for the later introduction of other formalisms. The highest level of the WEST++ design is presented.

3.1 The Generic Modelling and Simulation Architecture vision

The current proliferation of network technology and applications provides an ideal starting point for distributed modelling and simulation environments. The required basic hardware (networking and workstations) as well as software (distributed objects) infrastructure has become available to the intended users of a Generic Modelling and Simulation Architecture (GMSA). This is in contrast with the recent (1993) limited availability of enabling technology, when the author proposed a Framework for Concurrent Simulation Engineering (CSE) [VLRV93], which is the precursor of the current GMSA. Whereas at that time, only privileged labs had the necessary infrastructure to implement a CSE, any PC user connected to an IP network can now participate in a global modelling and simulation effort.

The High Level Architecture (HLA) provides a sound basic software architecture and methodology for cooperation between simulators. In accordance with the DoD Modeling and Simulation Master Plan (DoD 5000.59-P, dated October 1995), the Defense Modeling and Simulation Office (DMSO) is leading a DoD-wide effort to establish a Common Technical Framework to facilitate the interoperability of all types of models and simulations among themselves as well as to facilitate the re-use of M&S components. This Common Technical Framework includes the High Level Architecture.

An equivalent, standardized architecture in the distributed modelling realm does, to our knowledge, not yet exist. It therefore seems natural to design a distributed software architecture based on the methodological issues presented before. On the one hand, this is a much harder problem than that addressed by the HLA:

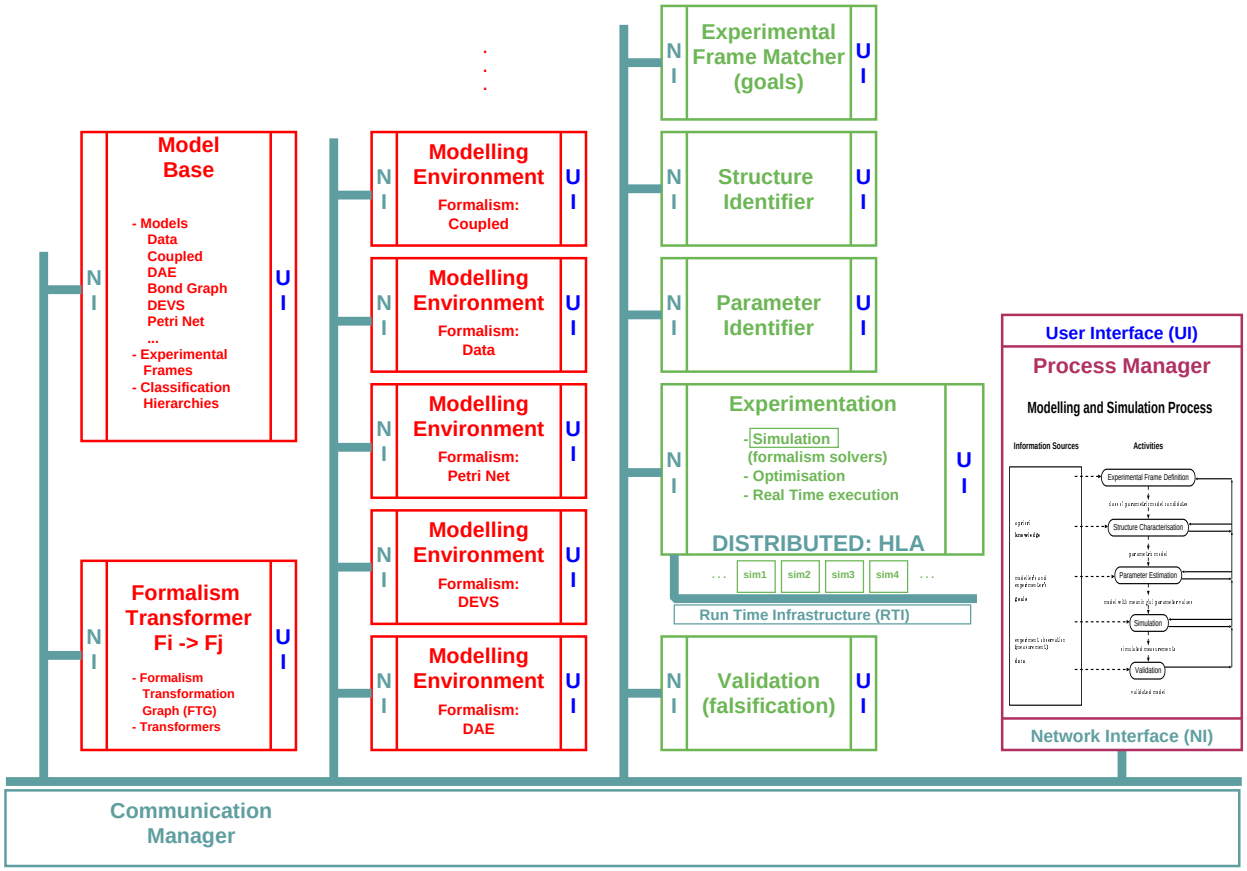

Figure 3.1: Generic Modelling and Simulation Architecture

the problem of *meaningful* model re-use and exchange. On the other hand, the correct coupling of different simulators, each with its own virtual time as in the HLA, does not occur so extensively in the context of distributed modelling: for all practical purposes, the modelling-related components of a GMSA have no "dynamics".

In Figure 3.1, a high-level description is proposed for a Generic Modelling and Simulation Architecture (GMSA). The figure presents an "object" view, in which each of the components represents a software object [OHE96], with its proper dynamic behaviour. The internal behaviour is quite complex, with sporadic interaction with other objects. The backbone of this interaction is the *Communication Manager (CM)*, which provides reliable, consistent communication between objects. The CM takes on the role of the Run Time Infrastructure (RTI) in the HLA. The CM consistency requirement is far easier to fulfil than the equivalent requirement (correct ordering of timestamped messages) in the RTI. The core problem here is to ensure *semantic consistency.*

All objects in the GMSA have a Network Interface (NI), taking care of network communication, managed by the Communication Manager. Each of the GMSA objects may have a User Interface (UI). Thanks to current client/server technology (such as X11), a user may simultaneously interact with multiple GMSA objects —without even being aware of their actual location on the network— through their respective User Interfaces.

The following describes the different components of the GMSA. As Figure 3.1 gives a high level view, the possibility of each of the components to have a distributed implementation in its own right, is not depicted.

The Model Base is a repository for models. These models can be described in different formalisms. The Model Base is capable of holding *all* knowledge we have about reality (including raw data; it suffices to employ the appropriate formalism). To allow meaningful re-use of models, Experimental Frames

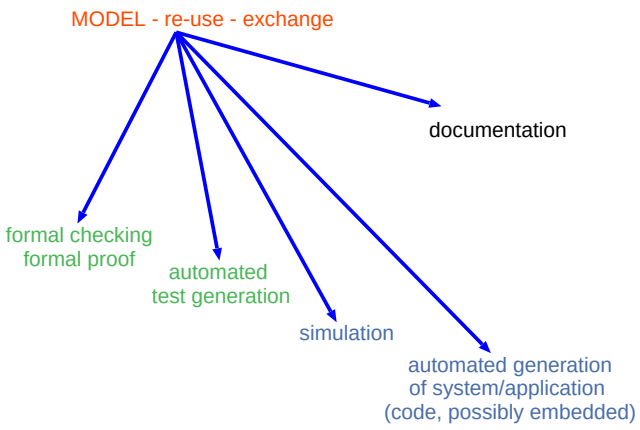

Figure 3.2: Model uses

(and their relationship to models) are also stored in the Model Base. Actually, in their most general form, Experimental Frames *are* themselves models. Finally, to allow automated model choosing, different classification hierarchies (such as inheritance trees or the System Entity Structure [Zei84a]) are kept in the Model Base.

The Formalism Transformer is the core of a Multi-Formalism modelling and simulation environment. Based on the information contained in the Formalism Transformation Graph (FTG), it activates the appropriate transformer to translate a model from one formalism into another. For certain formalisms, it also implements *closure*: given a coupled model with submodels which are all of formalism F, it generates a single equivalent flattened model in formalism F.

Modelling Environments implement the modelling operations inherent to a particular formalism. A good modelling environment is easy to use, intuitive, and often application/user specific. Above all, it unambiguously implements a formalism (the semantics). In practice, this means that all meaningful operations on models are supported and all meaningless ones lead to appropriate warnings. From the implementation point of view, some modelling environments may share software components such as a hierarchical editor. The Modelling Environments store/retrieve models in/from the Model Base and as such do not have local persistent memory.

The Process Manager implements the Modelling and Simulation Process as described before. It orchestrates the iterative traversal of the Experimental Frame matching, Structure Identification, Parameter Identification, Experimentation (in particular, simulation), and Validation phases. As such, it keeps track of the *global state* of the M&S enterprise. This is the top-level interface between the users and the GMSA.

Experimentation implements in essence the simulation kernels corresponding to the formalisms used in modelling. These simulation kernels “solve” the (execution level representation of) models built in the modelling environments. When multiple solvers are used, an HLA architecture is employed.

3.2 The MSL-USER modelling language

Models are at the core of any modelling and simulation system. To allow for manipulation, models need to be appropriately represented. Depending on the intended use and/or user, a model representation may need to satisfy different criteria. To optimally satisfy these different criteria, it is often useful to allow multiple representations of a single model, as will be discussed below. Figure 3.2 depicts the various uses of a model. Under certain conditions, a model’s correctness may be checked formally. If sufficient information is available, properties (such as the occurrence of deadlock in a telecommunication system) pertaining to the dynamic behaviour of a system may be proven. In many cases, formal proof is not possible. Test vectors

may however be generated automatically from the model. To investigate the behaviour space of the model under study, simulations of the model can be used. After formal checking, automated test generation, and dynamic simulation, the automated generation of the system-to-be-built (often software) from the very same model specification is desirable. Not in the least, a model is a form of documentation. A model is truly at the heart of re-use and exchange of knowledge about the dynamic behaviour of systems. Note how a limit case consists of non-dynamic, static models as they are often used in the design of non-real-time software.

3.2.1 Model Specification Language (MSL) requirements

Many issues are relevant when designing a Model Specification Language (MSL). The list of uses of a Model Specification Language given above leads to more concrete requirements.

Probably the most basic question to be answered is what information needs to be explicitly represented in an MSL.

A low-level example of this is `r = i`; with `r` a Real and `i` an Integer. We all internally coerce `i` to a Real and thus there is no need to explicitly write `r = (int)i`; In a modelling language, the same rationale should be used: if the user mentally assigns the "right" (*i.e.*, as given by the compiler, really by the language designers) semantics to a model, there is no need for explictly mentioning types, and/or introduce keywords, otherwise there is. It is of course likely that a clean, *orthogonal* design will automatically lead to the user "intuitively" understanding the meaning of the model, and far less "explicit" language constructs will be needed.

In all aspects of (modelling language) design, it is meaningful to traverse the following steps:

1. Mathematically describe the problem. This "formal specification" may not be ideal for the average user (and should only be used as a reference), but it is *unambiguous*; it can be used as the basis for formal proofs of statements about the specification, as the basis for implementation, and will usually enable an orthogonal design.

2. Some of the mathematical constructs may be given explicit names (a "segment" as described in the first chapter). These explicit names simplify discussion. As there is an underlying mathematical contruct to the name, the name is unequivocally defined.

3. While introducing a new concept into a modelling language, it should first of all be considered whether it is at all necessary. If the meaning is obvious to the user and the compiler can automatically infer the meaning (*i.e.*, type) in all cases, there is really no need to represent the concept explicitly. If the need to introduce new syntax does arise, this should be done in an orthogonal fashion. This implies that the meaning of the syntactic construct, when applied in different contexts (outside the intended one), should be investigated. Usually, this leads to deeper insight into the nature of concepts and may even lead to a modification and/or extension of the mathematical specification.

As a logical consequence of the above, it seems reasonable to express semantics in an MSL through the use of "types". In programming languages, types are a means to express certain "constraining" information about entities in a program [AC96]. Thanks to this information, it may be possible to guarantee safe/correct behaviour of a compiled program. This information is also used to generate efficient run-time code [ASU86]. One way of interpreting a type in a programming language is to see it as a "range" of values a variable of that type can take. Of course, this type information is a "coarse" boundary on the behaviour of that variable. For programming purposes (where, for example, time dependence does not play a role), such an approximation is usually sufficient (to infer safe/correct behaviour of the compiled program). From the point of view of designing a modelling language, it seems reasonable to try to encode as much "modelling knowledge" into types as possible. A (hoped for) consequence is that "model checking" reduces to a problem of "type checking" (and can be automated). There is one important realization when using programming language types for a modelling language. In the modelling world, the meaning of a model is inevitably related to its behaviour (possible after causality assignment and other transformations). At a low level, we interpret a type Real with a lowerbound and an upperbound as meaning that a variable can take any Real value between

lowerbound and upperbound (for the time being, assuming we're dealing with truly mathematical, infinite precision reals). Extending this intuitive reasoning, the meaning of the equation `x + y = 2` with `x` and `y` Reals is the *set of all values* (x,y) which satisfy this equation. Whether this set is eventually obtained purely through symbolic manipulation, by means of one or other numeric solver, or by a blend of these, does not matter (we are after all declaratively/denotationally modelling behaviour). As we can not, a priori, infer the set of all values of the variables in any model (that would mean it has to be solved), we can only determine a rough boundary on the values, and thus of the type of a whole model. When we talk about meaningful re-use and exchange of models, we mean we want to use a model in a certain context in such a way that its behaviour is what we need/expect. Checking/matching only the model interface type is a very coarse approximation. Such an approximation is usually sufficient in programming languages, but this may not be the case with models. Even in programming, it may not be sufficient and pre/post conditions are employed together with invariants in specifying program behaviour. These correspond partly to the Experimental Frame described in the first chapter. The main reason is that, though we write `x` is a Real, we implicitly mean `x` is a mapping, from the time set T into the set of Real values. The behaviour of the model is thus time-varying (in a "discrete event" or "continuous" way). The fact that there are different kinds of such relationships between time evolution and (state) variable evolution (depending on the nature of T and of the values) and the possible behaviours are trajectories in a multi-dimensional space implies that we should

1. Try to use "model types" to describe the different types of time-state relationships. In particular, this means not leaving time more or less implicit, but rather explicitly mention the type of time.

2. Use an Experimental Frame concept to encode the envelope boundary of allowed trajectories. An Experimental Frame is the context in which a model, to a certain degree of accuracy, accurately represents the behaviour of a system. Using the Experimental Frame, the correct use of a model in a certain context can be

   - checked at compile-time if the assertions can be symbolically evaluated;
   - embedded in the final code as assertions which will fail in case of incorrect use of the model.

We now describe the requirements for a Model Specification Language, in particular, the *goals* of types in a MSL:

1. To be capable of modelling (abstractions of behaviour of) *physical systems.* System models such as those described in the first chapter must at least be representable within the type theory. In particular, it must be possible to represent "time segments" [Zei84b] or trajectories as formalizations of the concept of time-variance (as opposed to variables as holders of values in programming languages, where time dependence is ignored).

   As a consequence of the "physical systems" requirement, it must be possible to represent attributes such as physical nature, units, and across/through. In the limit (by collapsing a continuous time base onto a discrete program counter time base) it must be possible to model *non-physical* systems such as software.

   As a generalization of traditional, causal physical system models, *non-causal* DAE (Differential Algebraic Equation) models [Cel91] must be representable.

2. To be capable of expressing *abstract, structured concepts* such as a set, product set, power set, record, function, *etc.* starting from basic types. The structuring constructs should allow the *complete* specification of abstract model constituents such as model interface and model parameters. Although some object-oriented systems (such as Smalltalk) treat types (actually, classes) as first class entities, *i.e.*, as objects (types, like objects, may be represented by finite collections of attributes-value pairs), types will not be considered as objects from the modeller's point of view. Obviously, from the type system implementer's point of view, types may be implemented as objects in an object-oriented environment.

3. To have *declarative sub-typing constructs* limiting a type (seen as a possible set of values).

4. To formalise *type equality* for the purpose of *type checking* [ASU86]. A type checking algorithm is the basis of checking and unifying types.

5. To satisfy general requirements:

   - Consistency, orthogonality: the same mechanisms (*e.g.*, sub-typing, type equality) must be used without exceptions, throughout the formalism.
   - Correctness: no internal contradictions may exist in the type system.
   - Exhaustive: cover *all* possible modelling needs (*i.e.*, it must be possible to describe the "type" of constructs in any modelling formalism).
   - Simplicity: elegant and minimal (may need syntactic glue later to make the type definition language user-friendly).
   - Extensibility: user-defined types must be allowed (*e.g.*, units).

3.2.2 Formalism versus language

The following presents the relationship between a formalism, its representation within a computer and its external representation, a modelling language.

In the design of any modelling language, it is essential to start from an underlying formalism. This formalism is the true essence of what the language stands for. The language is just an external representation. The language should:

- Reflect/express the underlying formalism in a *natural* way. Thus, the meaning or semantics of a model written in that language will be easily understood. Formally, the semantics of the language is expressed through mapping of the language structure onto the formalism.
- Be *simple* so as to not to be a burden for the modeller. This syntactic issue is not as crucial as the above semantic one, but may nevertheless make the difference between a usable and a non-usable language. Mainly, syntax should help the user to understand semantics in a "natural" way. Increasingly graphical modelling environments are used to hide syntactic complexity. In a sense, the graphical structure now replaces syntactic structure.

Figure 3.3 expresses the relationships between:

- A system formalism: For example, a state-based, general systems formalism.
- An internal representation: A *representation* (data-structure) as used in a computer program which tries to represent as closely as possible (one to one), the abstract mathematical entities from a system formalism. For reasons of performance or size, this representation may not be a perfect image of the formalism structures.
  Semantic rules to check compliance with the formalism (the formalism is more than just the data-structure). Also, the data structure together with semantic rules allows for meaninful knowledge exchange between heterogeneous environments. There need not be a shared a priori knowledge about formalisms, if one transmits both data structure and semantic rules. In checking compliance with a formalism, semantic rules will restrict the number of "valid" models. For example: in `x:= a + b`, the Left Hand Side (LHS) (`x`) must be a variable, not used on the LHS of an equation before.
  Note: One could "hide" these semantic checking rules in the parser. This would reduce the number of passes through the internal representation (*i.e.*, the Abstract Syntax Tree) thus speeding up the model checking phase. However, this also obfuscates the distinction between syntax and semantics. Furthermore, it would be hard to write a meta-description syntax as well as semantics of the formalism in the form of an MSL formalism CLASS.
  Semantic rules to map onto other formalisms (tranformation, see later).
  A *simulation kernel* for "solving" the model. If such a kernel exists, the model is called "concrete".
- One or more external representations: These are modelling languages defined by their particular syntax. Their semantics is defined through the relationship with the formalism (albeit represented in a

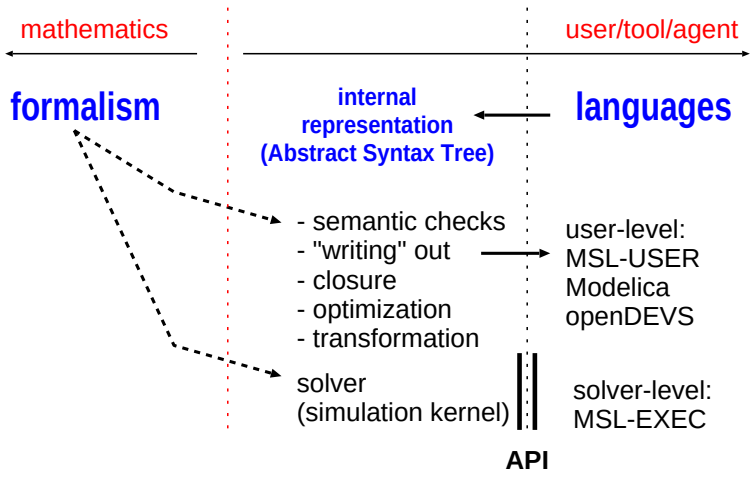

Figure 3.3: Relationship between formalism and representations

computer by means of the internal representation). This is done in terms of semantic mapping functions: which map an internal representation onto an external representation (code generation). In the opposite direction, for each language, a lexer, parser and semantic mapping description allow for transformation into the internal representation. In our implementation, the language MSL-USER is geared towards human users. The language MSL-EXEC, automatically generated from MSL-USER is meant to represent models at a level suitable for numerical simulation. It is linked with a numerical solver.

One can identify equivalence classes of languages corresponding to the same formalism. All the languages corresponding to one formalism are in a sense equivalent as they are all capable of expressing the same meaning (though their syntax may be widely different). The "language *L* can be mapped onto formalism *F*" defines the relationship through which the equivalence class "belongs to the same formalism" is defined.

For argument's sake we assume there is only one internal representation, which is not required. Assuming it to have a one-to-one correspondence with the formalism structure makes things easier and is probably good design (a software engineering consideration).

Obviously, within one abstraction, it will be possible to translate from one language into another by subsequent lexing/parsing to the internal representation followed by the appropriate code generation.

One clear requirement of this process is that there be no information loss. In particular, going from one language to the internal representation and back to that same language should not be lossy. This does not mean that input and output of that process should be syntactically identical however. As an example: if the underlying formalism is based on SET semantics, the order of equations does not matter. A more trivial example is the exact lexical layout such as the number of blanks and tabs of the model text.

All operations performed in a language should have their counterparts in the formalism. Transformations between languages corresponding to different formalisms should be linked to the respective transformations between formalisms. Rather than starting from a language syntax and a natural semantics and extendingit, our approach has been to start from the formalism and construct the language starting from there. Obviously, expressing semantics of a language thus constructed is far easier and correct.

It is important to consider not only the relationship of a formalism to different languages, but also to *simulation kernels* or *abstract interpreters* which are capable of *executing* the formalism. By executing, we mean the explicit traversal of the state-space of the model expressed in the formalism. The mechanism for doing so will vary widely with the formalism (*e.g.*, next event list, numerical integration, constraint propagation) For each formalism, a simulator (which may or may not be practical to implement in software/hardware) consists of a simulation *kernel* which takes a *model* description as well as initial values and generates from the full state trajectory. In certain contexts, a simulation kernel is called a solver or an interpreter. Simulation

kernels may be numerical or symbolic In either case, a backward link between the generated information and the model should be preserved (to retain semantic information).

A formalism is *concrete* if a kernel exists. If this is not the case, the formalism is *abstract*.

On the Abstract Syntax Tree implementation

The core of an MSL compiler is the Abstract Syntax Tree (AST). The *modularity* of the compiler is influenced by the *implementation style* used [App97]. In the Abstract Syntax Tree, different *kinds* of objects, also called nodes, are represented corresponding to the various syntactic structures in the language such as compound statements, assignment statements, *etc*. Different *interpretations* of these objects are possible: for type-checking, optimization (*e.g.*, constant folding), generation of MSL-EXEC code, and so on. Two styles of programming can be used:

Syntactic structure separate from interpretations This traditional style starts from an AST data structure consisting of nodes which are instances of a `union` type capturing all possible kinds of objects. For each interpretation, a function (with a union-type node as argument) describes exhaustively how to interpret each type of object. This interpretation is usually done recursively in terms of the node's sub-trees. It is easy and modular to add a new *interpretation*: the treatment of all *kinds* of objects is given in one place: the interpretation function (by means of an exhaustive `switch` statement).

Object-oriented In this style, each interpretation is just a method in all the classes. It is easy and modular to add a new kind of interpretation: all the interpretations of that kind are grouped together as methods of the new class. It is however not modular to add a new interpretation: a new method must be added to every class. Usually, the different kinds are fixed at the time of the design of the language (though in MSL, new kinds may be added as more formalisms are supported). Over time, new interpretations (semantic actions) will however frequently be added which makes an object-oriented style quite unusable.

The above rationale as well as experience with the pure object-oriented approach during the construction of the μCSL compiler [VVV90a], lead to the conclusion that we must adopt the "syntactic structure separate from interpretations" style of implementation for the AST. When adopting this style it is meaningful to separate the recursive traversal part from the semantic action part in the interpretation function implementation using a *visitor pattern* [GHJV95]. This makes code far more readable in case of all but a few AST nodes are just traversed without any semantic actions.

3.2.3 MSL-USER syntax and semantics

In the following, the syntax and semantics of the modelling language MSL-USER (version 3.1) is briefly given.

An MSL file (typically with extension `.msl`) consists of a sequence of statements. The statements are either declarations or objects.

MSL file

Syntax:

```
<MSL file> ::=
   <statements>
<statements> ::=
   <statements> <orientation> <statement>
<statement> ::=
   <declaration_statement>
 | <object>
<declaration_statement> ::=
```

```
| <type_decl_stmt>
| <class_decl_stmt>
| <obj_decl_stmt>
```

Semantics:

Syntactically, an MSL file consists of a *sequence* (order matters) of statements. Statements are either declarations or objects. Declarations can be of three types: TYPE, CLASS, or OBJect. Both at the top level (the MSL file level) and at lower levels, the `<orientation>` determines whether the sequence should be interpreted as a *row* (orientation separator is comma ",") or as a *column* (orientation separator is semicolon ";"). Depending on the context, the orientation distinction has a particular meaning. Currently, in a "vector" context, row and column have the usual meaning. In a "set" context, both are equivalent.
A valid MSL file may be *empty* as declarations may be empty.
In MSL, the *scope* of a declaration between {} covers the whole surrounding context. Thus, it is possible to refer to an entity before it is declared. At the top level, the surrounding context is the whole file. More details on scoping are given in the next section.

A TYPE declaration only specifies an abstract data type's signature, no (default) values. A CLASS attaches values (objects) to a TYPE. As a result of this definition, multiple classes can correspond to the same type [AC96]. An OBJect is an instance of a class or type. An object binds the different parts of a type structure to concrete values.

type declaration

Syntax:

```
<type declaration> ::= TYPE <type_name> <description> <type_decl>
```

Types can be declared by means of

- a type signature
- sub-typing of an existing type (subsumption)
- type extension

```
<type_decl> ::= <type_decl_signature>
              | <type_decl_subtyping>
              | <type_decl_extension>
```

Object:

To instantiate (define) an object of type T (or class C) (either named or unnamed). The object can optionally be given a value of the correct type T (or TypeOf(C)).

```
<object instantiation> ::= OBJ <object name> : <type> := <value>
```

---

type signature

Syntax:

```
<type signature> ::=

  empty      // only allowed for builtin atomic types
| <typename> // For type aliasing
             // Must be declared in scope
             // Builtin (both atomic and composite)
             // types are declared in the outermost scope
| <enumerated type> // user constructed atomic type
| <product type>
| <record type>
| <vector type>
| <function type>
| <powerset type>
| <union type>
```

Semantics:

`empty` $= \varepsilon$. Internally defined semantics for builtin types.

---

enumerated type

Syntax:

```
<enumerated type> ::= enum { <id_1>, <id_2>, ... , <id_n>}
```

Semantics:

$$\textbf{enum}\ \{ID_1, ID_2, \ldots, ID_n\} \in S\ \textbf{with}\ S = \{ID_1, ID_2, \ldots, ID_n\}$$

S is a set of unique identifiers (e.g., Boolean = {True, False}). The uniqueness of the names used in an enumerated type must be checked. Furthermore, identifier names may not overlap with names used in an enumerated type. It must be possible (within a certain scope) to unambiguously distinguish between keywords, variable names, and enumerated names.

Object:

Objects of enumerated type take as value an identifier from the enumeration set.

```
OBJ o: TYPE {signature := enum {<id_1>, <id_2>, ... , <id_n>};} := <id_i>;
```

MSL Builtin Atomic Types

During bootstrapping, the builtin type names are loaded into the outermost type namespace.

Generic type

Syntax:

```
TYPE Generic "builtin: type variable";
```

Semantics:

Type variable, will unify with any other type; any type is a sub-type of Generic which implies any object may be assigned to a variable of type Generic.

Object:

```
OBJ o: Generic := <any object value>;
```

Integer type

Syntax:

```
TYPE Integer "builtin: positive and negative Natural Numbers";
```

Semantics:

$\in \mathbb{Z}$

Object:

```
OBJ o: Integer := <integer value>;
```

Real type

Syntax:

```
TYPE Real "builtin: Real numbers";
```

Semantics:

$\in \mathbb{R}$

Object:

```
OBJ o: Real := <real value>;
```

Character type

Syntax:

```
TYPE Char "builtin: ASCII character";
```

Semantics:

$\in$ *charSet* (currently ASCII, later Unicode)

Object:

```
OBJ o: Char := 'c';
```

String type

Syntax:

```
TYPE String "builtin: Char* (implemented as atomic type
                      for efficiency reasons)";
```

Semantics:

$\in charSet*$

Object:

```
OBJ o: String := "string";
```

The semantic checker accepts the empty signature only for the above builtin atomic types.

Bottom type

Syntax:

```
TYPE Bottom "builtin: bottom type" = enum {null};
}
```

Semantics:

The type which is a sub-type of any other type. By virtue of this, only `null`, the only object of type `Bottom`, can be used to denote an unassigned value for objects of *any* type.

Object:

```
OBJ o: Bottom := null;
```

Boolean type

Syntax:

```
TYPE Boolean
{
 description "builtin: Logic type" = enum {True, False};
}
```

Semantics:

Predicate logic Boolean

MSL Composite types

As a type can be interpreted as the set of possible values a variable of that type can take, composite types are in essence compositions of sets (see reftext on types this is one possible interpretation of types).

In the following,

```
<type_i> ::= <type_name_i> | <unnamed type declaration i>
```

where `<type_name_i>` refers to a type signature by name and `<unnamed type declaration i>` gives a type declaration in place.

product type

Syntax:

```
<product type>    ::= <type_1> x <type_2> x ... x <type_n>
```

Semantics:

$$\mathbb{N} \times (T_1 \times T_2 \times \ldots T_n)$$

The projection operator "*proj*" allows access to the i-th element of a product type variable:

$$\begin{aligned} proj: \quad & \mathbb{N} \times (\mathbb{N} \times (T_1 \times T_2 \times \ldots \times T_n)) \rightarrow \cup_i T_i \cup \{\textbf{fail}\} \\ & (i,(n,(v_1,v_2,\ldots,v_n))) \rightarrow \textbf{ if } (1 \leq i \leq n)\ v_i \textbf{ else fail} \end{aligned}$$

Note: the relation type R between T_1 and T_2 is a subtype of product type $T_1 \times T_2$

Object:

```
OBJ o: TYPE {signature := <type_1> x <type_2> x ... x <type_n>} :=
 (<obj of type_1>, <obj of type_2>, ..., <obj of type_n>);

to only assign the k-th field:

 o := (,, ..., <obj of type_k>, ...,,);

or

 proj(o,k) := <obj of type_k>;
```

record type

Syntax:

```
<record type>     ::= record
                      {
                       <id_1> : <T_1>;
                       <id_2> : <T_2>;
                       ...
                       <id_n> : <T_n>;
                      }
```

Semantics:

Mapping from distinct identifier strings (labels) to items of certain types.

$$\begin{aligned} & \{str_i \rightarrow T_i\},\ IndexSet = 1,\ldots,n \\ & \forall i \in IndexSet : str_i \in String \\ & \forall i,j \in IndexSet : i \neq j \Rightarrow str_i \neq str_j \end{aligned}$$

The record element selection operator "." provides access to record elements based on labels:

$$\begin{aligned} . : \quad & String \times \{str_i \rightarrow T_i\} \rightarrow \cup_i T_i \\ & str \rightarrow T_i \textbf{ if } \exists str_i \textbf{ for which } str = str_i \\ & str \rightarrow fail \textbf{ if } !(\exists str_i \textbf{ for which } str = str_i) \end{aligned}$$

Object:

```
OBJ o: TYPE {signature := record {<id_1>:<T_1>; <id_2>:<T_2>; ... <id_n>:<T_n>;}
 := {<id_1>:=<obj of T_1>; <id_2>:=<obj of T_2>; ... <id_n>:=<obj of T_n>;};

Some fields may be left unassigned.

To only assign specific record field <id_k>:

 o := {<id_k>:=<obj of T_k>};

or

 o.<id_k> := <obj of T_k>;
```

---

vector type

Syntax:

```
<vector type>        ::= <type> [ <dimension> <vector orientation> ]
<vector orientation> ::= ';' | ','
```

Semantics:

$$IndexSet \rightarrow T,\ IndexSet = 1,\ldots,n,\ n \in \mathcal{N}$$

We actually distinguish between row vector and column vector. `T[n;]` means column vector; `T[n,]` means row vector. Their semantics is defined by means of the transpose operator $.^T$ with the following properties:

$$.^T : \quad rowVector \rightarrow columnVector$$
$$.^T \circ .^T \equiv I$$

$$transpose \circ transpose \equiv I$$

The [] operator gives access to elements of a vector:

$$[] : \quad IndexSet \times (IndexSet \rightarrow T) \rightarrow T$$
$$(i, vec) \rightarrow [](i, vec) = vec[i]$$

Object:

```
OBJ o: TYPE {signature := <T>[n;]} // column vector
 := [<obj of T 1>; <obj of T 2>; ... <obj of T n>];

OBJ o: TYPE {signature := <T>[n,]} // row vector
 := [<obj of T 1>, <obj of T 2>, ... <obj of T n>];

To assign k-th fields of a record object:

 o := [,, ..., <obj of T k>, ...,,];
```

```
or

 o[k] := <obj of T k>
```

---

function type

Syntax:

```
<function type>   ::= <domain type> -> <image type>
```

Semantics:

$$Domain \rightarrow Image$$

Note: the variables of domain type `T1` and image type `T2` may have attributes. In case of "traditional" functions (as in C), one of these attributes may specify "by value", "by reference", or "by name" argument passing and result returning).

The above funtion is a pure *mathematical* function without side-effects.

The `eval` operator makes the evaluation of functions explicit:

$$\begin{aligned} eval: \quad & (T_{dom} \times (T_{dom} \rightarrow T_{im})) \rightarrow T_{im} \\ & (arguments, f()) \rightarrow f(arguments) \end{aligned}$$

Object:

```
OBJ o : TYPE {signature := <domain type> -> <image type>}
 := (<object of domain type>) -> <object of image type>
    { set of implementation statements };

Example:

 OBJ o : TYPE {signature := Integer x Real -> Integer x Integer}
  := (i,r) -> (r_sum, r_product)
     { r_sum := i+r; r_product := i*r};
```

---

powerset type

Syntax:

```
<powerset type>   ::= set_of <type T>
```

Semantics:

The set of all subsets of type T

$$2^T \text{ or } \mathcal{P}ow(T)$$

---

union type

Syntax:

```
<union type>      ::= union { <type T_1>, <type T_2>, ... , <type T_n> }
```

Semantics:

$$T_1 \cup T_2 \cup \cdots \cup T_n,\ IndexSet = \{1, \ldots, n\}$$
$$\forall i, j \in IndexSet : i \neq j \Rightarrow T_i \cap T_j = \emptyset$$

The disjointness property will allow the automatic determination of type of objects in a union type.

A union type enables the implementation of *polymorphic* operators. If an operator is applicable to a union type (e.g., $T_1 \cup T_2$), it will take arguments of both types. Some mechanism must be provided to automatically select the appropriate operator definition depending on the actual type of the argument.

reference type

Syntax:

```
<reference type>  ::= reference <type>
```

Semantics:

A generic way of describing references to objects. The implementation may use pointers, or reference by name.

$$\begin{aligned} ref &: T \rightarrow reference(T) \\ deref &: reference(T) \rightarrow T \end{aligned}$$

Note: currently not implemented.

bracketed type

Syntax:

```
<bracketed type>  ::= ( <type signature> )
```

Semantics:

A means of overriding default type composition precedence.

In the absence of (), the following precedence relations hold (high to low):

$$record > set > vector > product > function$$

MSL subtyping (subsumption)

A *sub-type* is obtained by restricting the set of possible values an object of that type can take.

```
<type_decl_subtyping> ::= SUBSUMES <type_name> <type_decl_signature>
```

Subsumption (sub-typing) is implicitly used to describe the semantics of a series of object declarations in case there are multiple objects with *identical names* (illegal in programming languages). This semantics is independent of whether the duplicate declarations occur within a set $\{\dots\}$ or sequence

$$\dots$$

context. The semantics is *latest overrides*: a later declaration leads to an error if the declaration types are incompatible. If however, the later type is a sub-type of the former, the object takes on the sub-type.

MSL type extension

A type extension is obtained by extending the signature of the type. The extended signature must be an acceptable extension (supertype) of the original signature. Currently, the only used extension is the extension of the RECORD type with extra fields. Extension can be done recursively (*e.g.*, RECORD inside RECORD, *etc.*). Optionally, an extension can be empty (nothing gets added).

```
<type_decl_extension> ::= EXTENDS <type_name> <with_signature>
<with_signature>      ::=
                        | 'WITH' <signature>
```

Extension commonly leads to multiple object declarations with identical names. The previously described "lastest overrides" subsumption semantics is then used. Intuitively, this means sub-classes (discussed below) may only specialize (refine) super-classes.

MSL classes

A class is in essence a type with default values. The class signature is its type. An MSL class can be constructed in different ways:

- by assigning an OBJect value to a type
- through specialisation of an existing class. Specialisation may be empty (see <obj_value_stuff>). As a specialisation of a class is a subtype of the original class, empty specialisation can be used to provide subtyping information to the type checker.
- through extension of an existing class (or type). As with types, extension may be empty.

```
<class_decl_stmt> ::= CLASS <class name> <description> <class_decl>
<class_decl>      ::= <class_decl_regular>
                    | <class_decl_specialises_class>
                    | <class_decl_extends_class>
<class_decl_regular>            ::= '=' <signature> <obj_value_stuff>
<class_decl_specialises_class>  ::= SPECIALISES <class_name> <obj_value_stuff>
<class_decl_extends_class>      ::= EXTENDS <type_or_class_name> <extend_value_stuff>
<extend_value_stuff> ::=
                       | WITH <object>
<obj_value_stuff>    ::=
                       | ':=' <object>
```

In Appendix B, the above definitions are used to construct model libraries for

- generic models (generic.base.msl);
- Forrester System Dynamics models (sd.msl);
- waste water treatment models (wwtp.base.msl).

3.2.4 Lexical scoping

As both causal and non-causal formalisms must be represented in MSL-USER, it is meaningful to give syntactic support for this.

The following demonstrates the use of *lexical scoping* with both *set* and *sequential* semantics. In Appendix A, a small compiler implements the desired semantics. The compiler is built using the compiler compiler Gentle (`www.first.gmd.de/gentle/`).

With lexical scoping, an *object declaration* has a certain lexical scope *extent*. Within that extent, any *identifier application* with the same name as the object will be *resolved/bound to* that declared object.

Two types of scope extent are of practical use:

1. *Set semantics:* the order of declaration and application do not matter. The extent of a declaration is the whole enclosing set.

   Below, the extent of the declaration of object x on line (3) is the whole enclosing set reaching lines (1)–(6). Hence, *all* applications of x are bound to the declaration on line (3).

   ```
   {                              (1)
     x                            (2)
     OBJ x: 20                    (3)
     x                            (4)
     y                            (5)
   }                              (6)
   ```

   Note how in particular, the application on line (2) is bound to a declaration which comes *after* it. This is consistent with set semantics where order does not matter. In some sequential languages such as Pascal and C, a *forward* declaration is needed to let the compiler know that a declaration (or more often a definition) will follow. This extra burden to the user is tolerated as it avoids one extra compiler pass. One particular use for forward references in sequential languages is to allow cyclic references: one object declaration refers to an other and vice versa, hence no sorting of declarations will ever resolve the problem. Using set semantics alleviates this problem as shown in the MSL-USER example below:

   ```
   {
     OBJ a: Ta := {ref(b), ref(a)},
     OBJ b: Tb := {ref(a), ref(b)}
   }
   ```

2. *Sequential semantics:* the order of declaration and application *does* matter. The extent of a declaration ranges from the point of declaration to the end of the enclosing sequence. Below, the extent of the declaration of object `x` on line (3) ranges over lines (3)–(6). Hence, though application of `x` on line (4) is bound to the declaration on line (3), the application on line (2) is not within the scope of the declaration and an error will be reported (unless `x` is declared in an outer scope).

   ```
   [                              (1)
     x                            (2)
     OBJ x: 20                    (3)
   ```

```
    x                    (4)
    y                    (5)
  ]                      (6)
```

In both set and sequential semantics, it may happen that an object with the *same name* (*i.e.*, using the same identifier) is declared more than once within the same lexical scope. In the case of sequential semantics below,

```
[                        (1)
  OBJ x: 20              (2)
  x                      (3)
  OBJ x: 30              (4)
]                        (5)
```

the declaration on line (4) is within the scope of the declaration on line (2). In the case of set semantics below,

```
{                        (1)
  OBJ x: 20              (2)
  x                      (3)
  OBJ x: 30              (4)
}                        (5)
```

both declarations on lines (2) and (4) are in each other's scope. In a *denotational* terminology, we say overlapping declarations occur if the intersection of the respective scopes is non-empty. Such a situation is ambiguous and possibly erroneous and the *semantics* we use in this case is to only *retain the first* (as appearing in the source text) declaration and to ignore subsequent declarations. Alternately, the last declaration could be retained. Also, a *warning* message will be output.

In both set and sequential semantics, *nested lexical scoping* is supported. Here, if an object application cannot be found in the current scope, the enclosing scope is searched, and so on recursively. The *percieved* total enclosing scope, collecting all enclosing scopes is called the *environment.* Inner declarations can *hide* outer declarations as shown in the example below.

```
{                        (1)
  {                      (2)
    x                    (3)
    y                    (4)
    OBJ x: 30            (5)
  }                      (6)
  x                      (7)
  OBJ x: 20              (8)
  OBJ y: 40              (9)
  y                     (10)
}                       (11)
```

The application of `y` on line (4) is bound to the declaration on line (9) in the enclosing scope as it is not found in the local scope. The application of `x` on line (3) however is bound to the declaration on line (5) in the local scope, which *hides* the declaration of `x` in the enclosing scope on line (8). The application of `x` on line (7) however is bound to the declaration in *its* local scope, that on line (8).

If an application is not found in the local scope, nor in the environment (*i.e.*, recursively searching all enclosing scopes, all the way to the outermost level), the application cannot be bound to a declaration and an *error* results.

The following demonstrates the *combination* of nested sets and sequences. The values of the different applications are given on the right hand side with some comments.

```
{
  x                          6 : set, x is declared below
  OBJ x: 6
  y                        100 : set, y is declared below
  [
    x                        6 : seq, declaration from surrounding scope
    OBJ x: 20
    x                       20 : seq, declaration from this scope
    {
      x                     30 : set, x is declared below
      OBJ x: 30
      y                    100 : declaration from top level scope
                                 the surrounding level has seq
                                 semantics, thus y:200 is not declared yet
    }
    OBJ y: 200
    y                      200 : seq, declaration from the line above
  ]
  OBJ y: 100
  {
    x                       20 : set, x is declared below
    OBJ x: 20
    y                      100 : y from the surrounding scope
  }
  OBJ x: 999
  x                          6 : declared in this scope
}
```

The question *why* both sequential and set semantics (in combination with lexical nested scoping) are supported deserves some attention.

1. Sequential semantics: To express a *sequence* of operations, actions, . . . a matching semantics must be available. This choice was made in most procedural programming languages, as in those languages one wants to express exactly such a sequence. Supporting sequential semantics in MSL-USER allows us to transpose a piece of procedural code with the *same* semantics. This transposition is a valuable alternative to calling *external* sequential code as, such external code cannot be type-checked nor can global optimizations be performed. Above all, being able to represent sequential semantics allows us to explicitly represent *sorted* algebraic and differential equations in MSL-USER. In most modelling systems supporting sets of equations, the sorted equations can only be seen in the generated simulator code (MSL-EXEC in our case). One of the design rules for the MSL-USER compiler is that it must be possible to write out, in valid MSL-USER (*i.e.*, can be read in again), *every* intermediate step in the compilation process.

2. Set semantics: Many formalisms are based on set-theory. Often, a core component of the formalism is a *set* of states, a *set* of transitions, a *set* of equations in a DAE, *etc.* To meaningfully describe such formalisms in MSL-USER, exactly the same set semantics must be supported. Note how this set semantics is also useful in a modelling language where the class extension mechanism may lead to a concatenation of pieces of classes where declaration and use are out of order. As an example (further elaborated in the next chapter), the terminals of a biological model are vectors containing MassFlux elements. The size of the vectors is determined by the biochemical components the user wants to take into account. In our generic (mass balance) models, we want to take into account any number of components, which will however only be given by the user at the moment of

model instantiation. This is done through the declaration of an enumerated type `TYPE Components = ENUM H2O, CO2, S_S, ... END_ENUM`. In the generic model, we declare an interface to be of type `MassFlux[NrOfComponents]` with `NrOfComponents = Cardinality(Components)`. The reference to Components is a forward reference which is perfectly handled within the set semantics.

In the Gentle compiler in Appendix A, two passes are used:

1. The first pass is used to:
   - add a unique entry for each declared object to a global object symbol table named `Declarations`.
   - build a list of declared objects in each scope. This list contains references to the declarations (in the `Declarations` table) within that scope. In case of multiply declared objects, they are all entered in the list. During application-to-declaration binding, this will be checked for. Note how, in case of a multiply declared object (e.g., `OBJ a: T_a, OBJ a: T_b`), all unique entries will continue to exist (we cannot delete entries from a table in Gentle). This is despite the fact that we may choose to only retain the first declaration when resolving object "applications". When implementing this in another language, we would delete the additional declarations. Note how, when writing out the Abstract Syntax Tree (AST) and Symbol Table (ST), and reading in again, the erroneous extra declarations will have disappeared. The list of objects declared in a block is attached to the AST node for that scope. The list allows us to traverse all declarations. In the AST, every declaration node `declaration(IDENT, OBJECT)` is replaced by a node `decl_ref(Declarations)` referencing the object symbol table.
2. The second pass is used to resolve applications to declarations. In case of a set scope, the list of declarations in that scope is used to `Define()` these declarations in a hash table. In case of a sequence scope, the actual `decl_ref()` nodes are used to `Define()` the declarations as they are encountered. See the `RESOLVE/BIND NESTED IDENTIFIER APPLICATION TO DECLARATIONS` comments for a description of the use of `{Def|Has|Undef}Meaning` to keep track of the valid identifiers and their appropriate binding at each nesting level. During this pass, we may wish to remove the now redundant declaration nodes.

3.2.5 Generic model transformations

As opposed to formalism-specific transformations discussed in the previous chapter, some model transformations are formalism-independent.

Variable probes

Due to the encapsulation of model knowledge, only interface and parameters are externally visible. If one wants to inspect internal variables, these will have to be brought to the surface. This can be automated in the following way (for a single variable):

- atomic model: add a terminal to the model interface. Produce an equation linking the internal variable to the interface terminal. Assign OUT causality to that terminal.
- coupled model: the above procedure must be performed recursively from the atomic model where a variable needs to be inspected to the topmost coupled model level.

Manipulating parameters

For purposes of control, one wants to manipulate parameters of a model. From a formal standpoint, manipulating parameters is impossible (parameters are by definition constant during a simulation run). Thus, a new model has to be constructed whereby parameters are migrated to the interface:

- atomic model: add a terminal to the model interface for each "manipulated" parameter. Assign OUT causality to these terminals. Remove the manipulated parameters from the parameter section. The dynamic equations now implicitly refer to interface terminals rather than to parameters.

new CM = < {Model 1, Model 2, Transducer},
{Model 1.x -> Transducer.in, Transducer.out -> Model 2.y}>

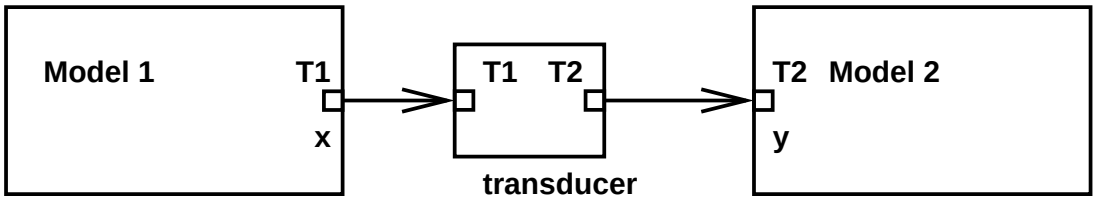

Figure 3.4: Inserting Transducers

- coupled model: the above procedure must be performed recursively from the atomic model whose parameters need to migrate to the interface, upwards. Therefore, parameter coupling equations (linking parameters of a coupled model and its sub-models) need to be updated.

This transformation is implemented in the MSL-USER compiler.

Transducers

For a coupled model to be meaningful, the TYPE of connected sub-model terminals must match. One might for example connect a discrete output to a continuous input or a distance output in meters to a distance input in inches. In both cases, the introduction of a "transducer model" (Figure 3.4) is meaningful. In the above examples, an interpolator transducer and a unit converting transducer can be used. Note the actual *change* in the coupled model CM: an extra transducer *model* has been added, resulting in a *new* coupled model.
A common use of transducers in WWTP models is in coupling between models which use equivalent (but different) sets of interface variables (*e.g.*, mass and volume *vs.* concentration).

Model invariants

When expressing a model in terms of algebraic and differential equation relations between state variables, one implicitly assumes the model is independent of *all* possible physical variables not included in the state variable list. Other models coupled to this model may be dependent on such variables (say, variable T). Obviously, coupling of two such models is not possible. The solution is to augment the interface of the model independent of T with a $T_{interface}$ terminal. Internally, a variable T and the trivial pass-through equations $T = T_{interfaceIn}$ and $T_{interfaceOut} = T$ are added. This explicitly represents the model's invariance of T. This approach is preferred over the use of "global" variables.

In WWTPs, the most common pass-through variable is temperature.

Flattening coupled models

Through model flattening, a concrete hierarchical model is transformed into an atomic model. This monolithic atomic model is fit to be executed using a simulation kernel appropriate for that atomic model's abstraction. Model flattening involves the recursive replacement of coupled models by a flattened version of their underlying sub-model configurations. This flattening is required as the semantics of a coupled model is merely related to sub-model identity and coupling and not to dynamic behaviour (which is only present in atomic models). Thus, flattening of a coupled model consisting of only atomic models will result in one atomic model of the same type as all the sub-models. This atomic model (a large set of algebraic and ordinary differential equations in our case) will allow for numerical simulation of the dynamic behaviour. The mechanism of closure in case of DAE models consists of:

- name unification: making local sub-model names globally unique (*e.g.*, by prefixing the unique name of the sub-model).

- concatenation of sub-model equations. This simple approach is possible thanks to the closure property of the DAE formalism (concatenation of DAE models is again a DAE model).

3.3 Meta modelling

In the following, we describe the concept of *meta modelling*: modelling at a meta level the various concepts used in different formalisms. It is believed meta modelling will allow for the automated construction of highly specialized modelling and simulation tools. The genericity of MSL-USER was a first step in this direction. In what follows, the concept of *multi-paradigm* modelling as a generalisation of multi-formalism modelling is introduced. Focus is on control engineering applications, a domain in which the need for dedicated modelling and simulation tools is high.

The analysis and design of engineered systems involves expertise from many disciplines and entails a variety of implementation technologies (*e.g.,* embedded software, microelectromechanical systems, analog circuits, and digital circuits). The *heterogeneous* nature of these systems invariably combines with an architecture of different concurrent components that interact through continuous signals or discrete message passing. The corresponding complexity has led to the use of more formal approaches to system design. Dedicated modelling formalisms are applied to different aspects and/or components of the system. Consequently, the complete system specification process combines several modelling, design, implementation, and realization formalisms such as differential equation modelling, continuous time signal processing, and discrete event controllers. Decomposition of the entire specification task allows teams of experts to concurrently work on their domain of expertise, *e.g.,* control law design, simulation, optimization, modelling, and verification.

To comprehensively handle system design in such a heterogeneous environment, multiple approaches based on different paradigms have to be combined. In the following, the definition by Nordstrom is used [Nor99] "A *modelling paradigm* is a set of requirements that governs how **any** system in a particular domain is to be modelled. These modelling requirements specify the types of entities and relationships that can be modelled; how best to model them; entity and/or relationship attributes; the number and types of views or aspects necessary to logically and efficiently partition the design space; how semantic information is to be captured by, and later extracted from, the models; any analysis requirements; and, in case of executable models, run-time requirements."

A tool that "understands" each of the corresponding formalisms (*i.e.,* has a model of them) can be used to ensure consistency between different formalisms, allow for quick adaptation to changing needs, exchange information, and efficiently provide tailored modelling environments that are maximally constrained with respect to the domain of operation. For example, a designed control law that is automatically translated into its implementation, *i.e.,* the hardware binding. Here the control language focuses on stability and other control characteristics, whereas the implementation has to deal with issues such as schedulability, reliability, and security, which requires different analysis formalisms. If consistency and cross coupling across these languages is ensured, implementation choices (*e.g.,* the "time for space" trade-off) can be conveniently conveyed back to the control design engineer.

Multi-paradigm modelling is also critical for reconfigurable systems as the supervising mechanisms that combine with a flexible control architecture are based on different modelling formalisms (even different plant models), and need to integrate with the control architecture. One solution is model integrated computing [SKB+95], which allows changes in the system model/specification and translates these automatically into software (or even reconfigures hardware).

Control system design is achieved by using many software tools, sophisticated development techniques and methodologies relying on library components and automatic (code) generation approaches. Specialized computer automated tools for each of these domains are very helpful or even indispensable to carry out the related tasks, as the process of control design requires the integration of, *e.g.,* modelling, simulation, control law design, dynamic control law integration with safety and redundancy management control logic (*e.g.,* surveillance functionality), and control robustness assessment. Typically, there is no single tool that addresses all these issues, and, therefore, a suite of tools is used throughout the design process. Because

these tools hardly ever are compatible, the sharing and coordinating of information flow between project teams inevitably leads to a lot of overhead in terms of collaboration, and is very error prone, inefficient, and expensive. Moreover, similar tasks may be carried out multiple times and even simultaneously.

All these issues are addressed by adopting a meta modelling approach to data exchange [Com94, Ern96, EW00, Fis99] and modelling paradigm and environment specification [EK00, KNLS00]. Here, an overview is provided of the use of meta modelling concepts as used in control system design. Section 3.3.1 reviews the specification requirements for multi-paradigm modelling environments. Section 3.3.2 discusses the use of a generic unifying language. Section 3.3.3 gives an overview of an alternative approach, the use of meta modelling, and shows how this supports the required flexibility and specificity needed for multi-paradigm modelling.

3.3.1 Modelling environment requirements

To facilitate computer automated control system design, modelling, and analysis, computer based environments need to be available that are tailored to the particular task at hand. The most efficient and flexible approach is to model the modelling environment and automatically generate a complete specification that can be directly compiled into an integrated development environment. This requires the specifications to be well structured, to define the precise syntax and semantics of a language, and to not be mixed with language implementation details.

A clean separation in concepts leads to [Gro99b, Nor99]:

1. *Syntactic specifications* that can be divided into (i) the concrete syntax, which captures the actual representation, *e.g.,* a textual language specified by Backus-Naur Form (BNF) constructs, and (ii) the abstract syntax, the language syntax devoid of implementation details, which allows for the representation of the essential constituents of a formalism.

2. *Semantic specifications* that may include model composition constraints to capture domain specific concepts and constraints. These can be classified as (i) static semantics that can be checked during model composition (*e.g.,* in logic circuits the number of loading components allowed to connect to one output), and (ii) dynamic semantics that can only be checked during execution (*e.g.,* whether a certain state is reached). More generally, dynamic semantics encompasses the semantics of model execution. The distinction between static and dynamic semantics is not related to representation, but rather to the availability of sufficient information to assert the validity of certain constraints before model execution. If this is not the case, the constraints need to be passed on to a model execution environment. Such constraints can be represented in the meta model structure or by a constraint language (*e.g.,* first order predicate logic).

3. *Presentation specifications* that are critical for specification of the complete modelling environment and that specify the appearance of entities, relationships, and attributes.

4. *Interpreter specifications* which are necessary to extract information from each of the models to allow, *e.g.,* documentation and execution. As such, it is a concrete realization of dynamic semantics.

The first two specify the modelling language and the latter two complete the specification of a modelling environment. In a graphical language, the syntax is a collection of modelling object types, possible relations between them, and their allowed attributes. Static semantics pertain to the well-formedness of language constructs, and they represent an invariant that must hold across the family of models that can be designed using the modelling language. Dynamic semantics relate to the interpretation of the model constructs and cannot be specified by language constructs.

3.3.2 Multi-paradigm modelling with a generic standard

One approach to deal with the issues of tool interoperability and multi-formalism approaches is to develop a unifying generic standard, *e.g.,* Modelica [EBB+99] and VHDL-AMS [Gro99a]. If such a standard allows

for the use of multiple formalisms in one environment it corresponds to a unifying super-formalism that can be used for model exchange and can mitigate the interoperability problems.

Because of the variety of formalisms that address different aspects and types of specification that are used throughout a system design lifecycle, if at all possible, it is difficult at best to establish one such generic formalism. For example, it would have to include the rather different syntax, semantics, representation and interpretation specifications of data flow diagrams, control flow diagrams, formalisms such as statecharts, Grafcet, and Petri Nets, physical modelling formalisms such as bond graphs and object diagrams, block diagrams, and process diagrams [DA92, Har87, HP88, KMR90, Mur89]. For example, in terms of their interpretation, computational models such as differential equations, state/event, discrete event, synchronous/reactive, and (a)synchronous message passing [DGG+99] are used. To capture all of these would require one underlying computational model that subsumes all others. However, modelling is not just a question of whether it is possible to represent system knowledge with a certain formalism, it critically depends on whether it can be done elegantly and intuitively [Lee99, LL00].

A standard unifying formalism works well if the area of application is sufficiently restricted [MML99, MOE98]. For example, Modelica concentrates on physical system modelling and builds on the combined differential equation and state/event computational models. This allows for a comprehensive language well suited to its purpose. However, because it does not separate abstract from concrete syntax, it may result in an overly rigid formulation. For example, the abstract syntax of an iterator construct contains an initial value, final value, and step size. A design decision is required whether to use the concrete syntax that corresponds to `i=0: 10: 2` or to use `i=0: 2: 10`. This choice will be incompatible with particular domains and cause an increased threshold to acceptance of the standard. Note that it is impossible to allow both variants, a common solution in case of such design decisions, which leads to a bloated language specification.

The use of standards relies heavily on the concept of libraries, *i.e.*, sets of predefined components typically related to a specific domain. Each library embodies a particular modelling paradigm. This implies that any one particular design tool is required to contain a compiler for each of the included formalisms, even those not applicable to the particular task at hand. Also, in Modelica, the presentation semantics are dissociated from the language syntax and semantics. Therefore, the choice between different visualisations of, for example an electrical resistor, is possible by constructing two separate libraries. Because of the inheritance construct, each of these components can inherit the same functionality, only specializing the graphical appearance. However, if one imports an electrical circuit designed with a particular library, the graphical presentation is fixed, *i.e.*, no automatic transformation to the desired presentation occurs because no knowledge is available of what a resistor is.

The use of a standard works well for modelling affinitive domains. In case of Modelica, this is the structure of a physical system. Behavioural models, such as block diagrams and statecharts, require a graphical notation and semantics that may differ significantly, and, therefore, may be hard to capture in the standard. For example, in physical systems, models based on energy flow have no computational causality, and, therefore, the representation does not concern direction of connections. In block diagrams, on the other hand, causality of input and output signals is inherent. Typically, this is indicated by adorning the relation with an arrowhead. The semantics of this cannot be easily added to a non causal relation. For example, in the Modelica block diagram library, the relations are still non causal, and the input-output behaviour is specified by the connected objects. This specification is not related to the graphical representation, though. So, for each port instance it is specified separately, whether it operates on input, output, or both. The graphical representation then is drawn as an arrowhead without this having a direct implication on the semantics.

In conclusion, the flexibility required for a standard calls for increasingly generic constructs such as undirected relations. Furthermore, given that the language needs to be sufficiently powerful to specify a multitude of formalisms, its genericity makes it hard to use for specific analyses, and it becomes hard to prove certain characteristics of a model. Also, such languages typically lack a *constraint language* to limit the family of models one has to deal with. Rather, constraints are hard coded into the model compiler. For example, the requirement that an electrical circuit includes at least one ground node cannot be specified. This allows for an entire class of electrical circuits (infinitely many) that can be modelled but cannot be executed. Finally,

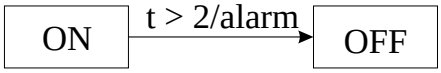

Figure 3.5: A state transition diagram model.

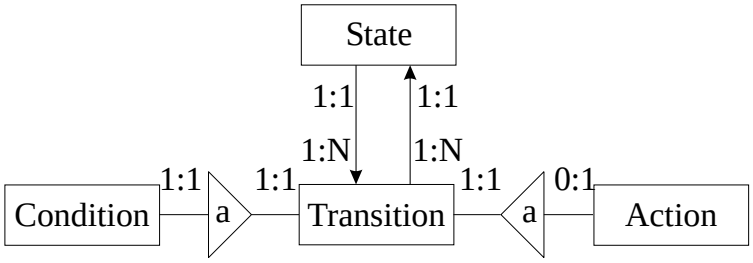

Figure 3.6: A model of state transition diagram models.

the rigidity of such a generic standard makes it hard to keep up with the state of the art and disallows users to define additional model specifications that they need in order to solve their specific problems.

3.3.3 Meta modelling

A proven method to achieve the required flexibility for a modelling language that supports many formalisms and modelling paradigms is to model the language itself. This is exemplified by, among others, the *domain modelling environment* (DOME) [Hon99, EK00] and the *multigraph architecture* (MGA) [SKB+95]. To illustrate this notion, consider the state transition diagram in Figure 3.5. When in the ON *state*, a *transition* to OFF occurs when the *condition* $t > 2$ is true and this generates an alarm *action.* The state, transition, condition, and action elements are part of any state transition diagram and their dependencies can be modelled as shown in Figure 3.6. This model specifies a family of state transition diagrams where each instantiation has states that are connected by transitions. Each state can have any number of exit transitions, indicated by the $1:1$ and $1:N$ cardinality on the downward arrow in the figure, *i.e.*, each state can have between 1 and N transitions and each transition has to exit between 1 and 1 states (it has to be connected to one and only one state), where N represents infinitely many. Each transition can enter only one state and each state may have any number of entering transitions indicated by the cardinality on the upward arrow. The transitions between states have two attributes, one condition that allows the state transition to be taken and one optional (indicated by the $0:1$ cardinality) action.

The model in Figure 3.6 can be used to specify different state transition diagram formalisms, *e.g.*, the action attribute can be made mandatory for each transition by changing the cardinality from $0:1$ to $1:1$. Furthermore, actions can be associated with states as well and hierarchical state machines can be modelled by giving each state a state attribute (*i.e.*, a relation with itself).

Such a model of the modelling language is called a *meta* model. It prescribes the possible mathematical structures (formalisms) that can be expressed in the modelling language and can be tailored to specific needs of particular domains. From the meta model specification, the modelling language can then be instantiated automatically. This requires the meta model modelling formalism to be sufficiently rich and support the constructs needed to define a modelling language. To allow for easy extension, the meta model modelling formalism can be modelled by a *meta meta* model. This meta meta model specification captures the basic elements that can be used to design a meta model modelling formalism. In case new concepts and structures are required, these can be conveniently modelled at a meta meta level.

For example, the state transition diagram meta model in Figure 3.6 is limited to the family of state transition diagrams. This restriction can be removed by modelling the model of state transition diagrams in Figure 3.6 by the meta meta model in Figure 3.7. It contains an abstract representation of the mechanisms that are part of the state transition diagrams meta model, *i.e.*, entities (states, transitions), attributes (actions and conditions), and relations between them. This meta meta model groups entities and relations by an *object*

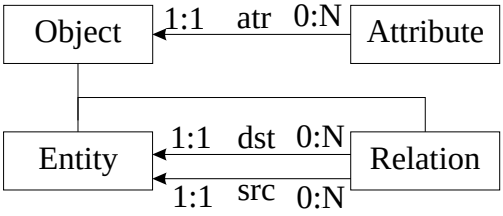

Figure 3.7: A state transition diagram meta meta model.

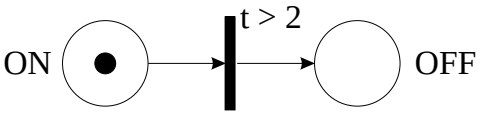

Figure 3.8: A Petri net model.

model component and each of them optionally has any number of attributes. It also shows that each relation connects to one entity as its source (marked src) and one entity as its destination (marked dst), and, therefore, directed links can be used. The meta meta model specification language has to consist of entities, attributes to specify the cardinality and relations to specify the three types, (i) source relations (src), (ii) destination relations (dst), and (iii) attribute relations (atr).

Given the meta meta model in Figure 3.7, a broader family of meta models can be described. For example, a Petri net as illustrated in Figure 3.8 consists of *places* (shown as transparent circles) and *transitions* (shown as solid rectangles). Each place has *connections* to transitions and each transition to places. A transition may have a condition and when this condition is true and all its input places, *i.e.*, places that are the source to the transition, contain a *token* (shown as a black dot inside a place), it may fire (the corresponding transition may be executed). The Petri nets meta model shown in Figure 3.9 specifies places and transitions that can connect to one another. Furthermore, the tokens are specified as optional attributes of a place and conditions as an optional attribute of a transition. The family of Petri nets that can be instantiated from this meta model allows places with multiple tokens. However, in some modelling paradigms, only one token per place is allowed, which can be conveniently changed in the meta model by specifying $0:1$ cardinality, and, consequently, constraining the family of Petri nets. Note that this represents both a static and dynamic constraint. When the Petri net is initialized, it can be ensured by the model editor that each place has been assigned at most one token. However, a dynamic check is still required whether this constraint is violated during execution.

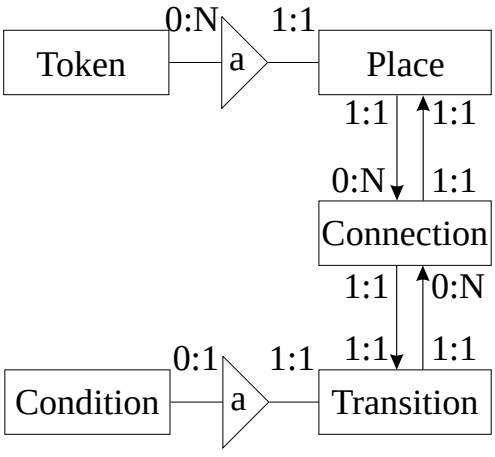

Figure 3.9: A model of Petri net models

Level	Description	Example
meta meta model	Modelling language for specifying meta models	`Relation hasDestination Entity`
meta model	Modelling language for specifying models, an instantiation of a meta meta model	`State connectsTo Transition`
model	The model of an object (which could be a model), an instance of a meta model	`when t > 2 transition from ON to OFF`
object data	An instance of a model	$0 < t \leq 2 \quad alarm = F$ $t > 2 \quad alarm = T$

Table 3.1: Four layer meta modelling structure.

In addition to the elements of the graphical language, often a *constraint language* is facilitated by a meta modelling language to specify domain specific constraints that are hard to incorporate otherwise in the meta model. This language can be used to rule out semantically incorrect models and greatly reduce the family of models that can be modelled [Nor99]. For example, a model of a family of Petri nets could include the constraint that there should never be more than ten tokens in the net (to model a resources limit) by an additional specification `Token.allInstances->size < 10`.

The outlined meta modelling approach leads to the four layer structure in Table 3.1 [Gro99b]. The object data row represents the data generated from a particular model, *e.g.*, the simulation results of a physical system model in the time domain. These data are one instance of the set of data that can be generated by the model. The model row represents the particular model such as the state transition diagram in Figure 3.5. At a meta level, the meta model row represents a class of models, *e.g.*, the model of the family of state transition diagrams in Figure 3.6. This meta model is described by a language that is specified by a meta meta model, the meta meta model row. This could be the model in Figure 3.7.

The apparent advantage of this approach is the tremendous flexibility that can be achieved. Consider the iterator construct discussed in section 3.3.2. At a meta level it can be specified to consist of an initial value, final value, and step size, but the concrete syntax can be instantiated as desired. This does not affect the abstract syntax nor the semantics, though, and exchange of models with this construct is inherently supported. Moreover, the concrete syntax of the iterator construct will be automatically adapted to the desired form when loaded by a different tool.

This flexibility is also manifested in the ease with which new formalisms can be designed. By adapting the model of a modelling formalism, and automatically generating a prototype of the modelling environment, design choices can be rapidly evaluated. Furthermore, if the same language for meta model specification is used, consistency between different formalisms can be achieved. For example, if a component in a block diagram has certain output signals, these values have to be computed internally. In case the particular component is modelled by a state transition diagram, the output of this model has to correspond with the block diagram output at a higher level.

To allow deeper specification of the semantics of a modelling formalism, in particular of the dynamic semantics, it is useful to express how a model structure (meta model) can be mapped onto other model structures. An invariant of this mapping must obviously be the modelled system's dynamic behaviour. Examples of mappings are the transformation between a Bond Graph and a corresponding system of differential and algebraic equations (DAE), or between a Statechart and an equivalent DEVS model. These mappings or model transformations are the basis for symbolic model analysis as well as for automatic (simulation)

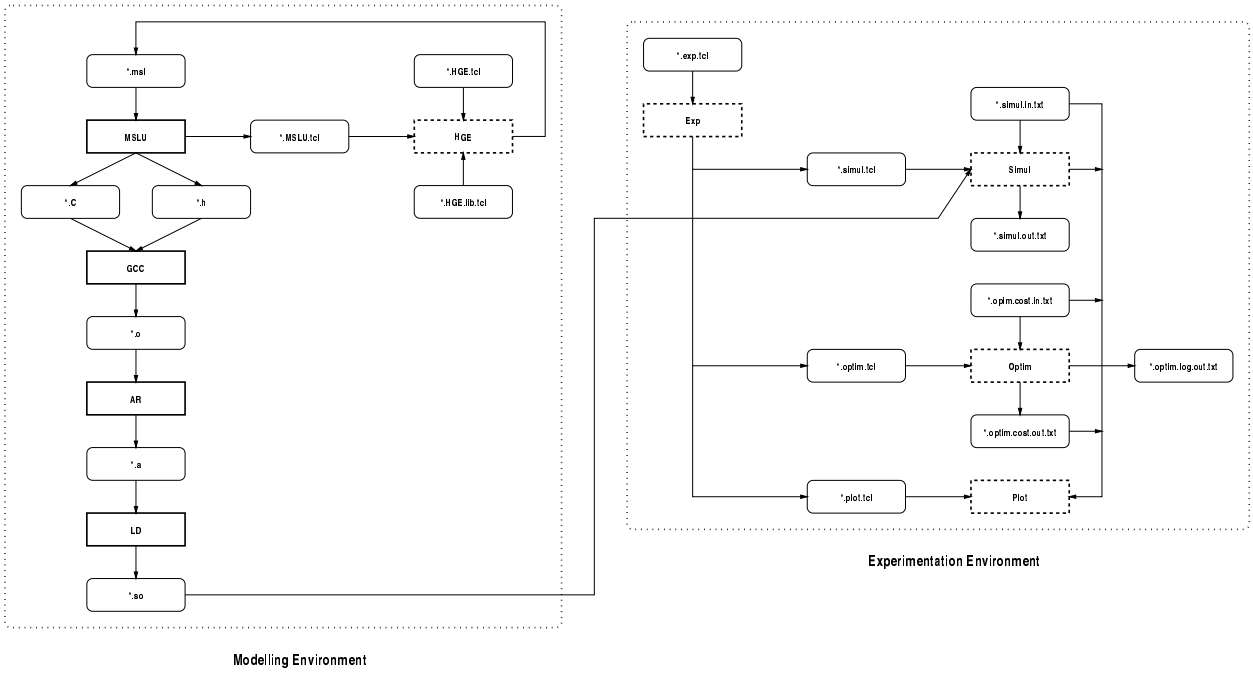

Figure 3.10: The WEST++ modelling and simulation process

code generation. Possible formalism transformations can be charted in a *Formalism Transformation Graph* (FTG) [Van00].

3.4 The WEST++ modelling and simulation environment

WEST++ [VCV98], originally an acronym for “Waste water treatment Environment for Simulation and Training”, is a distributed environment for modelling and simulation. Currently, it is mostly used for the study (analysis, optimal design, and control) of bio-activated sludge waste water treatment plants. The environment is generic however and supports multiple formalisms, most notably Forrester System Dynamics and DAEs. It is a limited implementation of the GMSA described in the first section of this chapter.

WEST++ makes a strict distinction between modelling and experimentation. The reason for this is not only modularity, but also to have the ability to substitute either the modelling or experimentation environment by different tools. Furthermore, a separate experimentation environment makes it possible to deploy a simulator without sharing model knowledge, which is often necessary in a commercial context.

The modelling and simulation process as supported by WEST++ is depicted in detail in Figure 3.10. It shows the flow of information through the system. It also shows how the Modelling Environment relates to the Experimentation Environment. In WEST++, the following information is used in the form of files:

- *.msl: MSL-USER model descriptions.
- *.C and *.h: MSL-EXEC model descriptions.
- *.o: Compiled MSL-EXEC models.
- *.a: Archive of compiled MSL-EXEC models.
- *.so: dynamic shared object version of compiled MSL-EXEC models archive.
- *.MSLU.tcl: Information extracted from a *.msl file for use with the Hierarchical Graph Editor.
- *.HGE.tcl: Hierarchical Graph Editor configuration file.
- *.HGE.lib.tcl: Hierarchical Graph Editor library file.
- *.exp.tcl: Experimenter configuration file.
- *.simul.tcl: Simulator configuration file.
- *.simul.in.txt: Simulation variable input file.
- *.simul.out.txt: Simulation variable output file.

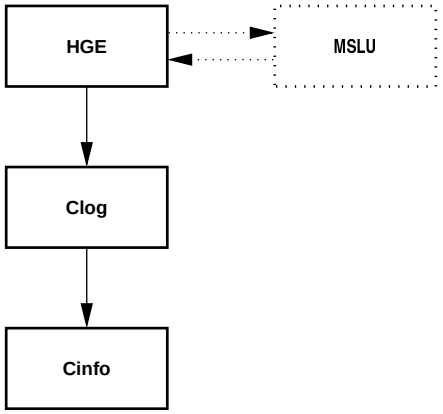

Figure 3.11: Modelling Environment dependencies

- *.optim.tcl: Optimiser configuration file.
- *.optim.cost.in.txt: Optimiser measurement data input file to be used during cost calculation.
- *.optim.cost.out.txt: Optimiser cost output file.
- *.optim.log.out.txt: Optimiser log output file.
- *.plot.tcl: Plot configuration file.

Apart from the above list of files containing information used by the system, each macro-module also has a resource file which should reside in the user's home directory:

- Crc.tcl: Used by the Info Server, the Log Server and all other macro-modules.
- HGErc.tcl: Used by the Hierarchical Graph Editor Server.
- plotrc.tcl: Used by the Plot Server.
- exprc.tcl: Used by the Experimentation Server.
- optimrc.tcl: Used by the Optimisation Control Server and the Optimisation Engine Server.
- simulrc.tcl: Used by the Simulation Control Server and the Simulation Engine Server.

3.4.1 Modelling Environment architecture

The WEST++ Modelling Environment currently allows for the translation of MSL-USER model descriptions into MSL-EXEC code and for the creation of coupled models in a graphical way. It operates in a distributed fashion and is made up of a number of so-called macro-modules. These macro-modules are self-contained and have a well-defined functionality. They communicate by sending messages over a TCP/IP network. All macro-modules are made up of a hyperwish and a set of Tcl sources [Ous94]. A hyperwish is a Tcl interpreter in which one or more extensions have been registered. These extensions have either been obtained via the public domain or were implemented specifically for WEST++. The implementation language for Tcl extensions is C/C++.

The translation of MSL-USER models into MSL-EXEC code is done through a stand-alone pre-compiler whereas the creation of coupled models uses the Hierarchical Graph Editor Server. The macro-module allows one to graphically construct a coupled model out of other models (atomic or coupled) and to generate MSL-USER code for these.

Figure 3.11 shows the dependencies between the macro-modules. Arrows point to those macro-modules one particular macro-module depends on. This means that the dependent macro-module cannot operate properly as long as the macro-modules it depends on are not fully available. Currently, the Modelling Environment is fairly simple compared to the Experimentation Environment. It only consists of an Info Server, a Log Server and the Hierarchical Graph Editor (HGE). The WEST++ Modelling Environment macro-modules are the following:

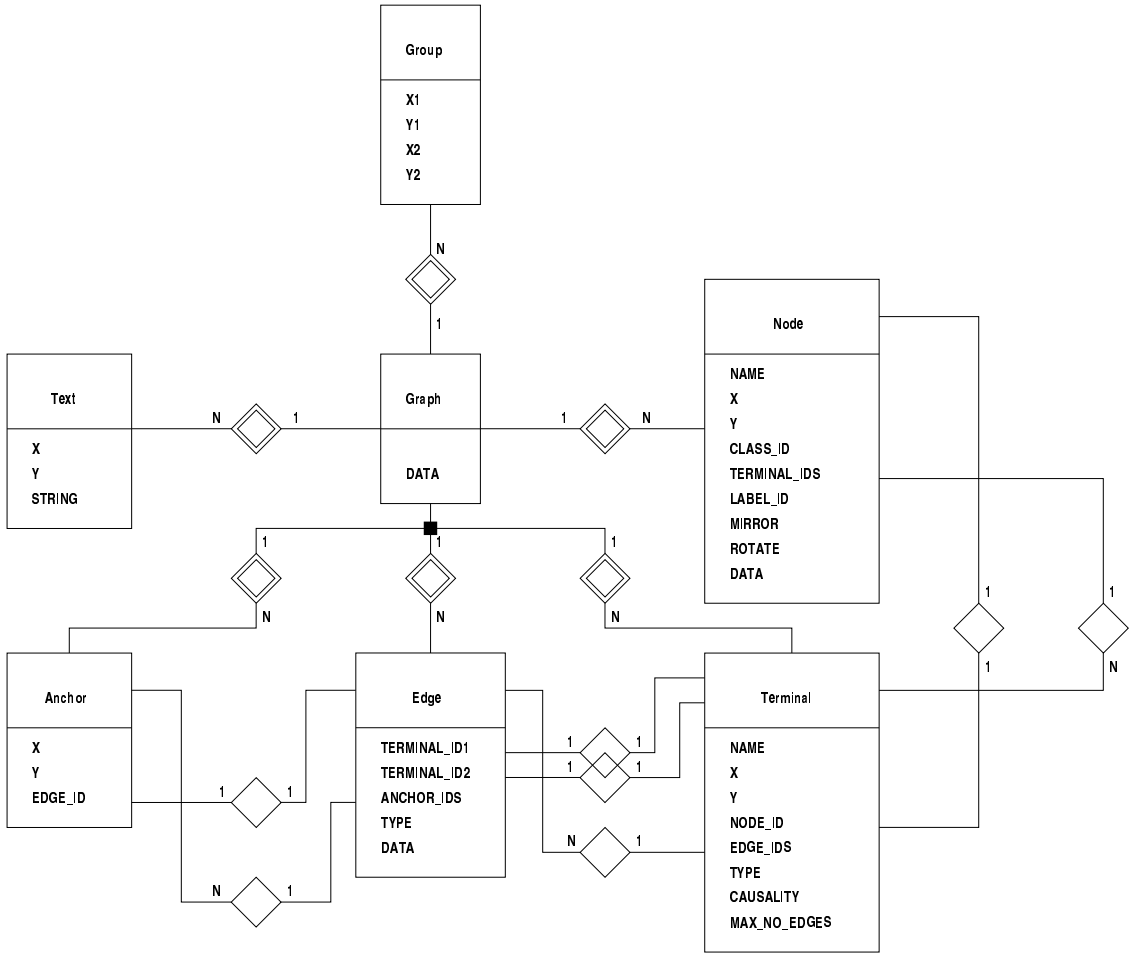

Figure 3.12: Hierarchical Graphical Editor (HGE) Entity-Relationship Diagram

- Info Server (Cinfo)
  This server keeps track of the whereabouts of all other servers. It is the only server which contains global information. In case a new server is started, it will always first register itself with the Info Server.
- Log Server (Clog)
  The Log Server monitors all messages sent over the network from one server to another. The Log Server is able to filter out messages which satisfy certain criteria. These are then sent to a log file. The Log Server is also resposible for managing informational messages, warnings and errors. The latter three messages may be *mapped*, under user control, onto display in a log, display in a popup window, and termination of the complete environment. During different phases of the WEST++ development (prototyping, testing, deployment), different mappings were used.
- Hierarchical Graph Editor Server (HGE)
  The HGE is a generic, highly configurable interactive graphical tool for the construction of annotated graphs. Graphs are data structures consisting of nodes and edges. The HGE can be tuned for a specific formalism/application by setting up a library with configurations for that application. Such libraries can be loaded from within the HGE and immediately change the entire behaviour of the tool. In WEST++, the HGE is mostly used for the construction of coupled models. For this purpose, icons representing components are placed on a canvas and interconnected. Subsequently, models are chosen for each component and the appropriate connection variables are linked. The coupled model can then be exported as a MSL-USER description. Figure 3.12 depicts the HGE's Entity-Relationship Diagram.

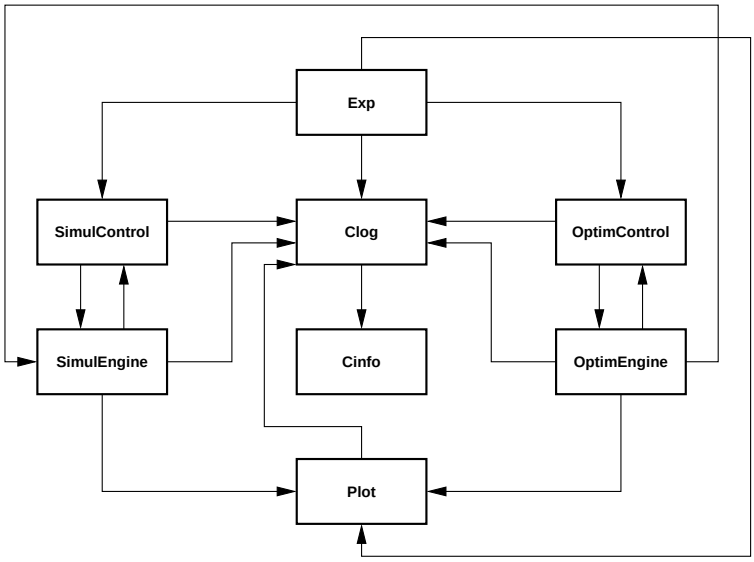

Figure 3.13: Experimentation Environment dependencies

3.4.2 Experimentation Environment architecture

The WEST++ Experimentation Environment currently allows for the simulation and optimisation of models described in the DAE formalism. Just as the Modelling Environment, it operates in a distributed fashion.

Figure 3.13 shows the dependencies between the macro-modules. Arrows point to those macro-modules a particular macro-module depends on. This means that the dependent macro-module cannot operate properly as long as the macro-modules it depends on are not fully available. The WEST++ Experimentation Environment macro-modules are the following:

- Info Server (Cinfo)
  Same as for the Modelling Environment.
- Log Server (Clog)
  Same as for the Modelling Environment.
- Simulation Engine Server (SimulEngine)
  The Simulation Engine Server is capable of simulating the model, encoded in MSL-EXEC (generated by the model compiler from MSL-USER), which has been linked into the Simulation Engine Server itself. Simulation data are either sent to file or to the Plot Server.
- Simulation Control Server (SimulControl)
  The Simulation Control Server acts as an interface between the Simulation Engine Server and the outside world. In most cases it will send incoming messages directly to the Simulation Engine Server. It is needed since the routines of the Simulation Engine Server are not re-entrant.
- Optimisation Engine Server (OptimEngine)
  The Optimisation Engine Server will iteratively start simulations using certain parameter values in order to optimise a certain criterion. To accomplish this, it has to interact intensively with the Simulation Control Server.
- Optimisation Control Server (OptimControl)
  The Optimisation Control Server acts as an interface between the Optimisation Engine Server and the outside world. In most cases it will send incoming messages directly to the Optimisation Engine Server. It is needed since the routines of the Optimisation Engine Server are not re-entrant.
- Plot Server (Plot)
  The Plot Server is able to open multiple windows and plot one or more line graphs in each of these. It does not have a data generator and is therefore totally dependent on other servers for the generation of data.

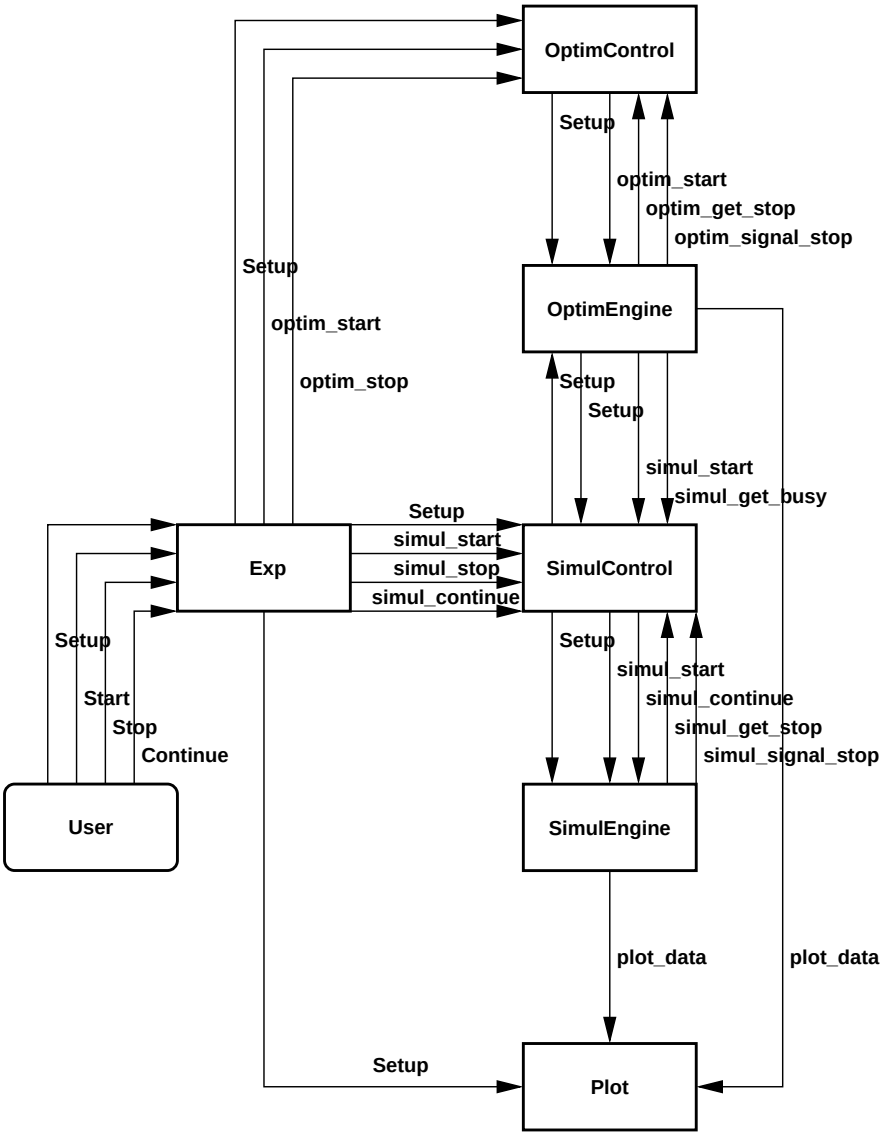

Figure 3.14: Experimentation Environment Calls

- Experimentation Server (Exp)
  This Experimentation Server is the topmost macro-module. Its graphical user-interface allows one to manage the Simulation and Optimisation Servers in a user-friendly way. It does not add any additional functionality to the system, but simply makes existing functionality easier to use.

Figure 3.14 shows the experimentation call graph, which encapsulates the simulator as well as the optimiser call graph. Setup as well as start, stop and continue commands are sent to the Experimentation and subsequently passed on to the appropriate macro-module depending on the experiment type. In case of simulation experiments, commands are passed on to the Simulation Control Server, in case of optimisation experiment, commands are passed on to the Optimisation Control Server.

Figure 3.15 shows the simulator's call graph. The simulator is configured by sending setup commands to the Simulation Control Server. These commands are directly passed on to the Simulation Engine Server. The same goes for the commands which instruct the simulator to start and continue its processing. The stop command just sets a flag inside the Simulation Control Server. During execution, this flag will be checked every now and then by the Simulation Execution Server. If the latter notices that the flag is set, it will abort its execution. When the simulation execution is completely aborted, a notification signal is sent to the

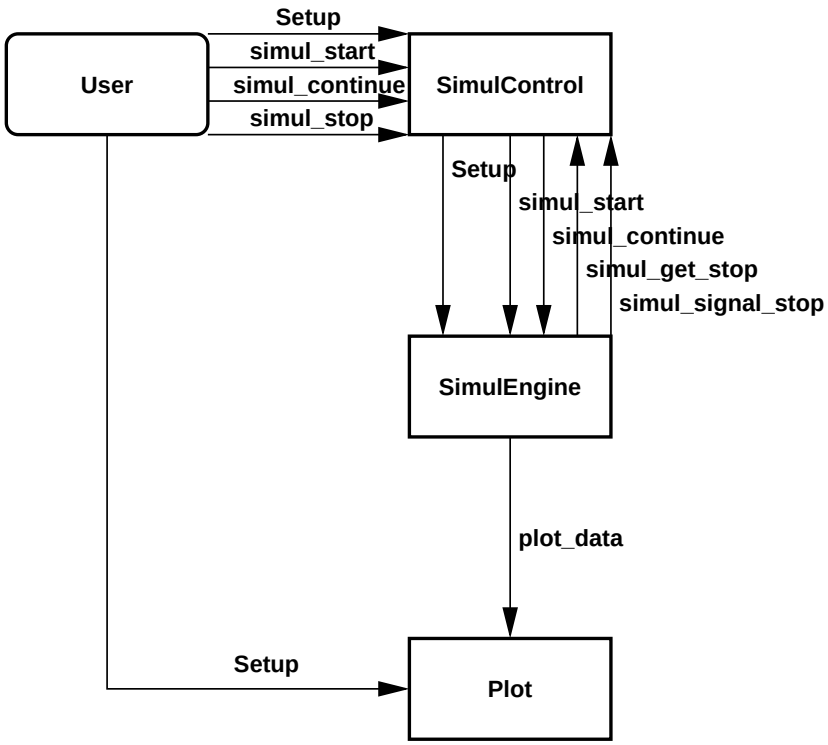

Figure 3.15: Simulation calls

Simulation Control Server. During simulation, data are sent to the Plot Server which is assumed to have been configured at an earlier stage. Configuration of the Plot Server has to occur in a direct way, and not via the Simulation Control Server.

Figure 3.16 shows the optimiser's call graph, which encapsulates the simulator call graph. Setting up the Simulation Engine and Optimisation Engine Servers has to be done by sending setup commands to the Simulation Control and Optimisation Control Servers. These are then passed on to the engines. The Optimisation Control Server can only receive start and stop commands, the continue command is not supported. Start commands are directly passed on to the Optimisation Engine Server, stop commands simply set a flag within the Optimisation Control Server. During optimisation, this flag is checked after each simulation run by the Optimisation Engine Server. When the abortion of an optimisation in completed, a notification signal is sent to the Optimisation Control Server. Simulations are started by the Optimisation Engine Server by sending start commands to the Simulation Control Server. The Optimisation Engine Server consequently checks the Simulation Control Server's busy flag in order to see if the simulation is still running. Note that the Simulation Control Server does not send any notification to the Optimisation Engine Server when the simulation is completed. This is because the simulator does not know by which server it is called. The Optimisation Engine Server also sends data to the Plot Server during optimisation. The Plot Server is supposed to be set up properly beforehand, in a direct way. Figure 3.17 shows the Plot Server's Entity-Relationship Diagram.

Summary

In this chapter, we have presented a Generic Modelling and Simulation Architecture (GMSA). This architecture subsumes distributed simulation architectures such as the High Level Architecture (HLA), and focuses on the life-cycle of models as much as on simulation. Simulation is after all just a small piece in the overall modelling and simulation aided problem solving enterprise.

Models constitute the core of any modelling and simulation system. Modelling languages are a means for model communication (exchange and re-use). The design and implementation of the non-causal, formalism-

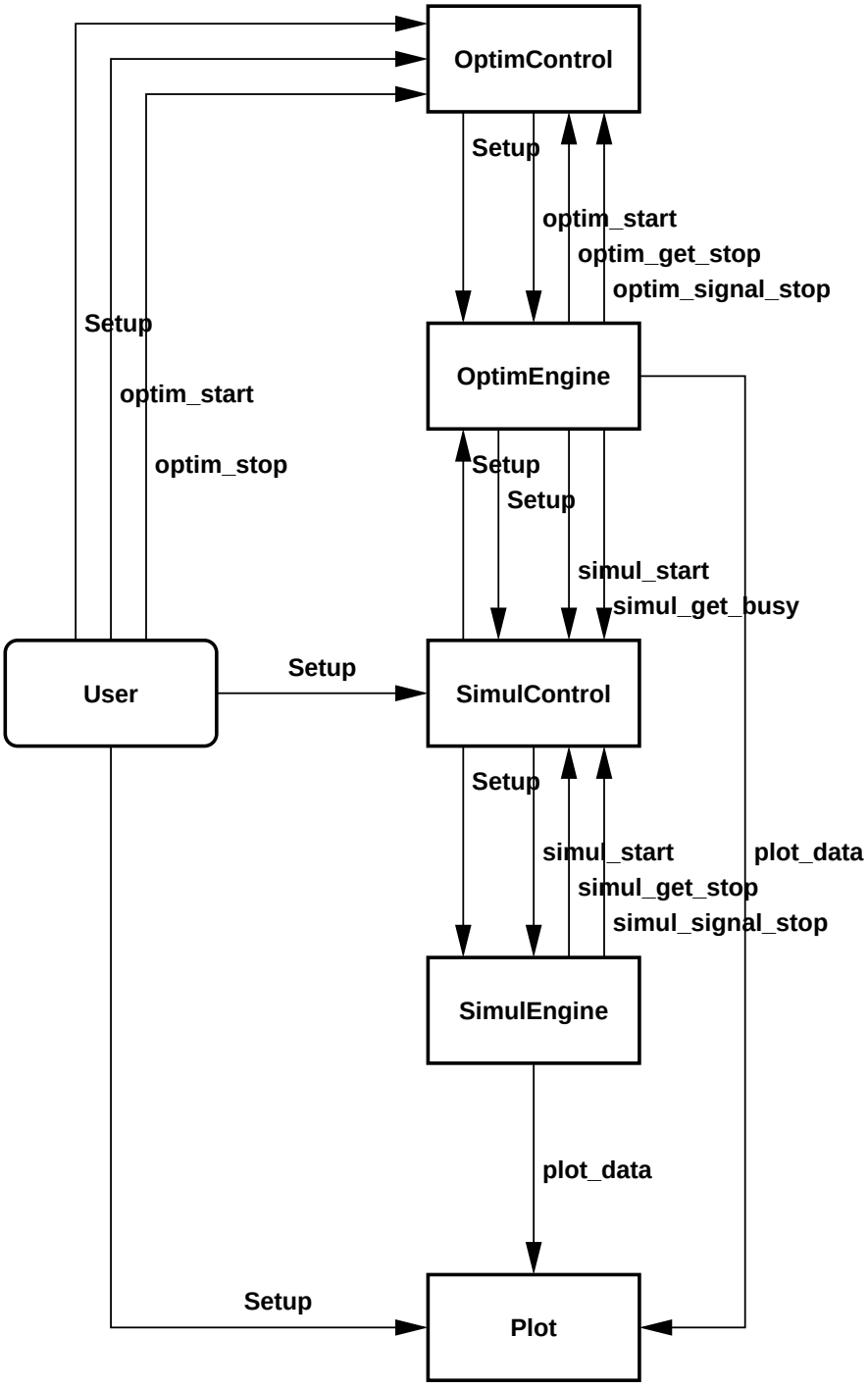

Figure 3.16: Optimization calls

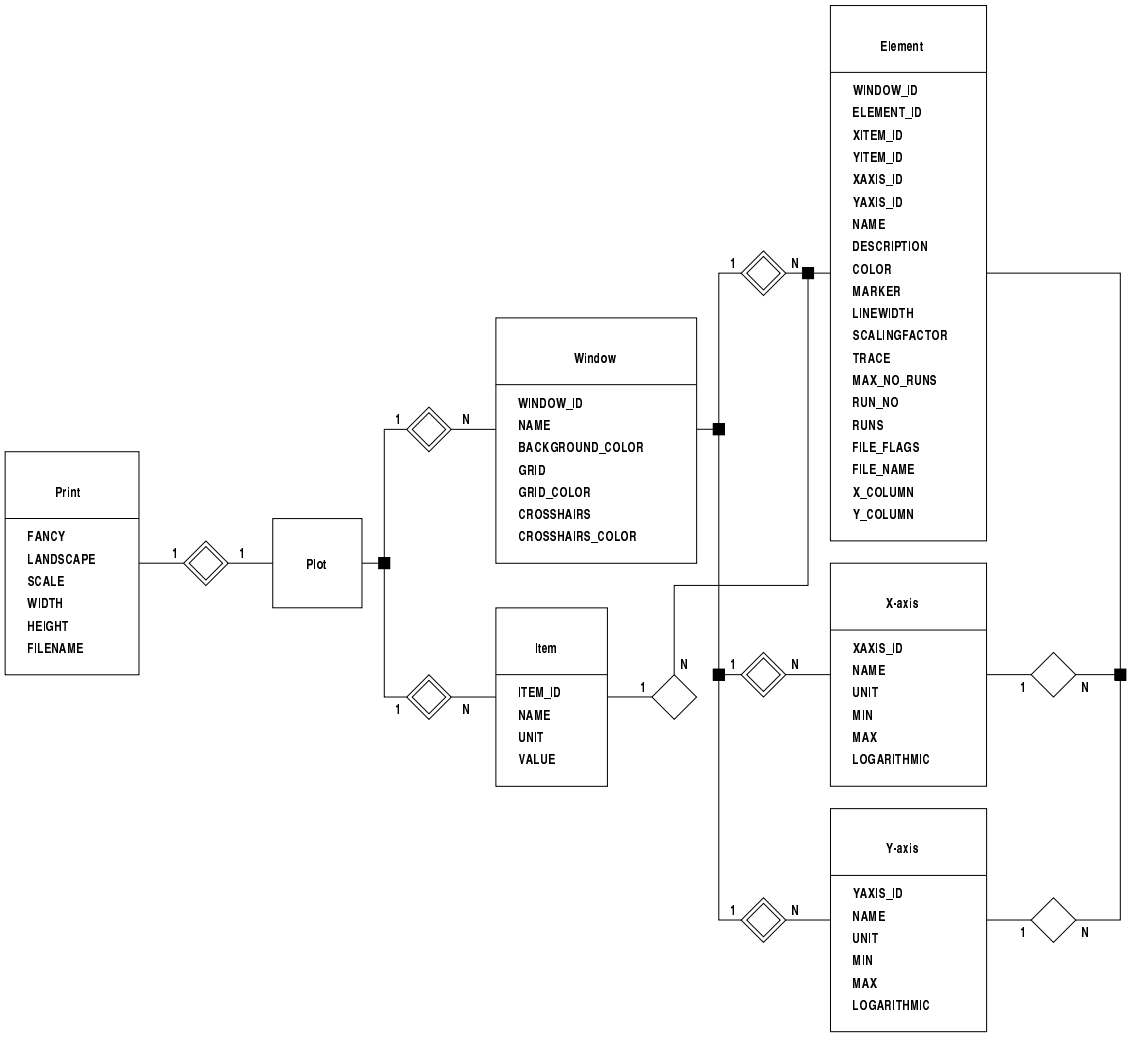

Figure 3.17: Plotter Entity-Relationship Diagram

neutral modelling language MSL-USER is described. Since the conception of MSL-USER, the Modelica language, whose design is partly a spin-off of the SiE Basic Research Working Group and incorporates features of MSL-USER, has become a de facto standard.

The author's most recent work is focused on the unification of multi-formalism modelling with meta-modelling. In meta-modelling, modelling languages are themselves modelled.

The MSL-USER modelling language and its link-level counterpart, MSL-EXEC, are used in the WEST++ distributed, interactive modelling and simulation environment described in the last section of this chapter. Though WEST++ was originally designed with training of Waste Water Treatment Plant (WWTP) operators in mind, it is in fact a general environment. This genericity now proves useful in the study of more and more aspects of WWTP behaviour (including intelligent, model based predictive control) in an increasingly broad context (including river and sewers).

In the next chapter, the WWTP case is elaborated.

4

WWTP Modelling and Simulation

This chapter introduces a full-scale case-study of the concepts presented in previous chapters: the analysis, design, control and optimization of bio-activated sludge waste water treatment plants (WWTPs). Here, the focus is on the structured design of a library of re-usable models in the modelling language MSL-USER rather than on the actual results of the simulations. For detailed simulation results, we refer to a large number of full-scale WWTP analysis and design exercises which have used the WEST++ modelling and simulation environment [VCV98] both at BIOMATH and in industry [RVM$^+$00, CPV$^+$98]. Increasingly, WEST++ is used to study the plant producing the WWTP influent, the sewer system collecting the municipal waste water and transporting it to the treatment plant [MHS$^+$00], the WWTP with its array of controllers [VDCV$^+$99], as well as the river system in which the effluent is dumped [SBH$^+$00, RBH$^+$00, VBH$^+$00], demonstrating the power of our approach at modelling complex systems (and superiority over mainly mono-formalism simulation-oriented –as opposed to modelling-oriented– systems such as Matlab/Simulink).

After a brief introduction to bio-activated sludge waste water treatment plants, modelling of this class of physical systems is set in the context of a general physical systems modelling methodology. At some point in the methodology, behaviour models must be constructed. To describe aeration tanks, the main components of WWTPs, the International Association for Water Quality (IAWQ) (recently renamed International Water Association (IWA)) Activated Sludge Models (ASM1 and ASM2) [HGG$^+$86, HGM$^+$94] are most commonly used. ASM1 is presented and it is shown how its matrix structure can be elegantly represented in MSL-USER. Though traditionally, the settling process in the WWTP clarifier, another crucial part of WWTPs, is modelled by simple models which hardly take into account spatial distribution, it is shown how the PDE to DAE translator described before can be used to simulate, within WEST++, sedimentation processes. Finally, the overall WEST++ modelling and simulation process, from hierarchical, graphical editing, through model calibration, to simulation and optimization, is briefly presented.

4.1 Activated sludge WWTPs

The problem of modelling and simulation of waste water treatment plants is gaining importance as a result of growing environmental awareness [JM98]. Compared to the modelling of well-defined (such as electrical and mechanical) systems, modelling of ill-defined systems such as WWTPs is more complex. In particular, choosing the “right” model is a non-trivial task.

Modelling is an inherent part of the design of a waste water treatment system [HGG$^+$86]. At the fundamental level, a design model may be merely conceptual. The engineer reduces the complex system with which he is dealing to a conceptual image of how it functions. That image then determines the design approach. Often,

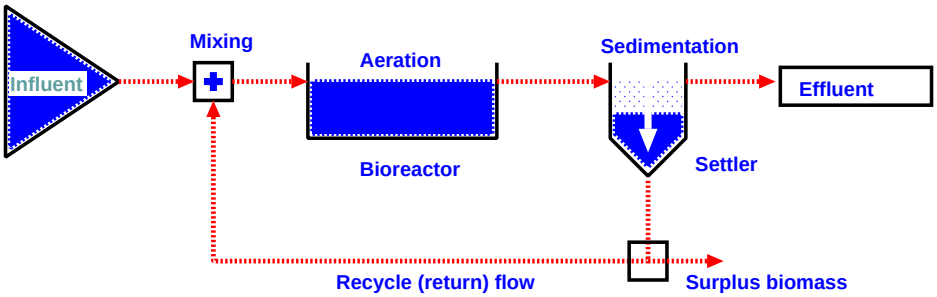

Figure 4.1: WWTP configuration

however, the engineer recognizes that the conceptual model alone does not provide sufficient information for design and thus he constructs a physical model, such as a lab-scale reactor or a pilot plant, on which various design ideas can be tested. Given sufficient time for testing, such an approach is entirely satisfactory. However, the engineer may find that limitations of time and money prevent exploration of all potentially feasible solutions. Consequently, he often turns to the use of mathematical models to further explore the feasible design space. He may devise empirical models which incorporate a statistical approach to mimic the end results obtained by studies on the physical model, or if his conceptual understanding expands sufficiently, he may attempt to formulate mechanistic models. These mechanistic models are the more powerful because they allow extrapolation of the design space to conditions beyond that experienced on the physical model. In this way, many potentially feasible solutions may be evaluated quickly and inexpensively, allowing only the most promising ones to be selected for actual testing in the physical model. The modelling of biological waste water treatment systems has passed through the above deductive-inductive (or modelling for *analysis* followed by modelling for *design* and *control*) sequence of events several times: for the removal of organic matter only, for nitrification, and for nitrogen removal by biological denitrification. Currently, the subject of sedimentation or settling is being studied intensely. In this context, WEST++ is both used for the design and control of new and existing WWTPs as well as to discover new models of sedimentation [A.V98a] and of biofilm growth [RVV99, VDVV00].

The waste water treatment processes dealt with in WEST++ are of the *activated sludge* type [VCV98]. This means the reduction of waste is performed by micro-organisms which convert some of the (non-toxic) waste water components as part of their metabolism. The general WWTP structure (often referred to as “configuration”) is shown in its simplest form in Figure 4.1. In full-scale operational plants, many more components will be present. The basic principles of operation remain the same however. The time-varying load (influent) enters the biological reactor where biodegradable components are converted by means of aerobic metabolism, by a community of micro-organisms (biomass), partly into new microbial biomass and partly into carbon dioxide, water and minerals. Because of the consumption of oxygen in the reactor, it is often named *aeration* tank. In many respects, the aeration basin is comparable to a conventional fermentation reactor. However, the purpose of the process is not to produce as much microbial biomass as possible, but rather to mineralize as much incoming waste materials as possible. It is hereby of paramount importance to minimize biomass production since the latter has to be removed and treated in a subsequent phase. Two important characteristics further distinguish the activated sludge system from conventional microbial fermentations. Firstly, the active biological component comprises not a pure culture but an association of bacteria, yeast, fungi, protozoa and higher organisms. These organisms grow on the incoming waste and interact with one another. Secondly, the sludge consists (in contrast with its qualification “active”) for an important part of dead cells and cell debris. Indeed, young active microbial cells tend to grow in a dispersed way. The system is therefore operated in such a way that the substrate is rate limiting and the microbial biomass is quasi starving. Under these conditions cells grow slowly and in flocs. Because of this, the water in the decantor separates in a clear top layer and a thick bottom layer. Hence, a crucial part of activated sludge treatment is to select a microbial community which mineralizes at a fair rate the incoming waste and thereby produces a minimum of new biomass which furthermore sediments readily and completely out of

the water when the latter reaches the decantor.

The flow of water brings about a constant wash out of the micro-organisms from the reactor to the settling tank (also known as decantor, settler, or clarifier) where both clarification and sludge thickening (through sedimentation) take place. Here, the micro-organisms which acquire a density sufficient to decant, are retained and removed with the underflow. A recycle flow of sludge from the settler to the bioreactor keeps the biomass inside the system (to treat the new influent). When properly operating, the amount of biomass in the system will increase monotonically and a small part of the sludge needs to be removed as excess (see Figure 4.1).

The effluent of the settler is the clarified waste water. This clarified effluent is typically returned to the natural environment (river, lake, ...).

Increasingly, the "system" modelled transcends the WWTP and includes the "environment" (in the engineering sense). The WWTP model is integrated into a conceptual model of the sewer system/polluting plant or of the river (and its natural water purification properties or toxicity tolerance) in which the effluent is dumped. A pertinent example of explicitly modelling the environment in WWTPs concerns the "experimentation environment". Often, sensors are used to monitor biological variables. These variables are difficult to measure, resulting in non-linear distortions of values by the sensor or by non-uniform delays due to sampling and processing. Obviously, failure to include an explicit model of the sensor will completely distort results (no matter how excellent the actual system model is). This is particularly bad if one wants to estimate model parameters by fitting simulation results to measured data (the sensor behaviour will be lumped into the parameters).

Waste water treatment practice has now progressed to the point where the removal of organic matter, nitrification and nitrogen removal by biological denitrification, can be accomplished in a single sludge system. The non-linear dynamics and properties of these biological processes are still not very well understood. As a consequence, a unique model cannot always be identified. This, in contrast to traditional mechanical and electrical systems where the model can be uniquely derived from physical laws. Also, the calibration of waste water treatment models is particularly hard: many expensive experiments may be required to accurately determine model parameters.

In general, models may be represented using different formalisms (*e.g.*, Algebraic (ALG), Ordinary Differential Equations (ODE), Petri Nets, Bond Graphs, ...). Currently, WWTP models are mostly of the ALG and ODE type.

4.2 Physical systems modelling methodology

To allow for computer aided model building and subsequent simulation/experimentation, a *model base* must be constructed. Models in this model base will be used for modular construction (*i.e.*, by hierarchically connecting component blocks) of complex models describing the behaviour of full-scale WWTPs. As is commonly the case, we will choose an appropriate level of abstraction, upon which Idealised Physical Models (IPMs) will be built. Idealised Physical Models [Bro90, TBBA95] represent behaviour at a certain level of abstraction. This often means using lumped parameter models (ODEs) even though the physical system has a spatial distribution (which would require PDE modelling), when the homogeneity assumption is a reasonable approximation.

The steps listed below form a general method for constructing a model base for any application domain (and chosen level of abstraction):

1. Choose an appropriate level of abstraction.
2. Identify relevant quantities.
3. Identify port structures.
4. Build model class hierarchy starting from general (conservation and constraint) laws and refining these for specific cases.

In the following sections, the different steps are further elaborated.

4.2.1 Relevant quantities

First of all, the quantities of interest must be identified. These quantities can be subsequently used to describe the *types* of entities used in modelling (at the I/O system level): constants, parameters, interface variables, and state variables. For numerical computation purposes it is sufficient to specify whether an entity is of Real, Integer, Boolean, or String type. When modelling a particular application domain, more information is available, and should be represented. Once represented in a model, the model compiler can make use of it to determine the legitimacy of the model (*e.g.*, dimensional checking), to perform "intelligent" transformation (*e.g.*, rendering equations in common units), and to generate efficient code (*e.g.*, by means of constraint propagation based on LowerBound and UpperBound information).
In MSL-USER, the type of physical quantities is encoded as a `TYPE PhysicalQuantityType`, a structure shown below:

```
TYPE UnitType
"The type of physical units. For the time being, a string"
= String;

TYPE QuantityType
"The different physical quantities. For the time being, a string"
= String;

TYPE CausalityType
" Causality of entities:
   CIN: input (cause) only
   COUT: output (consequence) only
   CINOUT: input and output (cause and consequence) are allowed
"
= ENUM {CIN, COUT, CINOUT};

TYPE PhysicalNatureType
"The nature of physical variables
 FIELD is used (in the physicalDAE context) to denote
 parameters and constants
"
= ENUM {ACROSS, THROUGH, FIELD};

TYPE PhysicalQuantityType
"The type of any physical quantity"
=
RECORD
{
 quantity   : QuantityType;
 unit       : UnitType;
 interval   : RealIntervalType;
 value      : Real;
 causality  : CausalityType;
 nature     : PhysicalNatureType;
};
```

Basic quantities

The `PhysicalQuantityType` structure can be specialized for specific quantities. Here for example, the physical quantity Area is defined.

```
  CLASS Area
  "A class for Area"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <- "Area";
```

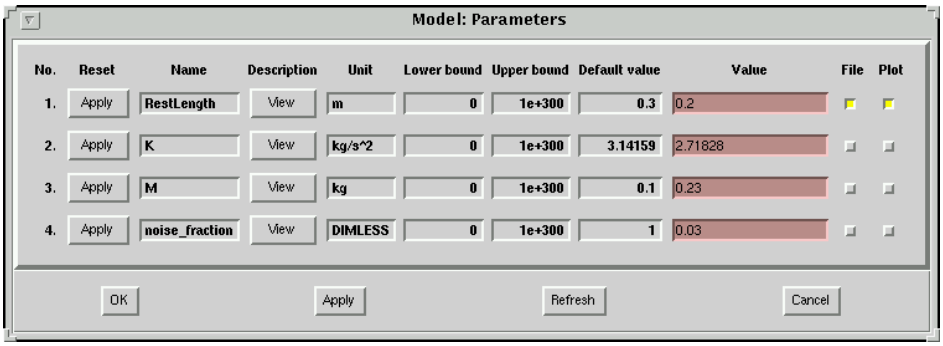

Figure 4.2: Units in the WEST++ experimentation environment

```
 unit      <- "m2";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF:};
:};
```

Definitions of physical quantity types are used to instantiate OBJects of those types. The ISO 1000 standard also defines physical constants such as the universal gravity constant whose MSL-USER description is given as an OBJect declaration below:

```
OBJ UniversalGravityConstant
"Universal gravity constant" : PhysicalQuantityType :=
 {:
  quantity  <- "G";
  unit      <- "m3/(g*s2)";
  value     <- 6.67259E-11;
 :};
```

In Appendix B, the quantities and units defined in the ISO 1000 standard [1292, Car97] are shown encoded in MSL-USER. In the WEST++ environment, the units are not only used during model compilation, but are also passed on to the simulation environment where the user is presented with variable names, values as well as their units (see Figure 4.2).

It is of course possible to describe, in MSL-USER, other consistent unit systems such as Atomic Units [Wil90, Car97].

Quantities typical for WWTPs

Simulation of activated sludge system behaviour, incorporating phenomena such as carbon oxidation, nitrification and denitrification, must necessarily account for a large number of reactions between a large number of *components* (the name given in WWTP modelling to the relevant quantities).

A matter that can cause confusion is the lack of a consistent measure of the concentration of organic material in waste water. Three measures have gained acceptance and are widely used: biochemical oxygen demand (BOD), total organic carbon (TOC), and chemical oxygen demand (COD). Of these it is believed that COD is the superior measure because it alone provides a link between electron equivalents in the organic substrate, the biomass and the oxygen utilized. Furthermore, mass balances can be made in terms of COD. Consequently, the concentrations of all organic materials, including biomass, are in COD units in the ASM1 model, which we will discuss shortly.

The organic matter in a waste water may be subdivided into a number of categories. This grouping of different compounds is an abstraction of reality due to the complexity of the problem. The first important subdivision is based on *biodegradability.* Non-biodegradable organic matter is biologically inert and passes through an activated sludge system unchanged. Two fractions, depending on their physical state, can be identified: *soluble* and *particulate.* Inert soluble organic matter, S_I, leaves the system at the same concentration that it enters. Inert suspended (particulate) organic matter, X_I, becomes enmeshed in the activated sludge and is removed from the system through sludge wastage. Biodegradable organic matter may be divided

S_I	soluble inert organic matter	$[M(COD)L^{-3}]$
S_S	readily biodegradable substrate	$[M(COD)L^{-3}]$
X_I	particulate inert organic matter	$[M(COD)L^{-3}]$
X_S	slowly biodegradable substrate	$[M(COD)L^{-3}]$
$X_{B,H}$	active heterotrophic biomass	$[M(COD)L^{-3}]$
$X_{A,B}$	active autotrophic biomass	$[M(COD)L^{-3}]$
X_P	particulate products arising from biomass decay	$[M(COD)L^{-3}]$
S_O	oxygen (negative COD)	$[M(-COD)L^{-3}]$
S_{NO}	nitrate and nitrite nitrogen	$[M(N)L^{-3}]$
S_{NH}	$NH_4^+ + NH_3$ nitrogen	$[M(N)L^{-3}]$
S_{ND}	soluble biodegradable organic nitrogen	$[M(N)L^{-3}]$
X_{ND}	particulate biodegradable organic nitrogen	$[M(N)L^{-3}]$
S_{ALK}	alkalinity	[molar units]

Table 4.1: Process variables

into two fractions: *readily* biodegradable and *slowly* biodegradable. For purposes of modelling, the readily biodegradable material, S_S is treated as if it were soluble, whereas the slowly biodegradable material, X_S, is treated as if it were particulate.

An overview of quantities used in the IAWQ ASM1 model is given in Table 4.1. As will be described in the sequel, each of these variables denoting a *component* of the waste water, indexes a column in the ASM1 stoichiometry matrix. In MSL-USER, the components are easily described as an enumerated type:

```
TYPE Components = ENUM {H_2O, S_S,...,X_{NH}};
```

Thus, the modeller refers to the components by their name, while, where necessary, the corresponding integer index is used. Though the WEST++ simulator uses the numerical values of the Components indices to address matrix elements, the experimentation environment presents the symbolic name of the index to the user. This reverse mapping is performed by the model compiler when generating MSL-EXEC code. Note how H_2O is explicitly modelled as a component.

Other quantities typical for WWTP modelling are stoichiometric parameters (see Table 4.2) and kinetic parameters (see Table 4.3). Their use in the IAWQ ASM1 model is explained below.

The MSL-USER representation of these is given in Appendix B.

4.2.2 Transferred quantities: terminals

Our ultimate goal is to build complex models by connecting more primitive sub-models or blocks, possibly built up of coupled models themselves. In the case of WWTP models, the sub-model types mostly correspond in a one-to-one relationship to physical entities such as aeration tanks, clarifiers, pumps, splitters, and mixing tanks. This ensures *structural validity* of the assembled models. Note how the building blocks need not match physical objects directly but may correspond to abstract concepts such as processes.

To connect sub-models, these sub-models require connection *ports* or *terminals*. This implies that interaction between the sub-models is assumed to *only* occur through the connections made between their terminals.

Y_H	heterotrophic yield
Y_A	autotrophic yield
f_r	fraction of biomass yielding particulate products
i_{XB}	mass N/ mass COD in biomass
i_{XP}	mass N/ mass COD in products from biomass

Table 4.2: Stoichiometric parameters

$\hat{\mu_H}, K_S, K_{O,H}, K_{NO}, b_H$	heterotrophic growth and decay
$\hat{\mu_A}, K_{NH}, K_{O,A}, b_A$	autotrophic growth and decay
η_g	correction factor for anoxic growth of heterotrophs
k_a	ammonification
k_h, K_X	hydrolysis
η_b	correction factor for anoxic hydrolysis

Table 4.3: Kinetic parameters

This assumption is made explicilty by the closure operation of coupled models which replaces connections by the appropriate algebraic equalities. Thus, choosing the nature (structure) of the terminals will determine the level of abstraction at which interaction is modelled.

In our WWTP models, different terminal types will be used. DataTerminals will represent sensor information to be used in controller blocks. The main terminal type is the WWTPTerminal however. In the basic model base discussed here, only flux of biochemical material is considered. Heat flow for example is not considered. This is one of the modelling assumptions mentioned in the discussion of the ASM1 model and is obvious from the WWTPTerminal definition:

```
CLASS WWTPTerminal
"
 The variables which are passed between WWTP model building blocks
 Currently, we only consider a flux of biochemical material
"
= MassFlux[NrOfComponents;];
```

The WWTPTerminal is a vector of mass fluxes for each of the components taken into consideration in the model. The size of the vector is given by the cardinality of the Components enumerated type and hence depends entirely on how many components the user includes in this type. Note how the actual Component declaration may be given after all other declarations (MSL-USER has set rather than sequential semantics).

```
OBJ NrOfComponents
 "
  The number of biological components considered in the WWTP models
 "
 : Integer := Cardinality(Components);
```

A few assumptions are made:

1. The same (WWTPTerminal) terminals are used everywhere in a configuration for biochemical transport.

2. All WWTP terminals of a model have the same cardinality. This is enforced thanks to the way we define Components and WWTPTerminal.

3. The number of components in WWTPTerminal is the same as the number of components (columns) in the stoichiometry matrix (which will be defined later on). Again, this is enforced thanks to the way we define Components and WWTPTerminal.

While the model compiler will check whether (type-) compatible terminals are connected and how many connections are allowed to/from a terminal, the graphical modelling environment will already perform a check during interactive modelling.

In the current model base, *only* biochemical mass transport is modelled by the connections. Concentration of a component *i* in a sub-model will be derived as M_i/V, the mass of the *i*-th component over the total volume of all components (including the most prominent one, H_2O). In the Bond Graph formalism, the conjugate variable of flow, pressure would be modelled explicitly too. Here, the pressure is assumed constant (the atmospheric pressure) and not represented explicitly. In Bond Graphs ideal physical connection terminals always contain conjugate variables together: mechanical (translational) force and velocity, electrical current and voltage, and hydraulic flowrate and pressure.

4.2.3 Inheritance hierarchy

At this point, models must be constructed for each type of building block. This is achieved in the form of a class *inheritance* hierarchy. Hereby, maximum *re-use* and *clarity* is achieved. Clarity is a direct result of the relationship between the inheritance hierarchy on the one hand and the different levels of specificity of the models on the other hand.

In the generic model base, GenericModelType is defined:

```
TYPE GenericModelType
"The signature of the generic part of any
 (whatever the formalism) model
"
=
RECORD
{
 comments   : String;
 interface  : SET_OF (InterfaceDeclarationType);
  // declared objects must be interfaces
 parameters : SET_OF (ParameterDeclarationType);
  // declared objects must be parameters
};
```

It shows how any model has a description (comments) part, an interface set and a parameter set. This type can be extended to describe the essence of coupled models:

```
TYPE CoupledModelType "The signature of a coupled (network) model"
EXTENDS GenericModelType WITH
RECORD
{
 sub_models : SET_OF (ModelDeclarationType);
 coupling   : SET_OF (CouplingStatementType);
};
```

For basic models in the DAE formalism, DAEModelType prescribes the structure:

```
TYPE DAEModelType
"The signature of a Differential Algebraic Equation (DAE) model
 within DAEModelType models, connect() has the following
 (flattening) semantics:
   quantity and unit are checked for equality
```

```
   equations are generated to equal (=) all algebraic and state
   variables all other labels are ignored
"
EXTENDS GenericModelType WITH
RECORD
{
 independent : SET_OF (ObjectDeclarationType);
 // independent variable (time)
 state       : SET_OF (PhysicalQuantityType); // variables
               // those variables occurring in
               // DERIV(v, [t]) statements are
               // derived state variables
 initial     : SET_OF (EquationType);
 equations   : SET_OF (EquationType);
 terminal    : SET_OF (EquationType);
};
```

Both CoupledModelType and DAEModelType are extensions of GenericModelType which means they inherit its structure (and add to it). The synonym PhysicalDAEModelType is used for DAEModelType when we use it for modelling physical systems:

```
TYPE PhysicalDAEModelType
"within physicalDAEModelType models, connect() has the following
 (flattening) semantics:
   quantity and unit are checked for equality
   quantity and unit are checked for equality
   equations are generated to equal (=) all across variables
   equations are generated to sum all through variables to zero
   all other labels are ignored
"
= DAEModelType;
```

The top-level inheritance hierarchy is hence

```
GenericModelType
|
|____ CoupledModelType
|____ DAEModelType == PhysicalDAEModelType
```

Some of the model classes are derived directly from PhysicalDAEModelType. The ones listed directly below are models of the settler [Tay82, ML93]. The Takacs model is a hand-crafted discretized (10-layer) model of the settling process. As was described before and will be demonstrated later in this chapter, WEST++-PDE is able to discretize a class of PDE models of the settling process. Once discretized, these models are of ordinary PhysicalDAEModelType.

```
PhysicalDAEModelType
|
|____ Takacs
|
|____ Otterpohl_and_Freund_for_Secondary_Clarifier
|
|____ Generated from PDE with WEST++-PDE
```

The sensor, controller, data filter, and transformer models are also derived from PhysicalDAEModelType. These models do not describe physical processes involving (transport of) matter and energy and do hence not adhere to physical laws. Though not subject to physical constraints, they do deal with the values of physical variables.

```
PhysicalDAEModelType
|
|____ Sensors
|         |____ FlowSensor
```

```
|          |____ DOSensor
|          |____ NH4Sensor
|          |____ NO3Sensor
|          |____ XSensor
|
|____ Controllers
|          |____ P_controller
|          |____ PI_controller
|          |____ PID_controller
|          |____ On_off_controller
|          |____ Backlash_controller
|          |____ Saturation
|          |____ RateLimiter
|          |____ DeadZone
|          |____ CoulombFriction
|
|____ Data_filters
|          |____ Sample_and_Hold
|          |____ Noise
|          |____ First_Order_Time_Lag
|          |____ TackacsHeigthOfSludge
|          |____ ASU_OUR
|          |____ ASU_Kla
|          |____ ASU_volume
|          |____ SDT_Volume
|          |____ TimeDelay
|
|____ Transformers
           |____ BOD_COD_transformer
           |____ SinusGenerator
           |____ DoubleSineGenerator
           |____ BlockGenerator
```

One of the oldest types of bioactivated sludge WWTP systems, trickling filters using biofilms grown on inert objects, has received considerable attention recently [RVV99, VDVV00, VVB$^+$00].

```
PhysicalDAEModelType
|
|____ Trickling_Filter
           |____ Rauch
```

As was mentioned before, WEST++ is not only used to model the activated sludge waste water treatment process but also parts of the environment, in particular, the (compartmentalized) river in which the treated effluent is dumped:

```
PhysicalDAEModelType
|
|____ River_models
           |____ Bulk_Benthic_River
           |____ BenthicRiver
```

The shallowness of the above inheritance hierarchy reflects the diverse nature of the different model types, not allowing for much re-use.

Now we look into the development of WWTPAtomicModel, from which many other model types are derived. Once the biochemical components considered are declared

```
 TYPE Components
 "
  The biological components considered in the WWTP models
```

```
 "
  = ENUM {H2O, S_I, S_S, S_O, S_NO, S_ND, S_NH, S_ALK,
          X_I, X_S, X_BH, X_BA, X_P, X_ND};
```

the basic mass balance equation for each of the components can be written:

```
    // The FluxPerComponent is the sum of all
    // incoming (positive) and outgoing (negative) fluxes

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
      state.FluxPerComponent[Comp_Index] =

    // If not only WWTPTerminal type terminals are present in the interface
    // (e.g., also ControlTerminal), we have to select only
    // those terminals from the interface which are of
    // WWTPTerminal type (or any SUBtype such as InWWTPTerminal of it)
    // as those are the only ones for which the mass balance law holds.

    (SUMOVER In_Terminal IN {SelectByType(interface,InWWTPTerminal)}:
    In_Terminal[Comp_Index])+
    (SUMOVER Out_Terminal IN {SelectByType(interface,OutWWTPTerminal)}:
    Out_Terminal[Comp_Index]);};

    // The mass balance equations.
    // These are composed of a term due to incoming and
    // outgoing fluxes and of a term due to biochemical
    // interactions between components.

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
      DERIV(state.M[Comp_Index],[independent.t]) =
      state.FluxPerComponent[Comp_Index]
      +state.ConversionTermPerComponent[Comp_Index];};
```

The rate of change of a component's mass consists of the net result of incoming and outgoing mass flux augmented with a reaction term due to biochemical interactions between different components. This reaction term is described in more detail in the following section. Logically, the next level (below WWTPAtomicModel) of classification would be to distinguish between models without volume (point-model abstractions where no mass is accumulated and hence no reactions occur) and models with volume. For efficiency reasons, the model without volume is encoded directly as a sub-class of PhysicalDAEModelType. For models with volume, the distinction must be made between models where volume is considered constant and those where volume may vary. This class hierarchy is depicted below and elaborated in `wwtp.base.msl` in Appendix B.

```
PhysicalDAEModelType
|
|_____WWTPAtomicModelWithoutVolume
|          |
|          |____ Pointsettler
|          |          |____ PrimaryPointSettler
|          |          |____ SecondaryPointSettler
|          |
|          |____ Splitter_combiners
|          |          |____ RelTwoSplitter
|          |          |____ AbsTwoSplitter
|          |          |____ TwoCombiner
|          |          |____ RelThreeSplitter
|          |          |____ AbsThreeSplitter
|          |          |____ ThreeCombiner
|          |
|          |____ Anaerobic_digestion
|          |          |____ Rozzi
|          |          |____ Desjardins & Lessard
|          |          |____ Gujer
```

```
|         |
|         |____ OtterpohlAndFreund (primary clarifier)
|         |____ Tay
|         |____ Lessard_and_Beck
|         |____ Marsili_Libelli
|
|____ WWTPAtomicModel
          |
          |____ WWTPAtomicModelWithVolume
                  |
                  |____ WWTPAtomicModelWithVariableVolume (Q_e = f(water volume in tank))
                  |         |
                  |         |____ BufferTank
                  |         |____ VarVolumeConversionModel
                  |                 |____ VarVolumeASMConversionModel
                  |                         |____ VarVolumeInOutIAWQ
                  |                                 |____ VarVolumeActivatedSludgeUnit
                  |
                  |____ WWTPAtomicModelWithVolumePumped (Q_e : parameter = pumped flow)
                  |         |
                  |         |____ StormTank
                  |         |____ DetentionTank
                  |
                  |____ WWTPAtomicModelWithFixedVolume (Q_e = Q_i)
                          |
                          |____ EqualisationTank
                          |____ FixVolumeConversionModel
                                  |____ FixVolumeASMConversionModel
                                          |____ FixVolumeInOutIAWQ
                                                  |____ FixVolumeActivatedSludgeUnit
```

4.3 The IAWQ ASM 1 model

The heart of the operation of an activated sludge waste water treatment plant is the aeration tank. Crucial in modelling the aeration tank's behaviour is to realistically model the inter-component biochemical reactions. These reactions must be representative of the most important fundamental processes occurring within the system. Furthermore, the model should quantify both the *kinetics* (rate-concentration dependence) and the *stoichiometry* (relationship that one component has to another in a reaction) of each process. Identification of the major processes and selection of the appropriate kinetic and stoichiometric expressions for each are the major conceptual tasks during development of a mathematical model.

Realizing the benefits to be derived from mathematical modelling, the International Association on Water Pollution Research and Control (IAWPRC) formed a task group in 1983 to promote the development of, and facilitate the application of, practical models to the design and operation of biological waste water treatment systems. The goals were to review existing models and to reach a consensus concerning the simplest one having the capability of realistic predictions of the performance of single sludge systems carrying out carbon oxidation, nitrification, and denitrification [HGG+86]. The model was to be presented in a way that made clear the processes incorporated into it and the procedures for its use.

The task group chose a *matrix format* for the presentation of the model. The first step in setting up the matrix is to identify the *components* of relevance in the model. The second step in developing the matrix is to identify the biological *processes* occurring in the system; *i.e.,* the conversions or transformations which affect the components listed.

13 components are considered in the IAWQ ASM1 model as discussed before: soluble and particulate inert organic matter, readily and slowly biodegradable substrate, active heterotrophic and autotrophic biomass, particulate products arising from biomass decay, oxygen, nitrate and nitrite nitrogen, NH_4^+ and NH_3 nitrogen, soluble biodegradable organic nitrogen, particulate biodegradable organic nitrogen. There are 8 pro-

Process	Stoichiometry (ν)			Kinetics (ρ)
	Biomass X_B $[M(COD)L^{-3}]$	Substrate S_S $[M(COD)L^{-3}]$	Oxygen S_O $[M(-COD)L^{-3}]$	Process Rate $[ML^{-3}T^{-1}]$
Growth	1	$-\frac{1}{Y}$	$-\frac{1-Y}{Y}$	$\frac{\hat{\mu}S_S}{K_S+S_S}X_B$
Decay	-1		-1	bX_B

Table 4.4: Process stoichiometry and kinetics

cesses: aerobic and anoxic growth of heterotrophs, aerobic growth of autotrophs, decay of heterotrophs and autotrophs, ammonification of soluble organic nitrogen, hydrolysis of entrapped organics and of entrapped organic nitrogen.

4.3.1 Matrix format and notation

The matrix format decided upon by the IAWQ task group is elaborated here. Consider the situation in which heterotrophic bacteria are growing in an aerobic environment by utilizing a soluble substrate as a source of carbon and energy. In one simple conceptualization of this situation, two fundamental processes occur: the biomass increases by cell growth and decreases by decay. Other events, such as oxygen utilization and substrate removal, also occur, but these are not considered to be fundamental because they result from biomass growth and decay and are coupled to them through the system stoichiometry. The simplest model of this situation must consider the concentrations of three components: biomass, substrate, and dissolved oxygen. The matrix incorporating the fate of these three components in the two fundamental processes is shown in Table 4.4.

The first step in setting up the matrix is to identify the *components* of relevance in the model. In this scenario these are biomass, substrate and dissolved oxygen, which are listed, with units, as columns in Table 4.4. In conformity with IAWPRC nomenclature, insoluble constituents are given the symbol X and the soluble components S. Subscripts are used to specify individual components: B for biomass, S for substrate and 0 for oxygen.

The second step in developing the matrix is to identify the *biological processes* occurring in the system;*i.e.*, the conversions or transformations which affect the components listed. Only two processes are included in this example: aerobic growth of biomass and its loss by decay. These processes are listed in the leftmost column of the table.

The *kinetic expressions* or *rate equations* for each process are recorded in the rightmost column of the table in the appropriate row. Process rates are denoted by ρ_j where j corresponds to the process index.

If we were to use the simple Monod-Herbert model for this situation, the rate expressions would be those in Table 4.4. The Monod equation, ρ_1, states that growth of biomass is proportional to biomass concentration in a first order manner and to substrate concentration in a mixed order manner. The Herbert expression, ρ_2, states that biomass decay is first order with respect to biomass concentration. The kinetic parameters used in the rate expressions are

- Maximum specific growth rate: $\hat{\mu}$
- Half maximum saturation coefficient K_S (since $\mu = \frac{\hat{\mu}}{2}$ when $K_S = S_S$)
- Specific decay rate: b

The elements within the table comprise the stoichiometric coefficients, ν_{ij}, which set out the mass relationships between the components in the individual processes. For example, growth of biomass $(+1)$ occurs at the expense of soluble substrate $(-\frac{1}{Y}$, Y is the yield parameter); oxygen is utilized in the metabolic process $-\frac{1-Y}{Y}$. The coefficients ν_{ij} are simplified by working in consistent units. In this case, all organic constituents

have been expressed as equivalent amounts of chemical oxygen demand (COD); likewise, oxygen is expressed as negative oxygen demand. The sign convention used in the table is negative for consumption and positive for production.

In matrix form, we obtain a stoichiometry matrix

$$\nu = \begin{pmatrix} 1 & -\frac{1}{Y} & -\frac{1-Y}{Y} \\ -1 & 0 & -1 \end{pmatrix}$$

and a kinetics vector

$$\rho = \begin{pmatrix} \frac{\hat{\mu}S_S}{K_S+S_S}X_B \\ bX_B \end{pmatrix}$$

Within a system, the concentration of a single component may be affected by a number of different processes. An important benefit of the matrix representation is that it allows rapid and easy recognition of the fate of each component, which aids in the preparation of mass balance equations. This may be seen by moving down the column representing a component.

The basic equation for a mass balance within any defined system boundary is:

$$\forall i \in Components : \frac{dM_i}{dt} = \Phi_i^{in} - \Phi_i^{out} + r_i.$$

The input and output terms are transport terms and depend upon the physical characteristics of the system being modelled. The system reaction term, r_i, is obtained by summing the products of the stoichiometric coefficients ν_{ij} and the process rate expression ρ_j for the component *i* being considered in the mass balance:

$$r_i = \sum_j \nu_{ji}\rho_j$$

For example the rate of reaction, *r*, for biomass X_B, at a point in the system would be:

$$r_{X_B} = \frac{\hat{\mu}S_S}{K_S+S_S}X_B - bX_B;$$

for soluble substrate S_S it would be:

$$r_{S_S} = -\frac{1}{Y}\frac{\hat{\mu}S_S}{K_S+S_S}X_B;$$

for dissolved oxygen S_O it would be:

$$r_{S_O} = -\frac{1-Y}{Y}\frac{\hat{\mu}S_S}{K_S+S_S}X_B - bX_B.$$

To create the mass balance for each component within a given system boundary (*e.g.*, a completely mixed reactor), the conversion rate would be combined with the appropriate flow terms for the particular system.

Another benefit of the matrix is that continuity may be checked by moving across the matrix, provided consistent units have been used because then the sum of the stoichiometric coefficients must be zero. This can be demonstrated by considering the decay process. Recalling that oxygen is negative COD so that its coefficient must be multiplied by -1, all COD lost from the biomass because of decay must be balanced by oxygen utilization. Similarly, for the growth process, the substrate COD lost from solution due to growth minus the amount converted into new cells must equal the oxygen used for cell synthesis.

Table 4.5 presents the full ASM1 matrix describing the process kinetics and stoichiometry for carbon oxidation, nitrification, and denitrification. The meaning and units for stoichiometric and kinetic parameters as well as process variables (components) used were given in Tables 4.2, 4.3, and 4.1.

Process	Stoichiometry (ν)												Kinetics (ρ)
	S_I	S_S	X_I	X_S	$X_{B,H}$	$X_{A,B}$	X_P	S_O	S_{NO}	S_{NH}	S_{ND}	X_{ND}	S_{ALK}	
Aerobic growth heterotrophs		$-\frac{1}{Y_H}$			1			$-\frac{1-Y_H}{Y_H}$		$-i_{XB}$			$-\frac{i_{XB}}{14}$	$\hat{\mu}_H \frac{S_S}{K_S+S_S} \frac{S_O}{K_{O,H}+S_O} X_{B,H}$
Anoxic growth heterotrophs		$-\frac{1}{Y_H}$			1				$-\frac{1-Y_H}{2.86Y_H}$	$-i_{XB}$			$-\frac{i_{XB}}{14\cdot 2.86Y_H} - \frac{i_{XB}}{14}$	$\hat{\mu}_H \frac{S_S}{K_S+S_S} \frac{K_{O,H}}{K_{O,H}+S_O} \times \frac{S_{SO}}{K_{NO}+S_{NO}} \eta_g X_{B,H}$
Anoxic growth autotrophs						1	$-\frac{4.75-Y_A}{Y_A}$	$\frac{1}{Y_A}$	$-i_{XB} - \frac{1}{Y_A}$			$-\frac{i_{XB}}{14} - \frac{1}{7Y_A}$	$\hat{\mu}_A \frac{S_{SH}}{K_{NH}+S_{NH}} \frac{S_O}{K_{O,A}+S_O} X_{B,A}$	
Decay of heterotrophs			$1-f_r$		-1		f_r					$i_{XB} - f_P i_{XP}$		$b_H X_{B,H}$
Decay of autotrophs			$1-f_r$			-1	f_r					$i_{XB} - f_P i_{XP}$		$b_A X_{B,A}$
Ammonification of soluble organic nitrogen										1	-1		$\frac{1}{14}$	$k_a S_{ND} X_{B,H}$
Hydrolysis of entrapped organics		1		-1										$k_h \frac{X_S/X_{B,H}}{K_X+X_S/X_{B,H}} [\frac{S_O}{K_{O,H}+S_O} + \eta_h \frac{K_{O,H}}{K_{O,H}+S_O} \frac{S_{NO}}{K_{NO}+S_{NO}}] N_{B,H}$
Hydrolysis of entrapped organic nitrogen											1	-1		$\rho_7 \frac{X_{ND}}{X_S}$

Table 4.5: IAWQ ASM1 matrix

In MSL-USER, the reaction(conversion) term

$$\forall i \in Components : \frac{dM_i}{dt} = \Phi_i^{in} - \Phi_i^{out} + r_i$$

is encoded in a straightforward manner as:

```
equations <-
{
 {FOREACH Comp_Index IN {1 .. NrOfComponents}:
   state.ConversionTermPerComponent[Comp_Index] =
    SUMOVER Reaction_Index IN {1 .. NrOfReactions}:
     (parameters.Stoichiometry[Reaction_Index][Comp_Index]
     *state.Kinetics[Reaction_Index])
    *state.V;};
}
```

The MSL-USER compiler will expand the above few lines into the appropriate equations based on the ASM1 matrix given in the `wwtp.VolumeASM1ConversionModel.body.msl` library (Appendix B). These equations will subsequently be manipulated to generate correct and efficient simulation code. Note that components which are transported but do not react (*i.e.*, only hydraulics, no chemistry or biology) have a column of zeroes in the stoichiometry matrix. In MSL-USER, by default, when a variable is not given a value, the initial value is 0. Thus, if we don't assign anything to elements of the stoichiometry matrix, it is a matrix of zeroes, which means no biochemical reactions take place.

4.3.2 Modelling assumptions and restrictions

When a waste water treatment system, like any system, is to be modelled, a certain number of simplifications and assumptions are made to make the model tractable. Some of these are associated with the physical system itself, whereas others concern the mathematical model. Often these simplifications and assumptions are implicit, which may cause the user to overlook them. When that happens there is a strong likelihood that they will be violated, which could destroy the utility of the results. To prevent this from happening the following section explicitly enumerates the major assumptions, restrictions and constraints associated with the model and the physical system it is designed to simulate. This constitutes the *Experimental Frame* of the model.

1. The system operates at constant temperature. Because many of the coefficients are functions of temperature, their functionality would have to be explicitly expressed in the rate expressions, ρ_j, to include time-variant temperature fluctuations.

2. The pH is constant and near neutrality. Although it is known that the pH influences many of the coefficients, few expressions are available for expressing that influence. Consequently, constant pH has been assumed. The inclusion of alkalinity in the model allows the user to detect potential problems with pH control.

3. No consideration has been given to changes in the nature of the organic matter within any given fraction (e.g. the readily biodegradable organic matter). In other words, the coefficients in the rate expressions have been assumed to have constant values. It is still possible, however, for the concentration associated with any influent fraction to vary with time. Thus, while variable input loadings can be handled, changes in *waste character* cannot.

4. The effects of limitations of nitrogen, phosphorus, and other inorganic nutrients on the removal of organic substrate and on cell growth have not been considered. It is well known that inadequate inorganic nutrients can lead to problems in sludge settleability.

5. The correction factors for denitrification, η_g and η_h are fixed and constant for a given waste water.

6. The coefficients for nitrification are assumed to be constant and to incorporate any inhibitory effects that other waste constituents are likely to have on them.

7. The heterotrophic biomass is homogeneous and does not undergo changes in species diversity with time. This assumption is inherent in the assumption of constant kinetic parameters. It also means that the effects of substrate concentration gradients, reactor configuration, etc. on sludge settleability are not considered.

8. The entrapment of particulate organic matter in the mass is assumed to be instantaneous.

In MSL-USER, constraints are, wherever possible, encoded in types. This allows for static (compile-time) checking. Valid Y_A values for example must lie in the range $[0, 4.57]$. Furthermore, `YieldForAutotrophicBiomass` specializes `PhysicalQuantityType` and thus inherits the structure as well as all the constraints imposed on that supertype.

```
CLASS YieldForAutotrophicBiomass
"A class for YieldForAutotrophicBiomass"
SPECIALISES PhysicalQuantityType :=
{:
  quantity  <- "Y_A";
  unit      <- "gCOD/gN";
  interval  <- {: lowerBound <- 0; upperBound <- 4.57 :};
:};
```

If this is not possible, assertions are added to the model which will generate warnings at run-time (dynamic checking).

4.4 PDE modelling of clarification

Understanding the interplay between the hydraulic and biochemical facets of the clarification process is currently an area of intense research. Until recently, very simple models were used such as a point splitter where the whole clarification process is abstracted as removing a certain percentage of purified water. This simple model and the more elaborate Tackacs model were mentioned in the class hierarchy and their MSL-USER representation is found in the libraries in Appendix B. The PDE to ODE transformation developed here is a contribution to the research into more realistic models for the clarification process.

In this section, we present the results of the description, automatic discretization and simulation of two test case partial differential equations (PDEs) in WEST++: the PDEs for batch and continuous sedimentation in a 1-D clarifier model, discussed in [A.V98a, A.V98b]. Previously, examples were provided for the discretization of the domain into several elements, setting up of collocation points for a given choice of the parameters (α, β), and casting the algebraic equations arising from the different boundary conditions into matrix form.

4.4.1 The continuous case

As discussed before, the PDE for the concentration $X(z,t)$ in a 1-D clarifier of length L with continuous sedimentation is given by:

$$\frac{\partial X(z,t)}{\partial t} = -\frac{\partial F(X(z,t))}{\partial z} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}. \tag{4.1}$$

where the convective flux F is the sum of the gravitational settling flux $G(X(z,t))$ and the bulk flow. The bulk flow consists of the downward underflow flux, the source flux coming into the clarifier at an inlet located at $z = z_f$, and the effluent. There are two cases to consider for the form of this bulk flow: the absence or presence of a pump in the clarifier. In the former case, the effluent simply overflows, while in the latter, the pump is used to draw the effluent out. In the PDE above, D_0 is the diffusivity or dispersion coefficient.

The boundary conditions for the PDE in the continuous case are given by:

$$\left.\frac{\partial X(z,t)}{\partial z}\right|_{z=0} = 0$$
$$\left.\frac{\partial X(z,t)}{\partial z}\right|_{z=L} = 0. \tag{4.2}$$

In the following, we present only the final forms of the PDEs as implemented in WEST++. Details of how to arrive at these, for the two cases mentioned above, are presented in [A.V98b].

Effluent overflow

If there is no pump present at the top of the clarifier, the PDE is given by:

$$\begin{aligned}\frac{\partial X(z,t)}{\partial t} &= -\left[\left(1-nX(z,t)\right)\, v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + X_f(t)\,\frac{Q_f(t)}{A}\,\delta(z-z_f) \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}.\end{aligned} \tag{4.3}$$

Here, v_0 and n are the parameters of the Vesilind settling velocity, $Q_u(t)/A$ is the underflow velocity, with A being the area of cross subsection of the clarifier. $X_f(t)$ and $Q_f(t)/A$ are the source concentration and source velocity respectively. For the numerical implementation, the delta function above is represented by a rectangular unit pulse of width 2σ centred around $z = z_f$. We then have three PDEs for the different regions of the clarifier: above the inlet, at the inlet, and below the inlet.

These three PDEs are given by:

1. for $\{0 \le z \le z_f - \sigma\}$:

$$\begin{aligned}\frac{\partial X(z,t)}{\partial t} &= -\left[\left(1-nX(z,t)\right)\, v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2};\end{aligned} \tag{4.4}$$

2. for $\{z_f - \sigma < z < z_f + \sigma\}$:

$$\begin{aligned}\frac{\partial X(z,t)}{\partial t} &= -\left[\left(1-nX(z,t)\right)\, v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + X_f(t)\,\frac{Q_f(t)}{A}\,\frac{1}{2\sigma} \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2};\end{aligned} \tag{4.5}$$

3. for $\{z_f + \sigma \le z \le L\}$:

$$\begin{aligned}\frac{\partial X(z,t)}{\partial t} &= -\left[\left(1-nX(z,t)\right)\, v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}.\end{aligned} \tag{4.6}$$

The boundary conditions specified for these PDEs are already given in Eq.(4.2).

This model and its boundary conditions can be transcribed into MSL-USER (version 4.0, used for PDE representation) almost literally:

```
foreach(z, range(-(0),-(1.5)),
 DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A )*DERIV(X, [z,])
                  + D_0*DERIV(X, [z,z])),
foreach(z, range(-(1.5),-(2.0)),
 DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A)*DERIV(X, [z,])
                    + X_f*Q_f/(2.0*sigma*A)
                   + D_0*DERIV(X, [z,z])),
foreach(z, range(-(2.0),+(5.0)),
 DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A)*DERIV(X, [z,])
                  + D_0*DERIV(X, [z,z])),
foreach(z, range(+(0),-(0)), DERIV(X, [z,]) = 0.0),
foreach(z, range(+(5),-(5)), DERIV(X, [z,]) = 0.0),
```

The syntax `-(x)` denotes an interval boundary "]" (open to the left), whereas `+(x)` denotes an interval boundary "[" (open to the right).

Effluent pumped

If there is pumping of the effluent out of the clarifier, the effluent flux is included explicitly in the PDE. Since the effluent velocity $Q_e(t)/A$ is unknown, we eliminate it using the relation $Q_e(t)/A = Q_f(t)/A - Q_u(t)/A$. The PDE is given by:

$$\begin{aligned}\frac{\partial X(z,t)}{\partial t} &= -\left[(1-nX(z,t))\ v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A}\,\theta(z_f - z)\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad - (X(z,t) - X_f(t))\,\frac{Q_f(t)}{A}\,\delta(z-z_f) \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}. \end{aligned} \tag{4.7}$$

We again represent the delta function by a unit pulse, as described earlier. The three PDEs for the implementation are given by:

1. for $\{0 \le z \le z_f - \sigma\}$:

$$\begin{aligned}\frac{\partial X(z,t)}{\partial t} &= -\left[(1-nX(z,t))\ v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A} - \frac{Q_f(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}; \end{aligned} \tag{4.8}$$

2. for $\{z_f - \sigma < z < z_f + \sigma\}$:

$$\begin{aligned}\frac{\partial X(z,t)}{\partial t} &= -\left[(1-nX(z,t))\ v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad - (X(z,t) - X_f(t))\,\frac{Q_f(t)}{A}\,\frac{1}{2\sigma} \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}; \end{aligned} \tag{4.9}$$

3. for $\{z_f + \sigma \le z \le L\}$:

$$\begin{aligned}\frac{\partial X(z,t)}{\partial t} &= -\left[(1-nX(z,t))\ v_0\, e^{-nX(z,t)} + \frac{Q_u(t)}{A}\right]\frac{\partial X(z,t)}{\partial z} \\ &\quad + D_0\,\frac{\partial^2 X(z,t)}{\partial z^2}. \end{aligned} \tag{4.10}$$

As before, the boundary conditions for these PDEs are given in Eq.(4.2).
In MSL-USER, this is represented as

```
foreach(z, range(-(0),-(1.5)),
 DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A - Q_f/A)*DERIV(X, [z,])
                  + D_0*DERIV(X, [z,z])),
foreach(z, range(-(1.5),-(2.0)),
 DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A)*DERIV(X, [z,])
                    + (X -X_f)*Q_f/(2.0*sigma*A)
                   + D_0*DERIV(X, [z,z])),
foreach(z, range(-(2.0),+(5.0)),
 DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A)*DERIV(X, [z,])
                  + D_0*DERIV(X, [z,z])),
foreach(z, range(+(0),-(0)), DERIV(X, [z,]) = 0.0),
foreach(z, range(+(5),-(5)), DERIV(X, [z,]) = 0.0),
```

with the boundary conditions as before. `theta` denotes the θ step-function (integral of a Dirac δ).

4.4.2 Discretization

As described before, the discretization for both cases above involving continuous sedimentation is done as follows: the domain of length L of the clarifier divided into three elements by four nodes located at $\{0, 1.5, 2.0, 5.0\}$. The unit pulse representing the delta funciton is assumed to span the second element, and has a width of $(2\sigma = 2\eta L = 0.1L)$. We choose one interior collocation point each on the first and second elements, and two on the third. We set $\alpha = \beta = 0$ on all three elements.
This discretization information is encoded in a file which drives the discretization process

```
3
0
1 0 0 1.5
1 0 0 2.0
2 0 0 5.0
```

This file's structure is

```
N_E
z_0
N(1) alpha(1) beta(1) z_1
N(2) alpha(2) beta(2) z_2
          .
          .
          .
N(N_E) alpha(N_E) beta(N_E) z_N_E
```

where

- `N_E` is the number of finite elements,
- `z_i` are the spatial locations (strictly monotonously increasing) of element boundaries,
- `z_0` and `z_N_E` are the domain boundaries,
- `N(i)` is the number of interior collocation points in finite element `i`,
- and `alpha(i)` and `beta(i)` are real numbers larger than -1 which determine the location of the collocations points in the finite element `i`.

We denote the concentrations at the four nodes by (u_1, u_2, u_3, u_4) and the concentrations at the four interior collocation points by (w_1, w_2, w_3, w_4). In the matrix formulation for the solution. we define vectors $U \equiv (u_1, u_2, u_3, u_4)$ and $W \equiv (w_1, w_2, w_3, w_4)$. Using the linear domain boundary conditions, and the linear element boundary conditions, we can relate U and W through a matrix equation:

$$PW = QU. \tag{4.11}$$

We can then substitute for the $\{u_i\}$ in the DAEs in terms of the $\{w_i\}$.

In our work, all concentrations are expressed in kilograms per cubic metre (kg/m^3), lengths in metres (m), and time in hours (h). In the graphs, these concentrations are relabelled for convenience so that $(u_1, w_1, u_2, w_2, u_3, w_3, w_4, u_4)$ correspond respectively to $(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8)$.

The length and area of the clarifier, the Vesilind parameters v_0, n, and the dispersion coefficient D_0 were taken from Watts et al. [WSK96]. The values are:

$$\begin{aligned} L &= 5\,m; \\ A &= 700\,m^2; \\ v_0 &= 7.65\,m/h; \\ n &= 0.306\,m^3/kg \\ D_0 &= 0.165\,m^2/h \\ Q_f &= 872\,m^3/h \\ X_f &= 4\,kg/m^3 \\ Q_u &= 397\,m^3/h \end{aligned} \tag{4.12}$$

Note that Q_f, X_f, Q_u are all constants for the present simulation. The discretization is carried out automatically by WEST++. The results of this discretization (MSL-USER 3.1 model classes to be included in the model base described earlier) are shown in Appendix C. Simulation results (in WEST++) demonstrating the validity of the approach are presented in the same Appendix.

4.4.3 The batch case

As described before the PDE for the concentration $X(z,t)$ in a 1-D clarifier undergoing batch sedimentation is given by:

$$\frac{\partial X(z,t)}{\partial t} = -\frac{\partial G(X(z,t))}{\partial z} + D_0 \frac{\partial^2 X(z,t)}{\partial z^2}. \tag{4.13}$$

D_0 is the dispersion coefficient or pseudo-diffusivity, $G(X(z,t))$ is the gravitational settling flux. As in the continuous case, we use the Vesilind settling velocity in $G(X(z,t))$.

The boundary conditions for the PDE above are given by:

$$\begin{aligned} G(X(0,t)) - D_0 \left.\frac{\partial X(z,t)}{\partial z}\right|_{z=0} &= 0 \\ G(X(L,t)) - D_0 \left.\frac{\partial X(z,t)}{\partial z}\right|_{z=L} &= 0. \end{aligned} \tag{4.14}$$

Discretization

For the batch case, the domain of interest $[0,5]$ is divided into two elements, by three nodes located at (0, 2, 5). We choose one interior collocation point on the first element, and two on the second, and set $\alpha = \beta = 0$ on both elements. We again define the matrices U, W, P, Q, as described for the continuous case. In the figures, we represent the concentrations $(u_1, w_1, u_2, w_2, w_3, u_4)$ by $(X_1, X_2, X_3, X_4, X_5, X_6)$.

The Vesilind parameters v_0, n, and the dispersion coefficient D_0 were taken from Watts et al. [WSK96] as before. The parameters for the batch case are thus:

$$\begin{aligned} L &= 5\,m; \\ v_0 &= 7.65\,m/h; \\ n &= 0.306\,m^3/kg \\ D_0 &= 0.165\,m^2/h. \end{aligned} \tag{4.15}$$

Three different concentration profiles similar to the continuous case were chosen for the initial values (w_1^0, w_2^0, w_3^0). These corresponded to the uniform profile $(9,9,9)kg/m^3$, the profile with postive upward concentration gradient $(11,8,6)kg/m^3$ and the profile with positive downward concentration gradient $(6,8,11)kg/m^3$.

Two groups of simulations were performed, depending on how the non-linear boundary conditions were dealt with. We describe the groups below:

- Group 1: This group corresponds to the simple linearization scheme described before, wherein the nonlinear flux $(G(X) = v_0 X e^{-nX})$ was simply replaced by the first term in its Taylor series in the boundary conditions, so that the matrix method could be used. That is, we set $(G(X) = v_0 X)$ at the boundaries.
- Group 2: A piece-wise linear approximation was made on $G(X)$, by inspecting a plot of $G(X)$ *vs.* X. (See Figure(4.3)). $G(X)$ is represented by three linear functions as follows:

$$G(X) = \begin{cases} v_0\,(m_1 X), & 0 \le X \le X_{max}\,; \\ v_0\,(m_2 X + c_2), & X_{max} < X \le X_0\,; \\ 0, & X > X_0\,. \end{cases} \tag{4.16}$$

  The slopes (m_1, m_2), the constant c_2 and (X_{max}, X_0), which are the values of X corresponding to the maximum of $G(X)$ and the $X-$ intercept of the straight line $y = v_0\,(m_2 X + c_2)$, are given below:

$$\begin{aligned} m_1 &= \frac{1}{e}; \\ m_2 &= \left(\frac{2}{e^2} - \frac{1}{e}\right); \\ c_2 &= \frac{2}{n}\left(\frac{1}{e} - \frac{1}{e^2}\right); \\ X_{max} &= \frac{1}{n}; \\ X_0 &= \frac{2}{n}\frac{(e-1)}{(e-2)}. \end{aligned} \tag{4.17}$$

Within each group above, two sets of simulations were performed:

- Set I: The Vesilind parameters and D_0 have the Watts values. Note that the batch settling results for Set I simulations can be compared to the limit case of Set V simulations of the continuous sedimentation results, where we have considered no influxes or outflows, but only settling with diffusion.
- Set II: The limit where there is only gravitational settling, that is, $D_0 = 0$.

Note that the other limit of pure diffusion would reduce to the same limit of the PDE for continuous sedimentation, and is not repeated here.

The MSL-USER representation of these models is similar to the continuous case, but the piecewise linear approximation of $G(X)/v_0$ requires the boundary conditions to be specified with an If/Then/Else construct.

The simulation results are presented in Appendix C.

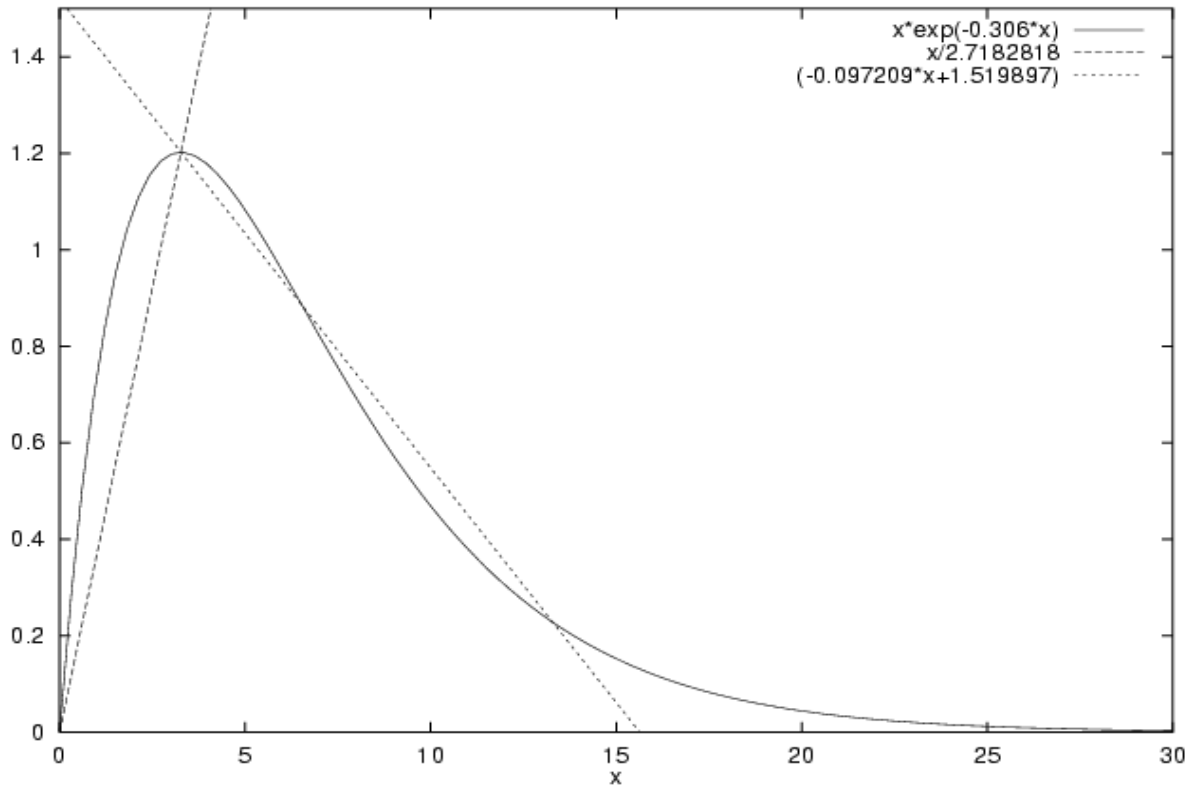

Figure 4.3: Piecewise linear approximation of $G(X)/v_0$

4.5 Modelling and simulation experiments

When an appropriate model base has been constructed for a given application domain, the WEST++ modelling environment allows for graphical component-based modelling. as depicted in Figure 4.4 (in this case of a simple WWTP). The user connects model icons in a hierarchical fashion. From this abstract specification, together with an MSL-USER library of dynamic models, one single MSL-USER model is produced. In particular, each icon put on the canvas results in the instantiation of an MSL-USER OBJect of the appropriate class. Connections between icons result in MSL-USER `connect` statements. This is shown in the code below generated from the above mentioned graphical representation.

```
// ************************************************************************
//
// University of Gent
// Department of Applied Mathematics, Biometrics and Process Control
//
// Project: WEST++
// Type: MSL
// Author: hv
//
// ************************************************************************


//
// Description: MSL-USER description of coupled model.
//              Macro-module: HGE
//              Date: Fri Dec  4 10:57:39 WET 1998
//

#include "wwtp.msl" // (* Mod: remove *)

#ifndef SIMPLE_WWTP_CLASS
#define SIMPLE_WWTP_CLASS

CLASS benchmarkClass
```

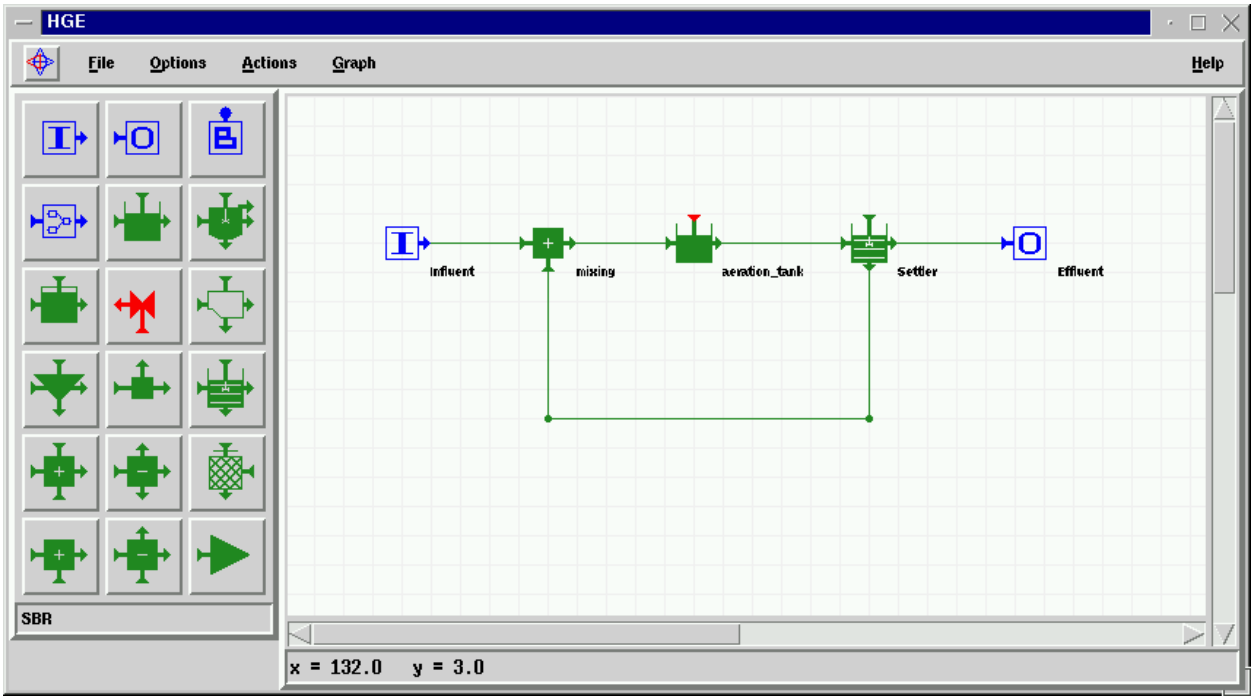

Figure 4.4: Simple WWTP Model

```
(* class = "coupled"; file = "/home/westpp/westpp/data/simpleWWTP/simpleWWTP.HGE.tcl"; *)
SPECIALISES CoupledModelType :=
{:
 interface <-
 {
  OBJ Influent (* terminal = "in_1" *) "Influent" : inWWTPTerminal := {:causality <- CIN:},
  OBJ Effluent (* terminal = "out_1" *) "Effluent" : outWWTPTerminal := {:causality <- COUT:},
 };
 parameters <-
 {
 };
 sub_models <-
 {
  OBJ aeration_tank : FixVolumeActivatedSludgeUnit,
  OBJ Settler : Takacs,
  OBJ mixing : TwoCombiner,
 };
 coupling <-
 {
  // parameter coupling
  // sub-model coupling
  connect(interface.Influent, sub_models.mixing.interface.Inflow),
  connect(sub_models.mixing.interface.Outflow, sub_models.aeration_tank.interface.Inflow),
  connect(sub_models.aeration_tank.interface.Outflow, sub_models.Settler.interface.Inflow),
  connect(sub_models.Settler.interface.Outflow, interface.Effluent),
 };
:};

#endif

OBJ simpleWWTP " ": simpleWWTPClass; // (* Mod: remove *)

// ***********************************************************************
```

Within the CoupledModel formalism, with sub-models of the DAE type, connections are expanded to the appropriate algebraic equalities. A compiler generates MSL-EXEC from this model for use within the experimentation environment.

It should be noted that a model base may contain *multiple* reasonable candidate models for a given part of a

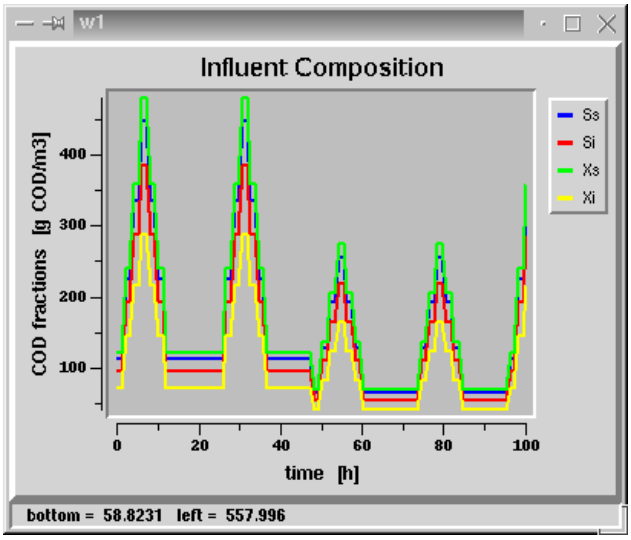

Figure 4.5: WWTP influent

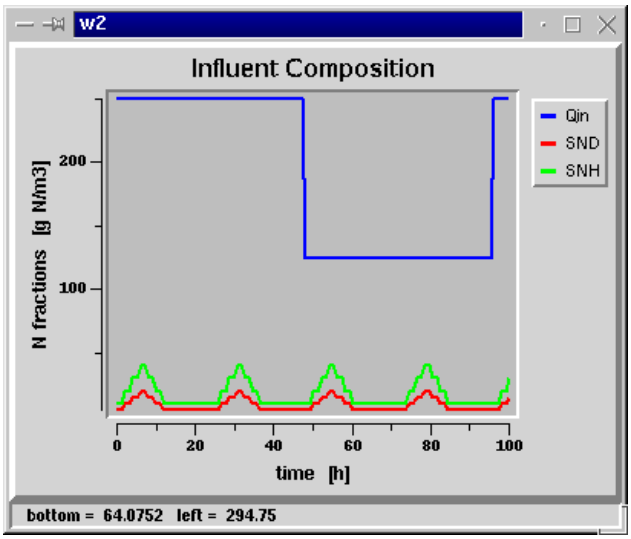

Figure 4.6: WWTP influent

WWTP. In WEST++, a model choosing routine retains feasible candidate models based on model features and user requirements and leaves the final choice to the user.

Without giving quantitative details, the following figures demonstrate how the benchmark model above can be used to realistically simulate WWTP behaviour. The influent in Figures 4.5 and 4.6 with flowrates (Figure 4.7 is broken down mainly in the bioreactor (Figure 4.8) which results is a purified effluent (Figure 4.9).

Once the parameter values have been *calibrated* to a particular waste water, a model may be used by the engineer to eliminate inefficient designs and to choose those alternative system configurations which are most likely to be economic. For a given system flowsheet, there is more than one choice of unit sizes which will result in a desired degree of treatment.

For this purpose, the WEST++ environment includes an *optimization* module which varies parameters (within bounds) so as to maximize a goal function. During optimization, sensitivity of the model to parameter variations (usually in the optimal parameter point) can be calculated. Currently this is done in a Monte Carlo fashion, though this could be achieved through symbolic model manipulation.

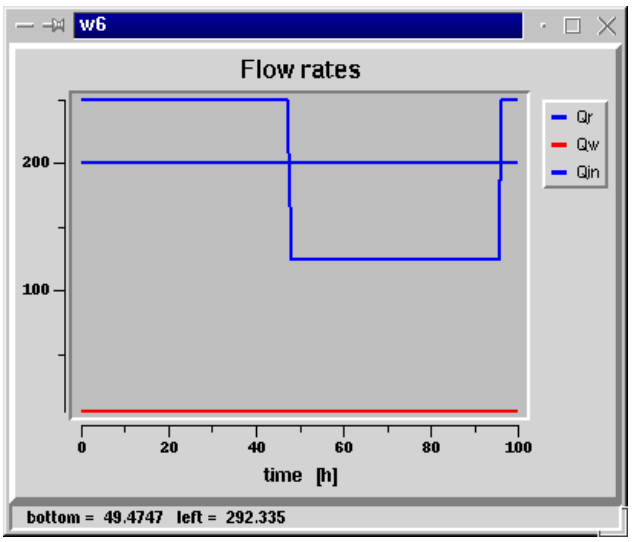

Figure 4.7: WWTP influent flowrates

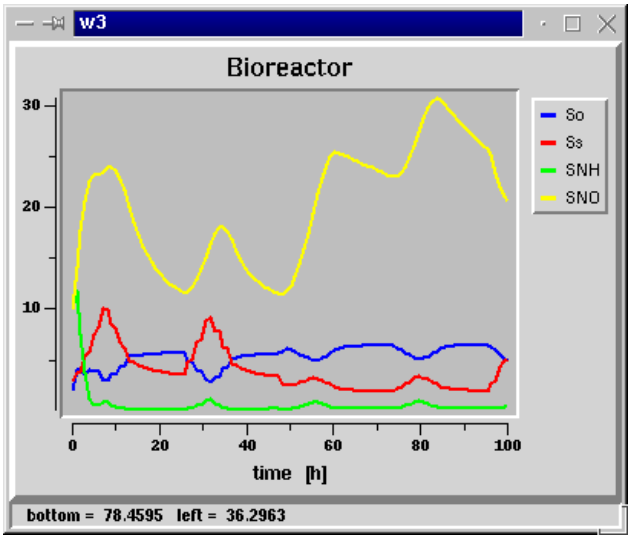

Figure 4.8: WWTP bioreactor

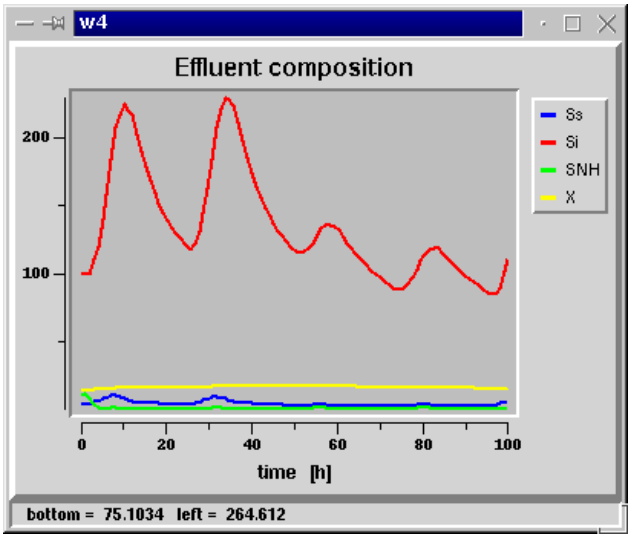

Figure 4.9: WWTP effluent

Summary

In this chapter, the concepts and techniques presented in previous chapters were applied in the activated sludge waste water treatment (WWTP) domain. After an introduction to activated sludge WWTP concepts, a physical systems modelling methodology was presented and applied to the WWTP domain. In particular, model knowledge is represented in MSL-USER, the modelling language used in the WEST++ modelling and simulation environment presented in the previous chapter. A core element in WWTP models is the Activated Sludge Model 1 (ASM1) of the International Association for Water Quality (IAWQ). This model, with its particular matrix format encoding stoichiometry and kinetics, is presented and subsequently represented in MSL-USER. The most detailed models of the clarifier part of a WWTP are described in the Partial Differential Equation (PDE) formalism. The automatic, symbolic discretization based on orthogonal collocation over finite elements presented previously is demonstrated for continuously fed as well as batch-fed models of the clarifier. Finally, the application of WEST++ to WWTP modelling, covering all phases from graphical modelling to numerical simulation, is briefly presented.

Conclusions

This thesis has developed a *multi-formalism* modelling and simulation methodology for complex systems. As such, its primary contribution is the *unification* of different modelling and simulation concepts, methods and techniques. To complement the theoretical concerns, computer implementation issues were dealt with and the study of activated sludge Waste Water Treatment Plants (WWTPs) by means of modelling and simulation was presented as a concrete application. A model (inter-formalism) *compiler* as well as the full-fledged *interactive modelling and simulation environment* WEST++ were developed. A central part of WEST++ is the new *declarative modelling language* MSL-USER.

In the thesis, some of the needs currently felt in the modelling and simulation community were addressed. These needs were identified by ESPRIT's Basic Research Working Group 8467 SiE-WG "Simulation in Europe", of which the author was a co-founder and co-ordinator. The work has generalized Zeigler's Theory of Modelling and Simulation by introducing *non-causal* modelling and *multi-formalism* modelling.

We now briefly review the thesis, pointing out original contributions.

In the first chapter, general modelling and simulation concepts such as verification and validation were introduced. In particular, a new, unified representation of modelling errors was introduced. The presentation brings structure to the multitude of different existing views and introduces original classifications of modelling formalisms. One important new dimension in classification is whether a modelling formalism is (computationally) causal. As not only the formalism a model is represented in is important, but also the process of manipulating those models, models of the modelling and simulation process were proposed. Such models may be used not only to describe the process, but also to prescribe the process and as such form the basis for automated modelling and simulation environments. The main contribution here is the Virtual Product Life-cycle (VPL) concept and a recursive process model for modelling and simulation supporting bottom-up as well as top-down construction and use of models.

In the second chapter, a rigourous presentation was given of diverse formalisms with a description of possible transformation between these. The transformations are at the basis of the implementation of model compilers. The transformations include

- transformation of event scheduling discrete event models to DEVS,
- transformation of Cellular Automata to DEVS,
- symbolic transformation of continuous formalisms based on graph algorithms. In particular, for causality assignment in non-causal models, Dinic's algorithm was introduced. This algorithm is more efficient than currently used algorithms,
- transformation between differential equations and transfer functions,
- transformation of Forrester's System Dynamics models to ordinary differential equations,
- transformation of a commonly used class of partial differential equations to ordinary differential equations based on orthogonal collocation over finite elements.

As may be apparent from the above transformations, the described formalisms are: I/O data trajectories, discrete event formalims in the form of four different "world views", the DEVS formalism, Cellular Automata, differential and algebraic equation formalisms (introducing non-causal models, how they enable model reuse and how to process them efficiently), the transfer function formalism, Forrester's System Dynamics formalism and a class of partial differential equations.

This chapter concluded by presenting an algorithm for "flattening" coupled multi-formalism models (mod-

ular networks). This was achieved by transforming the components to a common formalism. Knowledge about which transformation are feasible is kept in a Formalism Transformation Graph (FTG). The introduction of multi-formalism modelling, the FTG and the flattening algorithm forms a major contribution to the meaningful modelling of complex systems.

In the third chapter, the developments of the earlier chapters were used in the design of a declarative Model Specification Language (MSL-USER) as well as a full, interactive modelling and simulation environment (WEST++) to create, modify, and simulate MSL-USER models. The design requirements for MSL-USER were genericity, support for re-use and exchange, as well as for the representation of multiple formalisms. MSL-USER concepts are now used in the international modelling language standardization effort Modelica. MSL-USER is a first step towards meta modelling (the modelling of modelling formalisms), also described in this chapter.

In the fourth and last chapter, it was shown how in the domain of bio-activated sludge waste water treatment modelling, the presented concepts can be applied. In particular, a methodology for constructing model bases for physical systems was presented and illustrated for WWTPs. This lead to a generic model base for WWTP modelling. This model base has been the corner stone of all WWTP modelling at BIOMATH and its industrial partners. Finally, the application of WWTP models in the WEST++ tool was illustrated.

The work presented in this thesis should not be seen as an endpoint, but rather as a solid basis for further research and development. Current and future work will be focussed on:

- The further development of MSL-USER, merging it with Modelica. The unification of meta modelling with multi-formalism modelling.
- The application of multi-formalism modelling to software design (investigating how the Unified Modelling Language UML may be integrated).
- Detailed proofs for the mapping of discrete event world views onto DEVS. Formal verification of both multi-formalism transformations and model properties.
- The inclusion of more formalisms in the FTG. In particular, the relationship between Statecharts and DEVS needs to be investigated in detail.
- The mapping of Ordinary Differential Equations onto DEVS using quantization. Apart from performance considerations, this will also help bridge the gap between continuous and discrete (event) models (*i.e.*, hybrid modelling).
- An efficient, parallel implementation of DEVS on distributed memory “Beowulf” clusters.

Bibliography

[1292] ISO Technical Committee TC 12. *International standard ISO 1000: SI units and recommendations for the use of their multiples and of certain other units.* International Organization for Standardization, Casa Postale 56, CH-1211 Genève, Switzerland, 1992.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer-Verlag, 1996.

[ACS95] MGA Software, 200 Baker Avenue, Concord MA 01742, USA. *ACSL Reference Manual*, 1.1 edition, 1995.

[App97] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1997.

[AS65] M. Abramowitz and I.A. Stegun. *Handbook of Mathematical Functions.* Dover, New York, 1965.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[A.V98a] Indrani A.V. WEST++: 1-D clarifier models: Test cases. Technical Report 2, BIOMATH Department, Ghent, Belgium, 1998.

[A.V98b] Indrani A.V. WEST++: Transformation of a given PDE to a DAE using the orthogonal collocation method on finite elements. Technical Report 1, BIOMATH Department, Ghent, Belgium, 1998.

[AW95] George B. Arfken and Hans J. Weber. *Mathematical Methods for Physicists.* Academic Press, fourth edition, 1995.

[Baa88] Sarah Baase. *Computer Algorithms.* Addison-Wesley, 1988.

[Bal97a] Osman Balci. Principles of simulation model validation, verification, and testing. *Transactions of the Society for Computer Simulation International,* 14(1):3–12, March 1997. Special Issue: Principles of Simulation.

[Bal97b] Osman Balci. Verification, validation and accreditation of simulation models. In *Proceedings of the 1997 Winter Simulation Conference (Atlanta, GA)*. IEEE Computer Society Press, December 1997.

[Ban98] Jerry Banks, editor. *Handbook of Simulation: Principles, Methodology, Advances, Applications and Practice.* Wiley, 1998.

[Bar96] F.J. Barros. Dynamic structure discrete event system specification: Formalism, abstract simulators and applications. *Transactions of the Society for Computer Simulation International,* 13(1):35–46, 1996.

[Bar97] F.J. Barros. Modelling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation,* 7(4):501–515, 1997.

[Bar00] Paul Barton. Modeling, simulation and sensitivity analysis of hybrid systems. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 117–122. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[BB95] Don Bolinger and Tan Bronson. *RCS and SCCS.* O'Reilly & Associates, first edition, 1995.

[BCIN98] Jerry Banks, John S. Carson II, and Barry L. Nelson. *Discrete-Event System Simulation.* Prentice Hall of India, second edition, 1998.

[Bin95] David W. Binkley. C++ in safety critical systems. Technical Report NIST IR 5769, Technology Administration, National Institute of Standards and Technology (NIST), Computer Systems Laboratory, Gaithersburg, MD 20899, November 1995. `http://hissa.ncsl.nist.gov/sw_develop/safety.html`.

[BM93] Garrett Birkhoff and Saunders MacLane. *Algebra*. Chelsea Publishing Company, New York, N.Y., third edition, 1993.

[BO96] Louis G. Birta and F. Nur Özmizrak. A knowledge-based approach for the validation of simulation models: The foundation. *ACM Transactions on Modeling and Computer Simulation*, 6(1):76–98, January 1996.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[Boo98] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Object Technology Series. Addison-Wesley, second edition, 1998.

[Bos94] Hartmut Bossel. *Modeling and Simulation.* A.K. Peters, Ltd., 289 Linden Street, Wellesley, MA 02181, 1994.

[Bou91] Wiet Bouma. *Algebraïsche Specificaties.* Kluwer Programmatuurkunde, 1991.

[Bre84] Peter C. Breedveld. *Physical Systems Theory in Terms of Bond Graphs.* PhD dissertation, University of Twente, Enschede, Netherlands, 1984.

[Bro90] Jan F. Broenink. *Computer-Aided Physical-Systems Modeling and Simulation: a bond-graph approach*. PhD dissertation, University of Twente, Enschede, The Netherlands, 1990.

[BZF98] Fernando J. Barros, Bernard P. Zeigler, and Paul A. Fishwick. Multimodels and dynamic structure models: an integration of DSDE/DEVS and OOPM. In D.J. Medeiros, E.F. Watson, and Manivannan M.S., editors, *Proceedings of the 1998 Winter Simulation Conference*, pages 413–419. Society for Computer Simulation International (SCS), 1998.

[Car97] F. Cardarelli. *Scientific Unit Conversion.* Springer-Verlag, 1997.

[Cas93] Christos G. Cassandras. *Discrete Event Systems.* Irwin, 1993.

[CAw99] CAweb. Artificial life. Cellular automata website, The Santa Fe Institute, 1999. `http://alife.santafe.edu/alife/topics/ca/caweb/`.

[Cel91] François E. Cellier. *Continuous System Modeling.* Springer-Verlag, New York, 1991.

[CH97] Patrick Coquillard and David R.C. Hill. *Modélisation et Simulation d'Écosystèmes; Des modèles déterministes aux simulations à événements discrets.* Recherche en Écologie. Masson, 120, Bd. Saint-Germain, 75280 Paris Cedex 06, 1997.

[Cho96] A.C.-H. Chow. Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *Transactions of the Society for Computer Simulation International*, 13(2):55–68, June 1996.

[CK00] Alongkrit Chutinan and Bruce H. Krogh. Computing approximating automata for a class of hybrid systems. *Mathematical and Computer Modelling of Dynamical Systems (MCMDS)*, 6(1):30–50, March 2000.

[CL89] John Carroll and Darrell Long. *Theory of Finite Automata*. Prentice Hall, 1989.

[Cla92] Filip Claeys. *HGPSS: Object-georiënteerde "process-interaction" Simulatie*. Master's thesis, University of Ghent (RUG), Department of Applied Mathematics, Biometrics and Process Control (BIOMATH), Coupure links 653, B-9000 Gent, Belgium, June 1992.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer Verlag, third edition, 1987.

[Com94] EIA/CDIF Technical Committee. CDIF CASE data interchange format – overview, January 1994. EIA Interim Standard EIA/IS-106.

[CPV+98] F. Coen, B. Petersen, P.A. Vanrolleghem, B. Vanderhaegen, and M. Henze. Model-based characterisation of hydraulic, kinetic and influent properties of an industrial wwtp. *Wat. Sci. Tech.*, 37(12):317–326, 1998.

[CS92] Bruce A. Cota and Robert G. Sargent. A modification of the process interaction world view. *ACM Transactions on Modeling and Computer Simulation*, 2(2):109–129, April 1992.

[DA92] René David and Hassane Alla. *Petri Nets & Grafcet*. Prentice Hall Inc., Englewood Cliffs, NJ, 1992. ISBN 0-13-327537-X.

[DGG+99] John Davis, II, Ron Galicia, Mudit Goel, Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, and Yuhong Xiong. Ptolemy II – heterogeneous concurrent modeling and design in java. http://ptolemy.eecs.berkeley.edu, Department of Electrical Engineering and Computer Sciences, University of Califoria at Berkeley, 1999. version 0.1.1.

[Din70] E.A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[dKB84] Johan de Kleer and John Seely Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[DOD95] Military standard defense system software development. WWW Report, 1995. `http://www2.umassd.edu/SWPI/DOD/DOD2167A.html`.

[Dou99] Bruce Powel Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.

[DST93] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra, Systems and Algorithms for Algebraic Computation*. Academic Press, June 1993.

[EBB+99] Hilding Elmqvist, Bernhard Bachmann, Fabrice Boudard, Jan Broenink, Dag Brück, Thilo Ernst, Rüdiger Franke, Peter Fritzson, Alexandre Jeandel, Pavel Grozman, Kaj Juslin, David Kågedal, Matthias Klose, Nathalie Loubère, Sven-Erik Mattson, Pieter Mosterman, Henrik Nilsson, Martin Otter, Per Sahlin, André Schneider, Hubertus Tummescheit, and Hans Vangheluwe. Modelica – a unified object-oriented language for physical systems modeling: Tutorial and rationale. Report, The Modelica Design Group, December 1999. `http://www.modelica.org/`.

[Eck98] J. Dana Eckart. A *cellular* automata simulation system. Technical report, Radford University, Radford, VA 24242, August 1998. `http://www.cs.runet.edu/~dana/ca/tutorial.ps`.

[EK72] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1972.

[EK00] E. Engstrom and J. Kruger. A meta-modeler's job is never done: Building and evolving domain-specific tools with DOME. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 83–88. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[Elm78] Hilding Elmquist. *A Structured Model Language for Large Continuous Systems*. Doctoral dissertation, Lund Institute of Technology, Sweden, 1978.

[Ern96] Johannes Ernst. Data interoperability between CACSD and CASE tools using the CDIF family of standards. In *Proceedings of the 1996 International Symposium on Computer-Aided Control System Design*, pages 346–351, Dearborn, MI, September 1996.

[ET75] S. Even and Robert Endre Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[EW00] J. Ernst and S. Washburn. Zero-latency engineering for control design. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 71–76. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[F+96] B. Fucssteiner et al. *MuPAD User's Manual; Multi-Processing Algebra Data Tool*. Wiley, 1996.

[Far93] Stanley J. Farlow. *Partial Differential Equations for Scientists and Engineers*. Dover Publications, New York, 1993.

[FF62] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *ACM SIGPLAN Notices*, 34(5):192 – 203, May 1999. Proceedings of the 1999 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI).

[FFN91] William B. Frakes, Christopher J. Fox, and Brian A. Nejmeh. *Software Engineering in the UNIX/C Environment*. Prentice-Hall, 1991.

[FH93] Norman Fenton and Gillian Hill. *Systems Construction and Analysis: A Mathematical and Logical Framework*. International Series in Software Engineering. McGraw-Hill, Maidenhead, England, 1993.

[Fis99] Michael Fisher. Zero-latency engineering^{tm}. Aviatis Corp., White Paper, 1999.

[FL91] Paul A. Fishwick and Paul A. Luker, editors. *Qualitative Modelling and Simulation*, volume 5 of *Advances in Simulation Series*. Springer-Verlag, 1991.

[For61] Jay W. Forrester. *Industrial Dynamics*. System Dynamics Series. Productivity, paperback edition, 1961.

[For68] Jay W. Forrester. *Principles of Systems*. Pegasus Communications, 1968.

[For84] Kenneth D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[For90] Kenneth D. Forbus. The qualitative process engine. *Readings in Qualitative Reasoning about Physical Systems*, pages 220–235, 1990.

[Fow97] Martin Fowler. *UML Distilled, Applying the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, 1997.

[FY97] Lester Foster and Kevin Yelmgren. Accuracy in DoD High Level Architecture Federations. In Mohamed S. Obaidat and John Illgen, editors, *Summer Computer Simulation Conference (SCSC'97)*, pages 451–460. Society for Computer Simulation International (SCS), July 1997. Arlington, Virginia.

[Gar70] Martin Gardner. The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, October 1970.

[GCL92] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, October 1992.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[GMS93] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, 1993.

[Gor96] Geoffrey Gordon. *System Simulation*. Prentice Hall of India, second edition, 1996.

[Gra00] John E. Grayson. *Python and Tkinter Programming*. Manning Publications, 2000.

[Gro99a] IEEE 1076.1 Working Group. IEEE standard 1076.1-1999, March 1999. http://www.vhdl.org.

[Gro99b] The Object Management Group. OMG unified modeling language specification, June 1999. version 1.3, http://www.omg.org/.

[GSMC85] Laura Gardini, Alberto Servida, Massimo Morbidelli, and Sergio Carra. Use of orthogonal collocation on finite elements with moving boundaries for fixed bed catalytic reactor simulation. *Computers and Chemical Engineering*, 9(1):1–17, 1985.

[Gua87] Antoni Guasch. *MUSS: A Contribution to the Structural Analysis of Continuous System Simulation Languages*. PhD dissertation, Universitat Politècnica de Catalunya, Barcelona, December 1987.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[HB97] Myron Hecht and Robert Brill. Review guidelines for software written in high level programming language used in safety systems. In 25^{th} *Water Reactor Safety Meeting*, pages 1 – 8, Bethesda, MD, October 1997. `http://intranet.sohar.com/Webdocs/PubSearch/dlpub_mdb.cfm?sohar_no=9701%2`.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, pages 31–42, 1997.

[HGG+86] M. Henze, C.P.L. Grady, W. Gujer, G.v.R. Marais, and T. Matsuo. Activated sludge model no. 1. Scientific and Technical Report 1, International Association on Water Pollution Research and Control (IAWPRC, now IAWQ), London, England, July 1986. IAWPRC Task Group on Mathematical Modelling for Design and Operation of Biological Wastewater Treatment.

[HGM+94] M. Henze, W. Gujer, T. Mino, T. Matsuo, M.C. Wentzel, and G.v.R. Marais. Activated sludge model no. 2. Scientific and Technical Report 3, International Association on Water Quality (IAWQ), London, England, 1994.

[Hig94] Nicholas J. Higham. *Handbook of Writing for the Mathematical Sciences.* Society for Industrial and Applied Mathematics (SIAM), 3600 University City Science Center, Philadelphia, PA 19104-2688, 1994.

[Hil96] David R.C. Hill. *Object-Oriented Analysis and Simulation.* Addison-Wesley, 1996.

[HK89] Watts S. Humphrey and Marc I. Kellner. Software process modeling: Principles of entity process models. In *Proceedings of the 11th International Conference on Software Engineering*, pages 331–342, Pittsburg, PA, May 15-18 1989. ACM.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[Hon99] Honeywell. DOME guide. http://www.htc.honeywell.com/dome/, Honeywell Technology Center, Honeywell, 1999. version 5.2.1.

[HP88] Derek J. Hatley and Imtiaz Pirbhai. *Strategies for Real-Time Systems Specification.* Dorset House Publishing Co., New York, New York, 1988.

[Hum89] Watts S. Humphrey. *Managing the Software Process.* SEI Series in Software Engineering. Addison-Wesley, October 1989.

[Jam00] J. James. Thoughts on information operation detection as a nonlinear, mixed-signal identification problem: A control systems view. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 77–82. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[JLZS00] K. Johansson, J. Lygeros, J. Zhang, and S. Sastry. Hybrid automata: A formal paradigm for heterogeneous modeling. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 123–128. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[JM98] J. Jungblut and D.P.F. Möller. Modelling, simulation and optimization of biological wastewater treatment plants. In 12^{th} *European Simulation Multi-conference (ESM)*, pages 377–379. Society for Computer Simulation International (SCS), 1998. Manchester, UK.

[Kli85] George J. Klir. *Architecture of Systems Problem Solving.* Plenum Press, 1985.

[KMR90] D.C. Karnopp, D.L. Margolis, and R.C. Rosenberg. *Systems Dynamics: A Unified Approach.* John Wiley and Sons, New York, 2 edition, 1990.

[KNLS00] G. Karsai, G. Nordstrom, A. Ledeczi, and J. Sztipanovits. Specifying graphical modeling systems using constraint-based metamodels. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 89–94. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[Kop97] Sebastiaan Kops. WEST++: Generating DAE's from PDE's. Technical Report 1, BIOMATH Department, Ghent, Belgium, 1997.

[Kre86] Wolfgang Kreutzer. *System simulation: programming styles and languages*. Addison-Wesley, 1986.

[KSKP96] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the Society for Computer Simulation International*, 13(3):135–154, September 1996.

[Kui86] Benjamin Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:289–338, 1986.

[Kui88] Benjamin Kuipers. Qualitative simulation using time-scale abstraction. *International Journal of Artificial Intelligence in Engineering*, 3:185–191, 1988.

[Kui94] Benjamin Kuipers. *Qualitative Reasoning: Modeling and Simulating with Incomplete Knowledge*. MIT Press, Cambridge, MA, 1994.

[KV96] Eugène J.H. Kerckhoffs and Hans L. Vangheluwe. A multi-agent architecture for sharing knowledge and experimental data about waste water treatment plants through the internet. In Alexander Verbraeck, editor, *EUROMEDIA 96*, pages 12–18. Society for Computer Simulation International (SCS), December 1996.

[KVC+97] Sebastiaan Kops, Hans L. Vangheluwe, Filip Claeys, Filip Coen, Peter A. Vanrolleghem, Zhiguo Yuan, and Ghislain C. Vansteenkiste. The process of model building and simulation of ill-defined systems: Application to waste water treatment. In *Proceedings IMACS 2nd MATHMOD Conference*, February 1997. Vienna, Austria.

[KVC+99] Sebastiaan Kops, Hans L. Vangheluwe, Filip Claeys, Peter A. Vanrolleghem, Zhiguo Yuan, and Ghislain C. Vansteenkiste. The process of model building and simulation of ill-defined systems: Application to wastewater treatment. *Mathematical and Computer Modelling of Dynamical Systems (MCMDS)*, 5(4):298–312, 1999.

[KVVG94] Eugène J.H. Kerckhoffs, Hans L. Vangheluwe, Ghislain C. Vansteenkiste, and Philippe Geril. Improving the modelling and simulation process. Progress Report 1994:1, ESPRIT Basic Research Working Group 8467, University of Ghent (RUG), Department of Applied Mathematics, Biometrics and Process Control (BIOMATH),Coupure links 653, B-9000 Gent, Belgium, 1994.

[KVVG95] Eugène J.H. Kerckhoffs, Hans L. Vangheluwe, Ghislain C. Vansteenkiste, and Philippe Geril. Improving the modelling and simulation process. Progress Report 1995:1, ESPRIT Basic Research Working Group 8467, University of Ghent (RUG), Department of Applied Mathematics, Biometrics and Process Control (BIOMATH),Coupure links 653, B-9000 Gent, Belgium, 1995.

[LA90] Shem-Tov Levi and Ashok K. Agrawala. *Real-Time System Design.* McGraw-Hill, 1990.

[Lam86] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1986.

[Law76] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.

[Lee99] Edward A. Lee. Embedded software – an agenda for research. Technical Report M99/63, Department of EECS, University of California, Berkeley, CA 94720, 1999.

[Lee00] Edward A. Lee. What's ahead for embedded software. *IEEE Computer*, pages 18 – 26, 2000.

[LK91] Averill M. Law and David W. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, 1991.

[LL00] J. Liu and E. Lee. Component-based hierarchical modeling of systems with continuous and discrete dynamics. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 95–100. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, second edition, October 1992.

[LSV98] Pedro Garcia Lopez, Antonio F. Skarmeta, and Hans L. Vangheluwe. A push-based system for web-based simulation in an educational environment. In André Bargiela and Eugène Kerckhoffs, editors, *10th European Simulation Symposium*, pages 699–703. Society for Computer Simulation International (SCS), October 1998. Nottingham, UK.

[Lut96] Mark Lutz. *Programming Python*. O'Reilly & Associates, 1996.

[LW95] D.A. Linkens and H. Wang. Qualitative bond graph reasoning in control engineering - part i: Qualitative bond graph modeling and controller design. In François E. Cellier and José J. Granda, editors, *1995 International Conference on Bond Graph Modeling and Simulation (ICBGM '95)*, number 1 in Simulation, pages 183–188, Las Vegas, January 1995. Society for Computer Simulation, Simulation Councils, Inc. Volume 27.

[Mae94] P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, July 1994.

[Mag85] Brian Magee. *Popper*. Fontana Press (An Imprint of HarperCollins Publishers), London, 1985.

[MB01] Pieter J. Mosterman and Gautam Biswas. A modeling and simulation methodology for hybrid dynamic physical systems. *Transactions of the Society for Computer Simulation International*, 18, 2001. (to appear).

[MHS+00] J. Meirlaen, B. Huyghebaert, F. Sforzi, L. Benedetti, and P.A. Vanrolleghem. Fast, parallel simulation of the integrated urban wastewater system using mechanistic surrogate models. *Wat. Sci. Tech. (in Press)*, 2000.

[Mil93] Robin Milner. Elements of interaction. *Communications of the ACM*, 36(1):70–89, January 1993. Turing Award Lecture.

[ML93] S. Marsili-Libelli. Dynamic modelling of sedimentation in the activated sludge process. *Civil. Eng. Syst.*, 10, 1993.

[MML99] Dieter Moormann, Pieter J. Mosterman, and Gert-Jan Looye. Object-oriented computational model building of aircraft flight dynamics and systems. *Aerospace Science and Technology*, (3):115–126, 1999.

[MOE98] Pieter J. Mosterman, Martin Otter, and Hilding Elmqvist. Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica. In *SCS Summer Simulation Conference*, pages 314–319, Reno, Nevada, July 1998.

[Mon97] Y. Monsef. *Modelling and Simulation of Complex Systems; Concepts, Methods and Tools*. Frontiers in Simulation Series. Society for Computer Simulation International (SCS), 1997.

[Mor94] Carol Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, second edition, 1994.

[MS92] John D. McGregor and David A. Sykes. *Object-Oriented Software Development: Engineering Software for Reuse*. VNR Computer Library. Van Nostrand Reinhold, New York, 1992.

[MS00] Olaf Müller and Thomas Stauner. Modelling and verification using linear hybrid automata – a case study. *Mathematical and Computer Modelling of Dynamical Systems (MCMDS)*, 6(1):71–89, March 2000.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[MV72] M.L. Michelsen and J. Villadsen. A convenient computational procedure for collocation constants. *The Chemical Engineering Journal*, 4:64–68, 1972.

[MV00] Pieter Mosterman and Hans L. Vangheluwe. Computer automated multi paradigm modeling in control system design. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 65–70. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[Nan81] Richard E. Nance. The time and state relationships in simulation modeling. *Communications of the ACM*, 24(4):173–179, April 1981.

[Nee87] Francis Neelamklavil. *Computer Simulation and Modelling.* Wiley, 1987.

[Nor99] Gregory G. Nordstrom. *Metamodeling – Rapid Design and Evoluion of Domain-Specific Modeling Environments.* PhD dissertation, Vanderbilt University, Electrical Engineering, May 1999.

[NT00] Simin Nadjm-Tehrani. Formal methods for analysis of heterogeneous models of embedded systems. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 141–146. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[Nut99] James Nutaro. aDEVS-0.2. C++ library for parallel DEVS, University of Arizona, Tucson, 1999. `www.ece.arizona.edu/~nutaro/`.

[Ode98] James Odell. Agents and beyond: A flock is not a bird. *Distributed Computing*, pages 5–7, April 1998. `www.distributedComputing.com`.

[Oh95] M. Oh. *Modelling and Simulation of Combined Lumped and Distributed Processes.* PhD dissertation, University of London, London, March 1995.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, 1996.

[OM99] Martin Otter and Pieter Mosterman. The DSblock model interface, version 4.0. Technical report, Modelica Design, 1999. `www.Modelica.org/DSblock/dsblock4.0a.shtml`.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Par00] Taeshin Park. Implicit model checking: Formal verification technique for large-scale discrete systems. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 135–140. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[Pau93] Frances Newbery Paulisch. *The Design of an Extendible Graph Editor*. Lecture Notes in Computer Science. Springer-Verlag, 1993.

[PB96] Taeshin Park and Paul I. Barton. State event location in differential-algebraic models. *ACM Transactions on Modeling and Computer Simulation*, 6(2):137–165, April 1996.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

[Rai00] Jörg Raisch. Discrete abstractions of continuous systems – an input/output point of view. *Mathematical and Computer Modelling of Dynamical Systems (MCMDS)*, 6(1):6–29, March 2000.

[RBH+00] P. Reichert, D. Borchardt, M. Henze, W. Rauch, P. Shanahan, L. Somlydy, and Vanrolleghem P.A. River water quality model no. 1 (rwqm1): II. biochemical process equations. *Wat. Sci. Tech. (in Press)*, 2000.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, 1999.

[ROE99] Rudy Rucker, Richard Ottolini, and J. Dana Eckart. Artificial life frequently asked questions. Alife website, Santa Fe Institute, 1999. `http://alife.santafe.edu/alife/topics/cas/ca-faq/soft/soft.html`.

[RS94] Ashvin Radiya and Robert G. Sargent. A logic-based foundation of discrete event modeling and simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):3–51, 1994.

[RVM+00] D. Rousseau, F. Verdonck, O. Moerman, R. Carrette, C. Thoeye, J. Meirlaen, and P.A. Vanrolleghem. Development of a risk assessment based technique for design/retrofitting of wwtps. *Wat. Sci. Tech. (in Press)*, 2000.

[RVV99] W. Rauch, H. Vanhooren, and P.A. Vanrolleghem. A simplified mixed-culture biofilm model. *Wat. Res.*, 33, 1999.

[SAF+67] Jon C. Strauss, Donald C. Augustin, Mark S. Fineberg, Bruce B. Johnson, Robert N. Linebarger, and F. John Sansom. The SCi Continuous System Simulation Language (CSSL). *Simulation*, 9(6):281 – 303, December 1967.

[Sah96] Per Sahlin. *Modelling and Simulation Methods for Modular Continuous Systems in Buildings.* Doctoral dissertation, Royal Institute of Technology, Sweden, 1996.

[SB95] J.E.E. Sharpe and R.H. Bracewell. The use of bond graph reasoning for the design of interdisciplinary schemes. In François E. Cellier and José J. Granda, editors, *1995 International Conference on Bond Graph Modeling and Simulation (ICBGM '95)*, number 1 in Simulation, pages 116–121, Las Vegas, January 1995. Society for Computer Simulation, Simulation Councils, Inc. Volume 27.

[SBH+00] P. Shanahan, D. Borchardt, M. Henze, W. Rauch, P. Reichert, L. Somlydy, and Vanrolleghem P.A. River water quality model no. 1 (rwqm1): I. modelling approach. *Wat Sci Tech (in Press)*, 2000.

[SBS94] Per Sahlin, P. Bring, and A. Sowell. The neutral model format for building simulation. ITM Report 1994:2, Royal Institute of Technology, Stockholm, 1994.

[Sch74] Thomas J. Schriber. *Simulation Using GPSS.* Wiley, 1974.

[Sed92] Robert Sedgewick. *Algorithms in C++.* Addison-Wesley, 1992.

[SIM97] SIMULINK. *Dynamic System Simulation for Matlab.* The MathWorks, January 1997.

[SJ85] R.K. Srivastava and B. Joseph. Reduced order models for separation columns-v. selection of collocation points. *Computers and Chemical Engineering*, 9(6):601–613, 1985.

[SKB+95] J. Sztipanovits, G. Karsai, C. Biegl, T. Bapty, A. Ledeczi, and A. Misra. MULTIGRAPH: An architecture for model-integrated computing. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, pages 361–368, Ft. Lauderdale, Florida, November 1995.

[SKE00] Olaf Stursberg, Stefan Kowalewski, and Sebastian Engell. On the generation of timed discrete approximations of continuous systems. *Mathematical and Computer Modelling of Dynamical Systems (MCMDS)*, 6(1):51–70, March 2000.

[Sme88] Rein Smedinga. *Simulatie en Implementatie.* Addison-Wesley, Amsterdam, 1988.

[SS98] Tan Kiat Shi and Willi-Hans Steeb. *Symbolic C++, An Introduction to Computer Algebra Using Object-Oriented Programming*. Springer-Verlag, 1998.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[SV82] Jan Spriet and Ghislain C. Vansteenkiste. *Computer Aided Modelling and Simulation.* International Lecture Series in Computer Science. Academic Press, 1982.

[Tak96] Shingo Takahashi. General morphism for modeling relations in multimodeling. *Transactions of the Society for Computer Simulation International*, 13(4):169–178, December 1996.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial & Applied Mathematics, 1983.

[Tay82] J.-H. Tay. Development of a settling model for primary settling tanks. *Wat. Res.*, 16, 1982.

[TBBA95] J.L. Top, A.P.J. Breunese, J.F. Broenink, and J.M. Akkermans. Structure and use of a library for physical systems models. In François E. Cellier and José J. Granda, editors, *1995 International Conference on Bond Graph Modeling and Simulation (ICBGM '95)*, number 1 in Simulation, pages 97–102, Las Vegas, January 1995. Society for Computer Simulation, Simulation Councils, Inc. Volume 27.

[Van86] Hans L. Vangheluwe. The non-relativistic, quantummechanical solution of the Helium problem (in Dutch). B.sc. dissertation, Department of Theoretical Physics, Faculty of Science, University of Ghent (RUG), June 1986.

[Van88] Hans L. Vangheluwe. Study and implementation of an experimental Prolog interpreter (in Dutch). M.Sc. dissertation, Department of Informatics, Faculty of Applied Science, University of Ghent (RUG), September 1988.

[Van89a] Hans Vangheluwe. The beauty of ordered complexity. *Simulation News Europe*, 4:23–25, Spring 1989. Feature Article.

[Van89b] Hans L. Vangheluwe. Toepassingen van digitale simulatie. In *KIVI symposium*, Antwerp, April 1989. Dutch Benelux Simulation Society (DBSS).

[Van99] Hans L. Vangheluwe. Model semantics for software agents. Dagstuhl seminar no. 99271, report no. 245, Schloss Dagstuhl, July 1999.

[Van00] Hans L. Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134. IEEE Computer Society Press, September 2000. Anchorage, Alaska.

[VBH+00] P.A. Vanrolleghem, D. Borchardt, M. Henze, W. Rauch, P. Reichert, P. Shanahan, and L. Somlyódy. River water quality model no. 1: III. biochemical submodel selection. *Wat. Sci. Tech. (in Press)*, 2000.

[VBV91] Hans L. Vangheluwe, Jorge Barreto, and Ghislain C. Vansteenkiste. *Application of a Multi-faceted Modelling Methodology : an Example in Physiology*, volume 10-11 of *Mathematical and Intelligent Models in System Simulation*, page 6. J.C. Baltzer Scientific Publishing Company, 1991. Based on a presentation at the IMACS International Symposium on Mathematical and Intelligent Models in System Simulation, Brussels, September 3-7, 1990.

[VCK+96] Hans L. Vangheluwe, Filip Claeys, Sebastiaan Kops, Filip Coen, and Ghislain C. Vansteenkiste. A modelling and simulation environment for wastewater treatment plant design. In *European Simulation Symposium (ESS)*, pages 90–97. Society for Computer Simulation International (SCS), October 1996. Genoa, Italy.

[VCV98] Hans L. Vangheluwe, Filip Claeys, and Ghislain C. Vansteenkiste. The WEST++ wastewater treatment plant modelling and simulation environment. In André Bargiela and Eugène Kerckhoffs, editors, *10th European Simulation Symposium, Nottingham*, pages 756–761. Society for Computer Simulation International (SCS), October 1998. Nottingham, UK.

[vdCKV96] Wijnand van de Calseijde, Eugène J.H. Kerckhoffs, and Hans L. Vangheluwe. Sharing of waste water treatment plant experimental data through the internet: An initial study. In *European Simulation Symposium (ESS)*, pages 12–18. Society for Computer Simulation International (SCS), October 1996. Genoa, Italy.

[VDCV+99] A. Vanderhasselt, B. De Clercq, B. Vanderhaegen, W. Verstraete, and P.A. Vanrolleghem. On-line control of polymer addition to prevent massive sludge wash-out. *J. Environ. Eng.*, 125, 1999.

[VDVV00] H. Vanhooren, D. Demey, I. Vannijvel, and P.A. Vanrolleghem. Monitoring and modelling an industrial trickling filter using on-line off-gas analysis and respirometry. *Wat. Sci. Tech.*, 41(12):139–148, 2000.

[Ve90] Hans L. Vangheluwe (editor). DESiRE: Dynamic Expert Systems in Robotic Experimentation. Project report, University of Ghent (RUG), Université Catholique de Louvain (UCL), Delft University of Technology (TUD), 1990.

[VKV96] Hans L. Vangheluwe, Eugène J.H. Kerckhoffs, and Ghislain C. Vansteenkiste. Simulation for the Future: Progress of the ESPRIT Basic Research working group 8467. In Agostino Bruzzone and Eugène J.H. Kerckhoffs, editors, *European Simulation Symposium (ESS)*, pages XXIX – XXXIV. Society for Computer Simulation International (SCS), October 1996. Genoa, Italy.

[VKV01] Hans L. Vangheluwe, Eugène J.H. Kerckhoffs, and Ghislain C. Vansteenkiste, editors. *Modelling and Simulation Technologies White Book*. Society for Computer Simulation International (SCS), 2001. (To appear).

[VKVWV92] Ghislain C. Vansteenkiste, Eugène J.H. Kerckhoffs, Danny Van Welden, and Hans L. Vangheluwe. DESiRE: Dynamic Expert Systems in Real-Time Environments. In *Proceedings of the 2nd Beijing International Conference on System Simulation and Scientific Computing*, pages 14–19. Chinese Association for System Simulation, 1992.

[VLRV93] Hans L. Vangheluwe, Bo Hu Li, Y.V. Reddy, and Ghislain C. Vansteenkiste. A framework for concurrent simulation engineering. In Y.V. Reddy and Sa'ad Medhat, editors, *Simulation in Concurrent Engineering*, pages 50–55. Society for Computer Simulation International (SCS),

May 1993. Proceedings of the International Training Equipment Conference (ITEC), Den Haag, The Netherlands.

[vN66] John von Neumann. *Theory of Self-reproducing Automata.* University of Illinois Press, 1966. Edited and completed by Arthur W. Burks.

[VSB79] Ghislain C. Vansteenkiste, Jan Spriet, and J. Bens. Structure characterization for system modeling in uncertain environments. In B.P. Zeigler, M.S. Elzas, G.J. Klir, and T.I. Oren, editors, *Methodology in Systems Modelling and Simulation.* North-Holland, 1979.

[VV89] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. An automatic object-oriented continuous simulation environment. In Ghislain C. Vansteenkiste, Len Dekker, and Eugène J.H. Kerckhoffs, editors, *Parallel Information Processing in Simulation,* CREST, Subcommittee "Informatics and Information Technology", pages G186–G198. Society for Computer Simulation International (SCS), 1989.

[VV90] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. Development of an automatic object-oriented continuous simulation environment. In Bernard Zeigler and Jerzy Rozenblit, editors, *Proceedings of the AI, Simulation and Planning in High Autonomy Systems Conference,* page addendum. University of Arizona, Tucson, Arizona, IEEE Computer Society Press, March 1990. Tucson, Arizona.

[VV91] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. Development of an automatic object-oriented continuous simulation environment. *International Journal of General Systems,* 19(3):263–278, 1991. G. Klir, editor.

[VV95] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. Computer-aided modelling of complex systems. In *Proceedings of the ADIUS Sixteenth Annual Conference (Seattle),* 3800 Stone School Rd., Ann Arbor, Michigan 48108-2499, USA, June 1995. Applied Dynamics International.

[VV96a] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. A multi-paradigm modeling and simulation methodology: Formalisms and languages. In *European Simulation Symposium (ESS),* pages 168–172. Society for Computer Simulation International (SCS), October 1996. Genoa, Italy.

[VV96b] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. A rigorous approach to modelling and simulation of waste-water treatment plants. In Bernard Zeigler, editor, *Proceedings of the AI, Simulation and Planning in High Autonomy Systems Conference,* pages 39–46 (addendum), Harvill Building Rm 135 Box 9, University of Arizona, PO Box 210076, Tucson AX 85721-0076, USA, March 1996. University of Arizona, Engineering Professional Development.

[VV96c] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. SiE: Simulation for the Future. *Simulation,* 66(5):331 – 335, May 1996.

[VV96d] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. Simulation in Europe: a forum for basic research in modelling and simulation. In Bernard Zeigler, editor, *Proceedings of the AI, Simulation and Planning in High Autonomy Systems Conference,* pages 33–38 (addendum), Harvill Building Rm 135 Box 9, University of Arizona, PO Box 210076, Tucson AX 85721-0076, USA, March 1996. University of Arizona, Engineering Professional Development.

[VV97] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. Multi-formalism modelling and programming language types. In Windfried Hahn and Axel Lehmann, editors, *Simultion in Industry,* pages 105–109. Society for Computer Simulation International (SCS), October 1997. Passau, Germany.

[VV99] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. Integrating the WEST++ modeling and simulation environment with the AD RTS. In *Proceedings of the ADIUS Twentieth Annual Conference (Ann Arbor)*, 3800 Stone School Rd., Ann Arbor, Michigan 48108-2499, USA, June 1999. Applied Dynamics International.

[VV00] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. The cellular automata formalism and its relationship to DEVS. In Rik Van Landeghem, editor, 14^{th} *European Simulation Multi-conference (ESM)*, pages 800–810. Society for Computer Simulation International (SCS), May 2000. Gent, Belgium.

[VVB+00] H. Vanhooren, T. Verbrugge, G. Boeije, Demey. D., and P.A. Vanrolleghem. Adequate model complexity for scenario analysis of voc stripping in a trickling filter. *Wat. Sci. Tech. (in Press)*, 2000.

[VVKR97] Peter Vanrolleghem, Alexis Vanderhasselt, Peter Krebs, and Peter Reichert. Identification of a second-order one-dimensional clarifier model from on-line data. Technical Report 2, BIOMATH Department, Ghent, Belgium, 1997.

[VVKV89] Hans L. Vangheluwe, Jan D. Verweij, Eugène J.H. Kerckhoffs, and Ghislain C. Vansteenkiste. The object-oriented paradigm applied to continuous system simulation. In David Murray-Smith, John Stephenson, and Richard Zobel, editors, *Proceedings of the 3rd European Simulation Congress*, pages 163–169. Society for Computer Simulation International (SCS), September 1989. Edinburgh, Scotland.

[VVO88] Hans L. Vangheluwe, Ghislain C. Vansteenkiste, and Jean-Pierre Ottoy. SAPS, systems approach problem solver in Ctrl-C. In Eugène J.H. Kerckhoffs, Henk Koppelaar, Rik Van De Perre, and Ghislain C. Vansteenkiste, editors, *Proceedings of the SCSI USER1 Conference*, pages 206–209. Society for Computer Simulation International (SCS), September 1988. Ostend, Belgium.

[VVR97] Hans L. Vangheluwe, Ghislain C. Vansteenkiste, and Y.V. Reddy. Simformatics: Meaningful model storage, re-use, exchange and simulation. In Mohammad S. Obaidat and John Illgen, editors, *Summer Computer Simulation Conference, Arlington*, pages 936–941. Society for Computer Simulation International (SCS), July 1997. Arlington, USA.

[VVV90a] Hans L. Vangheluwe, Lode Vermeersch, and Ghislain C. Vansteenkiste. A continuous simulation program generator for the AD10. In *Proceedings of the ADIUS 1990 Conference*, 3800 Stone School Rd., Ann Arbor, Michigan 48108-2499, USA, 1990. Applied Dynamics International.

[VVV90b] Hans L. Vangheluwe, Lode Vermeersch, and Ghislain C. Vansteenkiste. Portable continuous simulation program generators. In *European Simulation Symposium (ESS90)*. Society for Computer Simulation International (SCS), November 1990. Gent, Belgium.

[VVV92] Hans L. Vangheluwe, Ghislain C. Vansteenkiste, and Lode Vermeersch. Physical systems modelling and simulation : a systematic approach to computer aided education. In *Proceedings of the Winter Simulation Conference*. Society for Computer Simulation International (SCS), January 1992. San Diego, CA.

[VVV+94] Hans Vangheluwe, Ghislain Vansteenkiste, Vladimir Visipkov, Yuri Merkuryev, Galina Merkuryeva, and Artis Teilans. *Design of a User Friendly Modelling and Simulation Environment*. In Antoni Guasch, editor, *International European Simulation Multi-conference (ESM)*, pages 282–286. Society for Computer Simulation International (SCS), June 1994. Barcelona, Spain.

[VVVV91] Peter A. Vanrolleghem, Lode Vermeersch, Hans L. Vangheluwe, and Ghislain C. Vansteenkiste. Introducing knowledge based predictive control in biological wastewater purification. In *CIM-Europe workshop "CIM in the Process Industry"*, March 1991.

[VVVW+91] Jan D. Verweij, Hans L. Vangheluwe, Danny Van Welden, Lode Vermeersch, Ghislain C. Vansteenkiste, and Eugène J.H. Kerckhoffs. An overview of the DESiRE project. In A. El Moudni, P. Borne, and S.G. Tzafestas, editors, *IMACS MCTS'91 "Modelling and Control of Technological Systems"*, volume 2, pages 738–744, Cité Scientifique, Villeneuve d'Asc Cedex, France, 1991. GERFIDN. Lille, France.

[VWVV89] Danny Van Welden, Hans L. Vangheluwe, and Ghislain C. Vansteenkiste. AI methods, useful tools for physical problems. In *CECAM*, Orsay, France, 1989.

[VZK+89] Jan D. Verweij, John J. Zenor, Eugène J.H. Kerckhoffs, Ghislain C. Vansteenkiste, and Hans L. Vangheluwe. A distributed simulation environment. In David Murray-Smith, John Stephenson, and Richard Zobel, editors, *Proceedings of the 3rd European Simulation Congress*, pages 56–64. Society for Computer Simulation International (SCS), September 1989. Edinburgh, Scotland.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[Weg95] Peter Wegner. Interactive foundations of object-based programming. *IEEE Computer*, 28(10):70–72, 1995.

[Wil90] E. Bright Wilson, Jr. *An Introduction to Scientific Research*. Dover Publications Inc., New York, 1990.

[Wir89] Niklaus Wirth. *Algoritmen en Datastructuren*. Academic Service, Postbus 81, 2870 AB Schoonhoven, 1989.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2), 1995.

[Wol86] Stephen Wolfram. *Theory and applications of cellular automata.* World Scientific, 1986.

[WSK96] R.W. Watts, S.A. Svoronos, and B. Koopman. One-dimensional modeling of secondary clarifiers using a concentration and feed velocity-dependent dispersion coefficient. *Wat.Res.*, 30(9):2112–2124, 1996.

[Wym67] A. Wayne Wymore. *A Mathematical Theory of Systems Engineering – the Elements.* Wiley Series on Systems Engineering and Analysis. Wiley, 1967.

[YVV98] Zhiguo Yuan, Hans L. Vangheluwe, and Ghislain C. Vansteenkiste. An observer-based approach to modeling error identification. *Transactions of the Society for Computer Simulation International*, 15(1):20–33, 1998.

[Zei84a] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, London, 1984.

[Zei84b] Bernard P. Zeigler. *Theory of Modelling and Simulation.* Robert E. Krieger, Malabar, Florida, 1984.

[Zei97] Bernard P. Zeigler. *Objects and Systems: Principled Design with Implementations in C++ and Java*. Springer-Verlag, 1997.

[Zei98] Bernard P. Zeigler. DEVS theory of quantized systems. Report, Advance Simulation Technology Thrust (ASST), DARPA contract N6133997K-0007, University of Arizona, Tucson, June 1998. www.acims.ece.arizona.edu.

[ZL98] Bernard P. Zeigler and J.S. Lee. Theory of quantized systems: Formal basis for DEVS/HLA distributed simulation environment. In Alex F. Sisti, editor, *Enabling Technology for Simulation Science II, SPIE AeroSense 98*, volume 3369, pages 49–58, August 1998. Orlando, FL.

[ZPK00] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, second edition, 2000.

Abbreviations

1-D	One Dimensional
2-D	Two Dimensional
ACM	Association for Computing Machinery
ACSL	A Continuous Simulation Language
ADSIM/RTS	Applied Dynamics International SIMulation language/Real Time Station
AI	Artificial Intelligence
ALG	Algebraic
API	Application Programmer's Interface
ASM1	Activated Sludge Model 1
ASM2	Activated Sludge Model 2
AST	Abstract Syntax Tree
BC	Boundary Condition
BIOMATH	Department of Applied Mathematics, Biometrics and Process Control
BNF	Backus-Naur Form
BOD	Biological Oxygen Demand
BR	Birth Rate
CA	Cellular Automaton
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CAM	Computer Aided Modelling
CASE	Computer Aided Software Engineering
CEC	Current Event Chain
CEL	Current Event List
CM	Communication Manager
COD	Chemical Oxygen Demand
CSE	Concurrent Simulation Engineering
CSSL	Continuous System Simulation Language
CVS	Concurrent Version System
DAE	Differential Algebraic Equation
DESS	Differential Equation Specified System
DEVS	Discrete EVent System
DFS	Depth First Search
DMSO	Defense Modeling and Simulation Office
DOME	Domain Modelling Environment
DR	Death Rate
DSblock	Dynamic System block
DoD	Department of Defense
EL	Event List
ES	Event Scheduling
ESPRIT	European Strategic Programme in Information Technology
EXP	Experimentation
FEC	Future Event Chain

FEL	Future Event List
FIFO	First In First Out
FSA	Finite State Automaton
FTG	Formalism Transformation Graph
GMSA	Generic Modelling and Simulation Architecture
GPSS	General Purpose Simulation System
HGE	Hierarchical Graphical Editor
HLA	High Level Architecture
I/O	Input/Output
IAT	Inter Arrival Time
IAWPRC	International Association on Water Pollution Research and Control
IAWQ	International Association for Water Quality
IP	Internet Protocol
IPM	Idealised Physical Models
IWA	International Water Association
LHS	Left Hand Side
M&S	Modelling and Simulation
MGA	Multigraph Architecture
MIMO	Multiple Input Multiple Output
MLQ	Material-Like Quantity
MSL	Model Specification Language
MSL-EXEC	Model Specification Language - Execution Level
MSL-USER	Model Specification Language - User Level
MSLU	MSL-USER
MoSS-CC	Modelling and Simulation-based System for Cost Calculation
MuPAD	Multi Processing Algebra Data tool
NI	Network Interface
ODE	Ordinary Differential Equation
P-DEVS	Parallel DEVS
PDE	Partial Differential Equation
PSA	Peugeot-Citroen
RCS	Revision Control System
RHS	Right Hand Side
RTI	Run Time Infrastructure
SCS	Society for Computer Simulation
SEI	Software Engineering Institute
SES	System Entity Structure
ST	Service Time
ST	Symbol Table
SiE-SIG	Simulation in Europe - Special Interest Group
SiE-WG	Simulation in Europe - Working Group
TCP	Transmission Control Protocol
TOC	Total Organic Carbon
UI	User Interface
UML	Unified Modellling Language
VPL	Virtual Product Life-Cycle
WEST++	Waste Water Treatment Environment for Simulation and Training
WWTP	Waste Water Treatment Plant
μCSL	micro Continuous Simulation Language

A
Lexical Scoping

```
--------------------------------------------------
-- COMPILER STAGES (passes)
--------------------------------------------------

'root'
   Stmnt(-> S)                    -- parse the concrete syntax
   CurrentScope <- nil            -- initially, the outer scope is nil
   CurrentScopeSetOrSeq <- seq    -- initialise (never used at level 0 however)
   CollectDeclarations(S -> Snew) -- setup Declarations table,
                                  -- replace each declaration by a reference
                                  -- into the table, replace each Identifier
                                  -- by a reference into the Applications
                                  -- table, and attach a list of
                                  -- declaration pointers (to Declarations
                                  -- made in that block) to each scope block.
   InitEnv                        -- initialise Environment info
   AssignDeclToApplic(Snew)       -- assign appropriately scoped object
                                  -- declarations to applications/identifiers
   PrintValues(Snew)              -- print values (objects) of identifiers

--------------------------------------------------
-- CONCRETE SYNTAX (of the prototype mini-language)
--------------------------------------------------

'nonterm' StmntList(-> STMNT)  -- a sequence of statements
   'rule' StmntList(-> S):
      Stmnt(-> S)
   'rule' StmntList(-> seq(S1, S2)):
      Stmnt(-> S1) StmntList(-> S2)

'nonterm' Stmnt(-> STMNT)
   'rule' Stmnt(-> scope(set, block(S), nil)): -- set
```

```
      "{" StmntList(–> S) "}"
   'rule' Stmnt(–> scope(seq, block(S), nil)): –– sequence
      "[" StmntList(–> S) "]"
   'rule' Stmnt(–> declare(Id, obj(Int))):      –– declaration
      "OBJ" Ident(–> Id) ":" Number(–> Int)
   'rule' Stmnt(–> expr(E)):                    –– expression
      Expr(–> E)

'nonterm' Expr(–> EXPR)
   'rule' Expr(–> appl(Id)):                    –– object application
      Ident(–> Id)

––————————————————————————————————-
–– ABSTRACT SYNTAX and GLOBAL DECLARATIONS
––————————————————————————————————-

–– For each declaration, an entry is added to the
–– object Symbol Table \verb|Declarations|. This entry can be filled
–– with the object's attributes. In different places, a reference
–– into this table will be used to access the object's attributes.
–– For the time being, we only keep track of an object's Id (via which we can
–– find out its String representation), and of the value given to
–– the object in that declaration.
'table' Declarations (Id:IDENT, Value:OBJECT)

–– For each application (in an expression) of an object, an entry is
–– added to the table \verb|Applications|. In this table, we
–– refer to the appropriate (scoped) entry \verb|Decl| in the Declarations table.
–– From \verb|Decl|, the \verb|Id| as well as \verb|Value| at declaration time
–– can be retrieved. When, during the \verb|CollectDeclarations()| pass,
–– \verb|appl(Id)| is replaced by \verb|appl_ref(X)| referring to
–– \verb|X|, an entry in the Applications table, the \verb|Id|
–– is entered into \verb|X|, as the appropriate \verb|Decl| is not yet known.
–– Only during the \verb|AssignDeclToApplic()| pass, will \verb|Decl| be
–– assigned.
–– This explains why \verb|Id:IDENT| is present in the
–– \verb|Applications| table {\em and} in the \verb|Declarations| table.
–– \verb|Applications| also has a \verb|LocalValue| entry, which is
–– where we can attach "annotations" (lists
–– of attribute-value pairs) to individual variable/identifier uses
–– (not used in this prototype).
'table' Applications (Id:IDENT, Decl:OBJECT, LocalValue:OBJECT)

–– For every scope (set or sequence), keep a reference to the next
–– enclosing scope
–– (or nil if this is the outermost scope), as well as
–– a list of references to declarations within that scope.
'table' Scopes (Outer:ScopeType, Decls:DeclarationList)

–– A list of references into the table of declarations
–– Such a list is added to the \verb|scope| node delimiting
```

```
-- a scope.
'type' DeclarationList
 list(Declarations, DeclarationList)
 nil

--
-- the actual ABSTRACT SYNTAX                                          90
--

'type' STMNT
 scope(SetOrSeq, STMNT, ScopeType)
 block(STMNT)              --
 declare(IDENT, OBJECT)
 decl_ref(Declarations) -- replaces declare() with a reference
                        -- into the SymbolTable "Declarations"
 expr(EXPR)
 seq(STMNT, STMNT)                                                   100
 deletedstmt           -- If declarations (or any other statement)
                       -- are deleted, they are replaced by
                       -- deletedstmt. Later, with another pass, all
                       -- deletedstmts may be removed.

'type' EXPR
 appl(IDENT)
 appl_ref(Applications) -- replaces appl() with a reference
                        -- into the table "Applications"
                                                                     110
'type' OBJECT
 obj(INT)
 obj_ref(Declarations)
 noobj

'type' SetOrSeq
 set
 seq

'type' ScopeType                                                     120
 scopeVars(Scopes)
 nil

--------------------------------------------------
-- SCOPE/NAME ANALYSIS
--------------------------------------------------

'var' CurrentScope: ScopeType

'action' CollectDeclarations(STMNT -> STMNT)                         130

  'rule' CollectDeclarations(scope(SetOrSeq, Stmt, nil)
         -> scope(SetOrSeq, NewStmt, scopeVars(Scope))):
   Scope :: Scopes       -- new Scopes table entry for this block scope
```

```
    CurrentScope -> CS    -- retrieve current scope
    Scope'Outer <- CS     -- set 'Outer scope for the block
    Scope'Decls <- nil    -- just entered scope, no decls yet
    CurrentScope <- scopeVars(Scope) -- set current scope to newly created Scope
    CollectDeclarations(Stmt -> NewStmt) -- recurse down
    Scope'Outer -> OuterScope    -- the scope of this level                    140
    CurrentScope <- OuterScope   -- restore the current scope

  'rule' CollectDeclarations(scope(SetOrSeq, Stmt,Scope)
         -> scope(SetOrSeq, Stmt,Scope)):
    Error("hould have found a nil OldScope !", 999)
    -- during parsing the Scope field was supposed to be set to nil

  'rule' CollectDeclarations(declare(Id, Obj) -> decl_ref(Declaration)):
    -- decl_ref() points into the SymbolTable for that declaration
    -- May later wish make one more pass to clean up the AST removing these    150
    -- if declarations turn out not to be needed any more.
    Declaration :: Declarations  -- a new declaration
    Declaration'Id <- Id         -- the Id it declares
    Declaration'Value <- Obj     -- value given at declaration time
    CurrentScope -> scopeVars(Sc)
    Sc'Decls -> DList
    Sc'Decls <- list(Declaration, DList) -- add declaration to current
                                         -- scope's declarationlist

  'rule' CollectDeclarations(block(Stmt) -> block(StmtNew)):                   160
    CollectDeclarations(Stmt -> StmtNew)

  'rule' CollectDeclarations(expr(appl(Id)) -> expr(appl_ref(X))):
    X :: Applications
    X'Id <- Id

  'rule' CollectDeclarations(seq(S1, S2) -> seq(S1New, S2New)):
    CollectDeclarations(S1 -> S1New)
    CollectDeclarations(S2 -> S2New)
                                                                             170
--------------------------------------------------
-- LEXICAL SCOPE HANDLING
--------------------------------------------------

  'type' EnvironmentList -- A list of enclosing Local environments
    env(Locals, EnvironmentList)
    emptyenv

-- Locals consists of a list of
-- 1. an identifier.                                                         180
-- 2. if the identifier had a meaning in the enclosing scope,
--      HiddenObject refers to the declaration in the enclosing
--      scope which is hidden in the local scope.
  'type' Locals
    locals(IDENT, HiddenObject:OBJECT, Locals)
```

```
 emptylocals

-- When traversing nested scopes, CurrentEnv holds
-- the EnvironmentList for this nesting level.
'var' CurrentEnv: EnvironmentList                                    190

-- When traversing nested scopes, CurrentLevel
-- indicates the current nesting depth. It may be used
-- for indentation etc.
'var' CurrentLevel: INT

-- When traversing nested scopes, CurrentScopeSetOrSeq
-- indicates whether we're in a set or a sequence scope.
-- This will influence the processing of decl_ref() nodes
-- which give rise to a Define() in a sequence scope and              200
-- are ignored in a set scope (as Define()ing was done in
-- DefineDecls().
'var' CurrentScopeSetOrSeq: SetOrSeq

-- Initialise the environment at the top level
'action' InitEnv

  'rule' InitEnv:
   CurrentEnv <- emptyenv   -- no environment
   CurrentLevel <- 0        -- no nesting yet                         210

-- enter a nested scope
-- save the enclosing environment
'action' EnterScope

  'rule' EnterScope:
   CurrentEnv -> Env                    -- the current environment
   CurrentEnv <- env(emptylocals, Env) -- becomes the enclosing
                                        -- environment in the next
                                        -- nesting                   220
                                        -- Initially, no locals found
   CurrentLevel -> N
   CurrentLevel <- N+1

-- leave a nested scope
-- restore the enclosing environment
'action' LeaveScope

  'rule' LeaveScope:
   CurrentEnv -> env(Locals, Env)                                    230
   ForgetLocals(Locals)
   CurrentEnv <- Env
   CurrentLevel -> N
   CurrentLevel <- N-1

-- Forget local meaning. Replace by outer scope meaning
```

```
-- where necessary.
-- If these locals are not defined in the outer scope, the Id stays
-- undefined. If a local Id is defined in an enclosing scope
-- (the environment), the meaning from the enclosing scope is restored
'action' ForgetLocals(Locals)

  -- no meaning in outer scopes
  'rule' ForgetLocals(locals(Id, noobj, Locals)):
   UndefMeaning(Id)
   ForgetLocals(Locals)

  -- forget local, restore outer scope meaning
  'rule' ForgetLocals(locals(Id, HiddenObject, Locals)):
   DefMeaning(Id, HiddenObject)
   ForgetLocals(Locals)

  -- nothing to forget
  'rule' ForgetLocals(emptylocals):

--------------------------------------------------------
-- RESOLVE/BIND NESTED IDENTIFIER APPLICATION TO DECLARATIONS
--------------------------------------------------------

'sweep' AssignDeclToApplic(ANY)

 'rule' AssignDeclToApplic(scope(SetOrSeq,Stmts,nil)):
  Error("Scope should have been assigned by now !", 999)

 'rule' AssignDeclToApplic(scope(set,Stmts,scopeVars(Scope))):
  CurrentScopeSetOrSeq -> SetOrSeq  -- this set is nested in a SetOrSeq
  CurrentScopeSetOrSeq <- set
  CurrentScope <- scopeVars(Scope) -- set CurrentScope to block's scope
  Scope'Decls -> DeclarationsInScope
  EnterScope
  DefineDecls(DeclarationsInScope)
  AssignDeclToApplic(Stmts)
  LeaveScope
  Scope'Outer -> OuterScope
  CurrentScope <- OuterScope    -- restore old scope
  CurrentScopeSetOrSeq <- SetOrSeq -- restore the nature of the
                                   -- enclosing scope

 'rule' AssignDeclToApplic(scope(seq,Stmts,scopeVars(Scope))):
  CurrentScopeSetOrSeq -> SetOrSeq  -- this seq is nested in a SetOrSeq
  CurrentScopeSetOrSeq <- seq
  CurrentScope <- scopeVars(Scope) -- set CurrentScope to block's scope
  EnterScope
  AssignDeclToApplic(Stmts)
  LeaveScope
  Scope'Outer -> OuterScope
  CurrentScope <- OuterScope    -- restore old scope
```

```
    CurrentScopeSetOrSeq <– SetOrSeq –– restore the nature of the
                                    –– enclosing scope
                                                                          290
  'rule' AssignDeclToApplic(decl_ref(Decl)):
   CurrentScopeSetOrSeq –> SetOrSeq
   eq(SetOrSeq, seq)  –– only within seq scope
   –– in case of set scope, the Defines have been done by means
   –– of DefineDecls()
   Decl'Id –> Id
   Define(Id, Decl)

  'rule' AssignDeclToApplic(appl_ref(X)):
   X'Id –> Id                                                             300
   Apply(Id –> DeclObj)
   X'Decl <– DeclObj
   –– this is where we could copy the declaration's Value into
   –– the application's LocalValue:
    where(DeclObj –> obj_ref(Decl))
    Decl'Value –> Val   –– copy of declared value, may be modified locally
    X'LocalValue <– Val –– set the value from the appropriate declaration

'action' DefineDecls(DeclarationList)
                                                                          310
   'rule' DefineDecls(list(Decl, DeclList)):
    DefineDecls(DeclList)
    Decl'Id –> Id
    Define(Id, Decl)

   'rule' DefineDecls(nil):

'action' Define(IDENT, Declarations)

   'rule' Define(Id, Decl)                                                320
    CurrentEnv –> env(Locals, Env)
    IsUndefined(Id, Locals)
    Hides(Id –> HiddenObject)
    CurrentEnv <– env(locals(Id, HiddenObject, Locals), Env)
    DefMeaning(Id, obj_ref(Decl))

   'rule' Define(Id, Decl)  –– only get here if IsUndefined() fails
    id_to_string(Id –> Name)
    print("Identifier") print(Name) print("declared again")
    print("ignoring this last declaration")                               330

'condition' IsUndefined(IDENT, Locals)

     'rule' IsUndefined(Id, locals(LocalId, Hidden, ListRest)) :
      ne (Id, LocalId)
      IsUndefined (Id, ListRest)

     'rule' IsUndefined(Id, emptylocals):
```

```
'action' Apply(IDENT –> OBJECT)

  'rule' Apply(Id –> O):
   HasMeaning(Id –> O)

  'rule' Apply(Id –> noobj):  -- if HasMeaning() fails
   id_to_string(Id –> Name)
   print("Identifier") print(Name) print(": no declaration found")
   print("(after searching nesting scopes)")

'action' Hides(IDENT –> OBJECT)

  'rule' Hides(Id –> O):
   HasMeaning(Id –> O)

  'rule' Hides(Id –> noobj): -- if HasMeaning() fails

--------------------------------------------------
-- PRINT SCOPED IDENTIFIER VALUES
--------------------------------------------------

'sweep' PrintValues(ANY)

 'rule' PrintValues(appl_ref(Applic)):
  Applic'Decl –> O
  PrintObject(O)

'action' PrintObject(OBJECT)

  'rule' PrintObject(noobj):
   print("noobj, this node should be removed from the AST")

  'rule' PrintObject(obj(N)):
   print(N)

  'rule' PrintObject(obj_ref(X)):
   X'Id –> ID
   id_to_string(ID –> Name)
   print(Name)
   X'Value –> obj(N)
   print(N)

--------------------------------------------------
-- BASIC IDENT OPERATIONS (opaque)
--------------------------------------------------

'type' IDENT

'action' string_to_id (STRING –> IDENT)
'action' id_to_string (IDENT –> STRING)
```

'**action**' DefMeaning (IDENT, OBJECT)
'**action**' UndefMeaning (IDENT)
'**condition**' HasMeaning (IDENT –> OBJECT)

```
--------------------------------------------------
-- ERROR and I/O HANDLING (opaque)
--------------------------------------------------

'action' Error (STRING, INT)

--------------------------------------------------
-- BASIC TOKENS
--------------------------------------------------

'token' Ident (–> IDENT)
'token' Number (–> INT)
```

B

MSL-USER Libraries

B.1 MSL-USER Generic Base Library

```
// Description: MSL-USER/Generic/Base definitions.

//#################################################################
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
// implementation: Hans Vangheluwe
// topic: generic/base definitions
// contact: Hans Vangheluwe
//#################################################################

#ifndef GENERIC_BASE
#define GENERIC_BASE

// Contains generic declarations for the modelling of
// dynamic (DAE based) physical systems.
//
// Builtin types are the only types for which
// an empty signature is allowed.
// During bootstrapping, the builtin type names
// are loaded into the outermost type namespace.
// The semantics of these types is given implicitly.
//
// Builtin atomic types
//

TYPE Generic "builtin: type variable";
// The Generic type is a "type variable". It will unify with any
// other type; any type is a sub-type of Generic which implies any
// object can be an instance of type Generic.

TYPE Integer "builtin: positive and negative Natural Numbers";

TYPE Real "builtin: Real numbers";

TYPE Char "builtin: ASCII character";

TYPE String
 "builtin: Char* (implemented as atomic type for efficiency reasons)";

TYPE Bottom "builtin: bottom type" = ENUM {null};
// The Bottom type is a sub-type of any other type.
// By virtue of this, "null", the only object of
// type Bottom, can be used to denote an unassigned value
// for objects of any type.

TYPE Boolean "builtin: Logic type" = ENUM {True, False};

//
// Builtin composite types
//

TYPE TypeDeclarationType
"builtin: type of TYPE declaration statement";
TYPE ClassDeclarationType
"builtin: type of CLASS declaration statement";
```

```
TYPE ObjectDeclarationType
"builtin: type of OBJ declaration statement";

TYPE DeclarationType
"type of a declaration (TYPE, CLASS, or OBJ) statement"
= UNION {TypeDeclarationType, ClassDeclarationType, ObjectDeclarationType};

TYPE ExpressionType
"builtin: type of expressions";

TYPE EquationType
"builtin: type of equations";

TYPE GenericIntervalType
"
 Generic Interval. Only meaningful if used
 to specialise with endpoints of a type for which
 an order relation is defined.
"
= RECORD
  {
   lowerBound: Generic;
   upperBound: Generic;
   lowerIncluded: Boolean;
   upperIncluded: Boolean;
  };

TYPE RealIntervalType "Interval of real numbers"
SUBSUMES GenericIntervalType =
RECORD
{
 lowerBound: Real; // Real is sub-type of Generic
 upperBound: Real; // Real is sub-type of Generic
 lowerIncluded: Boolean;
 upperIncluded: Boolean;
};


//
// type declarations for physical systems
//

TYPE UnitType
"The type of physical units. For the time being, a string"
= String;

TYPE QuantityType
"The different physical quantities. For the time being, string"
= String;

TYPE CausalityType
" Causality of entities:
   CIN: input (cause) only
   COUT: output (consequence) only
   CINOUT: input and output (cause and consequence) are allowed
"
= ENUM {CIN, COUT, CINOUT};

TYPE PhysicalNatureType
"The nature of physical variables
 FIELD is used (in the physicalDAE context) to denote
 parameters and constants
"
= ENUM {ACROSS, THROUGH, FIELD};

TYPE PhysicalQuantityType
"The type of any physical quantity"
=
RECORD
{
 quantity    : QuantityType;
 unit        : UnitType;
 interval    : RealIntervalType;
 value       : Real;
 causality   : CausalityType;
 nature      : PhysicalNatureType;
};

//
// Formalism independent model stuff
//

TYPE InterfaceDeclarationType
"declarations within an interface" = DeclarationType;

TYPE ParameterDeclarationType
"declarations within parameter section" = DeclarationType;

TYPE ModelDeclarationType
"declarations within sub˙models section" = DeclarationType;

TYPE CouplingStatementType
"parameter coupling and connect() statements" = EquationType;

TYPE GenericModelType
"The signature of the generic part of any (whatever the formalism) model"
```

```
=
RECORD
{
 comments    : String;
 interface   : SET_OF (InterfaceDeclarationType);
  // declared objects must be interfaces
 parameters  : SET_OF (ParameterDeclarationType);
  // declared objects must be parameters
};

TYPE CoupledModelType "The signature of a coupled (network) model"
EXTENDS GenericModelType WITH
RECORD
{
 sub_models : SET_OF (ModelDeclarationType);
 coupling   : SET_OF (CouplingStatementType);
};

TYPE DAEModelType
"The signature of a Differential Algebraic Equation (DAE) model
 within DAEModelType models, connect() has the following
 (flattening) semantics:
    quantity and unit are checked for equality
    equations are generated to equal (=) all algebraic and state variables
    all other labels are ignored
"
EXTENDS GenericModelType WITH
RECORD
{
 independent : SET_OF (ObjectDeclarationType); // independent variable (time)
 state       : SET_OF (PhysicalQuantityType); // variables
                                              // those variables occurring in
                                              // DERIV(v, [t]) statements are
                                              // derived state variables
 initial     : SET_OF (EquationType);
 equations   : SET_OF (EquationType);
 terminal    : SET_OF (EquationType);
};

TYPE PhysicalDAEModelType
"within physicalDAEModelType models, connect() has the
following
  (flattening) semantics:
    quantity and unit are checked for equality
    quantity and unit are checked for equality
    equations are generated to equal (=) all across variables
    equations are generated to sum all through variables to zero
    all other labels are ignored
"
= DAEModelType;

#endif
```

B.2 MSL-USER Generic Quantities Library

```
// Description: MSL-USER/Generic/SI quantity definitions.

//#####################################################################
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
// implementation: Frederik Decouttere
// topic: generic/SI quantity definitions
// contact: Jurgen Meirlaen
//#####################################################################

#ifndef GENERIC_QUANTITY_SI
#define GENERIC_QUANTITY_SI

// SI quantities, units, constants
//
// Based on the ISO 1000 standard
//


// Part 1: Space and time

   CLASS Angle
   "A class for angle"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "Angle";
    unit      <- "rad";
    displayunit <- "deg";
   :};

   CLASS SolidAngle
   "A class for SolidAngle"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "SolidAngle";
    unit      <- "sr";
```

```
:};

TYPE StringType
"The class for all kind of strings + some extra's"
=
RECORD
{
 quantity    : QuantityType;
 value       : String;
 unit        : UnitType;
};

CLASS Date
"A class for date"
SPECIALISES StringType :=
{:
 quantity  <- "Date";
:};

CLASS Time
"A class for time"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Time";
 unit      <- "d";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF :};
:};

CLASS Length
"A class for Length"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Length";
 unit      <- "m";
:};

CLASS Area
"A class for Area"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Area";
 unit      <- "m2";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF:};
:};

CLASS Volume
"Volume"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Volume";
 unit      <- "m3";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF:};
:};

CLASS AngularVelocity
"A class for AngularVelocity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "AngularVelocity";
 unit      <- "rad/d";
:};

CLASS Velocity
"A class for Velocity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Velocity";
 unit      <- "m/d";
:};

// every change of something
// per unit of time is in fact also
// a velocity, also called a rate

CLASS Rate
"A class for rate"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Rate";
 unit      <- "dUnit/dt";
:};

CLASS FlowRate
"Flow rate"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <- "FlowRate";
 unit     <- "m3/d";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF :};
:};

// DO NOT forget this is not the same as
// e.g. d(volume)/d(pH)
// which is in fact a ratio

CLASS Ratio
```

```
  "A class for ratio"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "Ratio";
   unit      <– "dUnit/dUnit";
  :};

  CLASS AngularAcceleration                                                    140
  "A class for AngularAcceleration"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "AngularAcceleration";
   unit      <– "rad/d2";
  :};

  CLASS Acceleration
  "A class for Acceleration"
  SPECIALISES PhysicalQuantityType :=                                          150
  {:
   quantity  <– "Acceleration";
   unit      <– "m/d2";
  :};

// Part 2: Periodic and related phenomens

  CLASS Frequency
  "The type of frequency"
  SPECIALISES QuantityType :=                                                  160
  {:
   quantity  <– "Frequency";
   unit      <– "Hz";
   interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF :};
  :};

// Part 3: Mechanics

  CLASS Mass
  "A class for Mass"                                                           170
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "Mass";
   unit      <– "g";
   interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF; :};
  :};

  CLASS Density
  "A class for Density"
  SPECIALISES PhysicalQuantityType :=                                          180
  {:
   quantity  <– "Density";
   unit      <– "g/m3";
   interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF; :};
  :};

  CLASS SpecificVolume
  "Specific volume ((density)^-1)"
  SPECIALISES PhysicalQuantityType :=
  {:                                                                           190
   quantity  <– "SpecificVolume";
   unit      <– "m3/g";
   interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF:};
  :};

  CLASS LinearDensity
  "A class for LinearDensity"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "LinearDensity";                                               200
   unit      <– "g/m";
   interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF; :};
  :};

  CLASS MomentOfInertia
  "A class for MomentOfInertia"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "MomentOfInertia";
   unit      <– "g*m2";                                                        210
  :};

  CLASS Momentum
  "A class for Momentum"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "Mass";
   unit      <– "g*m/d";
  :};
                                                                               220
  CLASS Force
  "A class for Force"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "Force";
   unit      <– "N";
  :};
```

```
CLASS AngularMomentum
"A class for AngularMomentum"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Mass";
 unit      <- "g*m2/d";
:};

CLASS MomentOfForce
"A class for MomentOfForce"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "MomentOfForce";
 unit      <- "N*m";
:};

CLASS Pressure
"A class for Pressure"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Pressure";
 unit      <- "Pa";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF; :};
:};

CLASS NormalStress
"A class for NormalStress"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "NormalStress";
 unit      <- "Pa";
:};

CLASS Diffusivity
"A class for Diffusivity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Diffusivity";
 unit      <- "m2/d";
:};

CLASS DynamicViscosity
"A class for DynamicViscosity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "DynamicViscosity";
 unit      <- "Pa*d";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF; :};
:};

CLASS KinematicViscosity
"A class for KinematicViscosity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "KinematicViscosity";
 unit      <- "m2/d";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF; :};
:};

CLASS SurfaceTension
"A class for SurfaceTension"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "SurfaceTension";
 unit      <- "N/m";
:};

CLASS Energy
"A class for Energy"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Energy";
 unit      <- "J";
:};

CLASS Power
"A class for Power"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "Power";
 unit      <- "W";
:};

// Part 4: Heat
CLASS KelvinTemperature
"A class for KelvinTemperature"
SPECIALISES PhysicalQuantityType :=
{:
 quantity     <- "KelvinTemperature";
 unit         <- "K";
 interval     <- {: lowerBound <- 0; upperBound <- PLUS_INF; :};
:};

CLASS CelsiusTemperature
"A class for CelsiusTemperature"
SPECIALISES PhysicalQuantityType :=
```

```
{:
 quantity  <– "CelsiusTemperature";
 unit      <– "degC";
 interval  <– {: lowerBound <– –273.15; upperBound <– PLUS_INF; :};
:};

CLASS LinearExpansionCoefficient
"A class for LinearExpansionCoefficient"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "LinearExpansionCoefficient";
 unit      <– "1/K";
:};

CLASS CubicExpansionCoefficient
"A class for CubicExpansionCoefficient"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "CubicExpansionCoefficient";
 unit      <– "1/K";
:};

CLASS RelativePressureCoefficient
"A class for RelativePressureCoefficient"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "RelativePressureCoefficient";
 unit      <– "1/K";
:};

CLASS PressureCoefficient
"A class for PressureCoefficient"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "PressureCoefficient";
 unit      <– "Pa/K";
:};

CLASS IsothermalCompressibility
"A class for IsothermalCompressibility"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "IsothermalCompressibility";
 unit      <– "1/Pa";
:};

CLASS IsentropicCompressibility
"A class for IsentropicCompressibility"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "IsentropicCompressibility";
 unit      <– "1/Pa";
:};

CLASS Heat = Energy;

CLASS HeatFlowRate
"A class for HeatFlowRate"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "HeatFlowRate";
 unit      <– "W";
:};

CLASS DensityOfHeatFlowRate
"A class for DensityOfHeatFlowRate"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "DensityOfHeatFlowRate";
 unit      <– "W/m2";
:};

CLASS ThermalConductivity
"A class for ThermalConductivity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "ThermalConductivity";
 unit      <– "W/(m*K)";
:};

CLASS CoefficientOfHeatTransfer
"A class for CoefficientOfHeatTransfer"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "CoefficientOfHeatTransfer";
 unit      <– "W/(m2*K)";
:};

CLASS SurfaceCoefficientOfHeatTransfer
"A class for SurfaceCoefficientOfHeatTransfer"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "SurfaceCoefficientOfHeatTransfer";
 unit      <– "W/(m2*K)";
:};
```

```
CLASS ThermalInsulance
"A class for ThermalInsulance"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "ThermalInsulance";
 unit      <– "m2*K/W";
:};

CLASS ThermalResistance
"A class for ThermalResistance"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "ThermalResistance";
 unit      <– "K/W";
:};

CLASS ThermalConductance
"A class for ThermalConductance"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "ThermalConductance";
 unit      <– "W/K";
:};

CLASS ThermalDiffusivity
"A class for ThermalDiffusivity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "ThermalDiffusivity";
 unit      <– "m2/d";
:};

CLASS HeatCapacity
"A class for HeatCapacity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "HeatCapacity";
 unit      <– "J/K";
:};

CLASS SpecificHeatCapacity
"A class for SpecificHeatCapacity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "SpecificHeatCapacity";
 unit      <– "J/(g*K)";
:};

// The specific heat capacity is most often taken in a “direction”
// i. e. at constant pressure or constant volume. which one is meant
// should be specified in the appropriate aliases

CLASS RatioOfspecificHeatCapacities
"A class for RatioOfSpecificHeatCapacities"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "RatioOfSpecificHeatCapacities";
 unit      <– "-";
:};

CLASS IsentropicExponent
"A class for IsentropicExponent"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "IsentropicExponent";
 unit      <– "-";
:};

CLASS Entropy
"A class for Entropy"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "Entropy";
 unit      <– "J/K";
:};

CLASS SpecificEntropy
"A class for SpecificEntropy"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "SpecificEntropy";
 unit      <– "J/(g*K)";
:};

CLASS SpecificEnergy
"A class for SpecificEnergy"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "SpecificEnergy";
 unit      <– "J/g";
:};

// In thermodynamics, energy comes in many flavors. The ones defined
// by the ISO are defined as aliases to the basic one.
// All of these energy forms are also
// defined in a specific, i. e. divided by mass version.
```

```
CLASS ThermodynamicEnergy          = Energy;
CLASS HelmholtzFreeEnergy          = Energy;
CLASS GibbsFreeEnergy              = Energy;
CLASS Enthalpy                     = Energy;

CLASS SpecificThermodynamicEnergy = SpecificEnergy;
CLASS SpecificHelmholtzFreeEnergy = SpecificEnergy;
CLASS SpecificGibbsFreeEnergy     = SpecificEnergy;
CLASS SpecificEnthalpy            = SpecificEnergy;


CLASS PlanckFunction
"A class for PlanckFunction"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "PlanckFunction";
 unit      <– "J/g";
:};

// Part 5: Electricity and magnetism
CLASS ElectricCurrent
"A class for ElectricCurrent"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "ElectricCurrent";
 unit      <– "A";
:};

CLASS ElectricCharge
"A class for ElectricCharge"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "ElectricCharge";
 unit      <– "C";
:};

CLASS ElectricPotential
"A class for ElectricPotential"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "ElectricPotential";
 unit      <– "V";
:};

CLASS Capacitance
"A class for Capacitance"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "Capacitance";
 unit      <– "F";
 interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF; :};
:};

CLASS Inductance
"A class for Inductance"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "Inductance";
 unit      <– "H";
 interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF; :};
:};

CLASS Resistance
"A class for Resistance"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "Resistance";
 unit      <– "Ohm";
 interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF; :};
:};

CLASS Conductance
"A class for Conductance"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "Conductance";
 unit      <– "S";
 interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF; :};
:};


// Part 6: Light and related electromagnetic radiations

CLASS LuminousIntensity
"A class for LuminousIntensity"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <– "LuminousIntensity";
 unit      <– "cd";
:};


// Part 7: Physical chemistry and molecular physics

CLASS AmountOfSubstance
```

```
  "A class for AmountOfSubstance"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "AmountOfSubstance";
   unit      <– "mol";
   interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF; :};
  :};

// often used in chemistry, so I put it here

  CLASS pH
  "A class for pH"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "pH";
   interval  <– {: lowerBound <– 0; upperBound <– 14 :};
  :};

  CLASS Concentration
  "A class for concentration"
  SPECIALISES PhysicalQuantityType :=
  {:
   nature <– "ACROSS";
   quantity <– "Concentration";
   unit <– "g/m3";
   interval <– {: lowerBound <– 0; upperBound <– PLUS_INF :};
  :};

// Part 8: Characteristic numbers
// Momentum transport

  CLASS ReynoldsNumber
  "A class for ReynoldsNumber"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "ReynoldsNumber";
   unit      <– "-";
  :};

  CLASS EulerNumber
  "A class for EulerNumber"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "EulerNumber";
   unit      <– "-";
  :};

  CLASS FroudeNumber
  "A class for FroudeNumber"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "FroudeNumber";
   unit      <– "-";
  :};

  CLASS GrashofNumber
  "A class for GrashofNumber"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "GrashofNumber";
   unit      <– "-";
  :};

  CLASS WeberNumber
  "A class for WeberNumber"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "WeberNumber";
   unit      <– "-";
  :};

  CLASS MachNumber
  "A class for MachNumber"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "MachNumber";
   unit      <– "-";
  :};

  CLASS KnudsenNumber
  "A class for KnudsenNumber"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "KnudsenNumber";
   unit      <– "-";
  :};

  CLASS StrouhalNumber
  "A class for StrouhalNumber"
  SPECIALISES PhysicalQuantityType :=
  {:
   quantity  <– "StrouhalNumber";
   unit      <– "-";
  :};
```

```
// Transport of heat
   CLASS FourierNumber
   "A class for FourierNumber"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "FourierNumber";
    unit      <- "-";
   :};

   CLASS PecletNumber
   "A class for PecletNumber"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "PecletNumber";
    unit      <- "-";
   :};

   CLASS RayleighNumber
   "A class for RayleighNumber"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "RayleighNumber";
    unit      <- "-";
   :};

   CLASS NusseltNumber
   "A class for NusseltNumber"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "NusseltNumber";
    unit      <- "-";
   :};

   CLASS BiotNumber = NusseltNumber;
// The name Biot number, Bi, is used
// when the Nusselt number is reserved
// for convective transport of heat.

   CLASS StantonNumber
   "A class for StantonNumber"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "StantonNumber";
    unit      <- "-";
   :};

// Constants of matter
   CLASS PrandtlNumber
   "A class for PrandtlNumber"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "PrandtlNumber";
    unit      <- "-";
   :};

   CLASS SchmidtNumber
   "A class for SchmidtNumber"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "SchmidtNumber";
    unit      <- "-";
   :};

   CLASS LewisNumber
   "A class for LewisNumber"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <- "LewisNumber";
    unit      <- "-";
   :};

//
//end SI unit;
//

//
// begin constants of nature
//

// (from: E.R. Cohen, and B.N. Taylor: The 1986 Adjustment of the Fundamental
//         Physical Constants, CODATA Bulletin, Pergamon: Elmsford, NY, 1986.
//         see also: http://physics.nist.gov/PhysRefData/codata86/article.html
//                   http://physics.nist.gov/PhysRefData/codata86/codata86.html)
//

  OBJ C "Velocity of light in vacuum" : Velocity :=
  {:
    value     <- 299792458;
  :};

  OBJ G_EARTH "Gravity acceleration on earth" : Acceleration :=
  {:
    value     <- 9.81;
  :};
```

```
OBJ AvogadroConstant
 "The Avogadro Constant" : PhysicalQuantityType :=
 {:
  quantity  <- "AvogadroConstant`NA";
  unit      <- "1/mol";
  value     <- 6.0221367E23;
 :};

OBJ UniversalGravityConstant
"Universal gravity constant" : PhysicalQuantityType :=
 {:
  quantity  <- "G";
  unit      <- "m3/(g*s2)";
  value     <- 6.67259E-11;
 :};

OBJ PlancksConstant
"Plancks constant" : PhysicalQuantityType :=
 {:
  quantity  <- "H";
  unit      <- "J*s";
  value     <- 6.6260755E-34
 :};

OBJ BoltzmannConstant
"Boltzmann constant" : PhysicalQuantityType :=
 {:
  quantity  <- "K";
  unit      <- "J/K";
  value     <- 1.380658E-23
 :};

OBJ UniversalGasConstant
"Universal gas constant" : PhysicalQuantityType :=
 {:
  quantity  <- "RO";
  unit      <- "J/(mol*K)";
  value     <- 8.314510
 :};

OBJ StefanBoltzmannConstant
"Stefan Boltzmann constant" : PhysicalQuantityType :=
 {:
  quantity  <- "SIGMA";
  unit      <- "W/(m2*K4)";
  value     <- 5.67051E-8
 :};

OBJ AbsoluteZeroTemperature
"Absolute zero temperature" : PhysicalQuantityType :=
 {:
  quantity  <- "T`ZERO";
  unit      <- "degC";
  value     <- -273.15
 :};

#endif
```

B.3 MSL-USER System Dynamics Library

```
// ********************************************************************
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
//
// Project: WEST++/MSL-USER
// File: sd.msl
// Type: MSL
// Author: hv
// Date: $Date: 1999/11/14 13:33:02 $
// Revision: $Revision: 1.5 $
// ********************************************************************

// Description: MSL-USER/System Dynamics model base.

#ifndef SD
#define SD

#include "generic.msl"

// don't specify quantity, unit, ...
CLASS SDTerminal SPECIALISES PhysicalQuantityType;

CLASS ConstantClass
  (* class = "constant"; category = "" *)
  "Constant:
   Produces at its output 'out', the value of the parameter 'c'"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
```

```
    {
      OBJ out (* terminal = "out" *) "out" :
        SDTerminal := {: causality <- "COUT" :};
    };
    parameters <-
    {
      OBJ c "c" : SDTerminal := {: value <- 0; :};
    };
    independent <-
    {
      OBJ t "t": Time;
    };
    equations <-
    {
      interface.out = parameters.c;
    };
  :};

CLASS PopulationClass
  (* class = "levelNrate"; category = "" *)
  "System Dynamics Population:
   Produces at its output 'level', the solution
   of the differential equation
    d level/d t = birth`rate - death`rate"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
      OBJ birth_rate (* terminal = "birth`rate" *) "birth`rate" :
        SDTerminal := {: causality <- "CIN"; value <- 0 :};
      OBJ death_rate (* terminal = "death`rate" *) "death`rate" :
        SDTerminal := {: causality <- "CIN"; value <- 0 :};
      OBJ level (* terminal = "level" *) "population level" :
        SDTerminal := {: causality <- "COUT" :};
    };
    independent <-
    {
      OBJ t "t": Time;
    };
    state <-
    {
      OBJ population "population level": SDTerminal;
      OBJ pop_change_rate "population level change rate": SDTerminal;
    };
    equations <-
    {
      DERIV(state.population, [independent.t]) = state.pop_change_rate;
      state.pop_change_rate = interface.birth_rate - interface.death_rate;
      interface.level = state.population;
    };
  :};

CLASS Product1Class
  (* class = "product1"; category = "" *)
  "Product:
   Produces at its output 'out', the product
   of one input 'in' and one parameter 'c'"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
      OBJ in (* terminal = "in" *) "in" : SDTerminal :=
        {: causality <- "CIN" :};
      OBJ out (* terminal = "out" *) "out" : SDTerminal :=
        {: causality <- "COUT" :};
    };
    parameters <-
    {
      OBJ c "c" : SDTerminal;
    };
    independent <-
    {
      OBJ t "t": Time;
    };
    equations <-
    {
      interface.out = parameters.c * interface.in;
    };
  :};

CLASS Product2Class
  (* class = "product2"; category = "" *)
  "Product:
   Produces at its output 'out', the product
   of its 2 inputs 'in`1' and 'in`2' and one parameter 'c'"
  SPECIALISES
  PhysicalDAEModelType :=
  {:
    interface <-
    {
      OBJ in_1 (* terminal = "in`1" *) "in`1" :
        SDTerminal := {: causality <- "CIN" :};
      OBJ in_2 (* terminal = "in`2" *) "in`2" :
        SDTerminal := {: causality <- "CIN" :};
      OBJ out (* terminal = "out" *) "out" :
        SDTerminal := {: causality <- "COUT" :};
```

```
    };
    parameters <–
    {
      OBJ c "c" : SDTerminal;
    };
    independent <–
    {
      OBJ t "t": Time;
    };
    equations <–
    {
      interface.out = parameters.c * interface.in_1 * interface.in_2;
    };
  :};

CLASS DivisionClass
  (* class = "division"; category = "" *)
  "Division:
    2 inputs: denominator and divider
    1 output: output
    output = mult`factor*denominator/divider"
  SPECIALISES
  PhysicalDAEModelType :=
  {:
    interface <–
    {
      OBJ denominator (* terminal = "denominator" *) "denominator" :
      SDTerminal :=
      {:
        causality <– "CIN";
        value <– 1;
      :};
      OBJ divider (* terminal = "divider" *) "divider" : SDTerminal :=
      {:
        causality <– "CIN";
        value <– 1;
      :};
      OBJ output (* terminal = "output" *) "output" :
        SDTerminal := {: causality <– "COUT" :};
    };
    parameters <–
    {
      OBJ mult_factor "multiplication factor" : SDTerminal;
    };
    independent <–
    {
      OBJ t "t": Time;
    };
    equations <–
    {
      interface.output = parameters.mult_factor *
                         interface.denominator / interface.divider;
    };
  :};

CLASS SumClass
  (* class = "sum"; category = "" *)
  "Sum:
   Produces the weighted sum of its 2 inputs on its output.

    out = weight`1*in`1 + weight`2*in`2

   By default the weight factors are both 1 and the
   inputs are both 0.

   For a pure subtraction, it is sufficient to set
    weight`1 = 1
    weight`2 = -1
  "
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <–
    {
      OBJ in_1 (* terminal = "in`1" *) "in`1" : SDTerminal :=
      {:
        causality <– "CIN";
        value <– 0;
      :};
      OBJ in_2 (* terminal = "in`2" *) "in`2" : SDTerminal :=
      {:
        causality <– "CIN";
        value <– 0;
      :};
      OBJ out (* terminal = "out" *) "out" :
        SDTerminal := {: causality <– "COUT" :};
    };
    parameters <–
    {
      OBJ weight_1 "weight factor of input in`1" : SDTerminal :=
      {: value <– 1; :};
      OBJ weight_2 "weight factor of input in`2" : SDTerminal :=
      {: value <– 1; :};
    };
    independent <–
    {
      OBJ t "t": Time;
```

```
    };
    equations <-
    {
      interface.out = parameters.weight_1*interface.in_1 +
                      parameters.weight_2*interface.in_2;
    };
  :};

CLASS SwitchClass
  (* class = "switch"; category = "" *)
  "Switch:
   Based on the value a 'condition' input,
   produces on its 'output' port
    the value of the 'in˙true' port if (condition ¿ 0)
    the value of the 'in˙false' port if (condition ¡= 0)
  "
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
      OBJ condition (* terminal = "condition" *) "condition" : SDTerminal :=
      {:
        causality <- "CIN";
        value <- 1;
      :};
      OBJ in_true (* terminal = "in˙true" *)
      "input whose value is copied in case (condition ¿ 0)" : SDTerminal :=
      {:
        causality <- "CIN";
        value <- 0;
      :};
      OBJ in_false (* terminal = "in˙false" *)
      "input whose value is copied in case (condition ¡= 0)" : SDTerminal :=
      {:
        causality <- "CIN";
        value <- 0;
      :};
      OBJ output (* terminal = "output" *) "output" :
        SDTerminal := {: causality <- "COUT" :};
    };
    independent <-
    {
      OBJ t "t": Time;
    };
    equations <-
    {
      interface.output =
        IF (interface.condition > 0) THEN
          interface.in_true
        ELSE
          interface.in_false;
    };
  :};

#endif
```

B.4 MSL-USER WWTP Quantities Library

```
// Description: MSL-USER/WWTP/Quantity definitions.

//#################################################################
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
// implementation: Frederik Decouttere, Henk Vanhooren
// topic: wwtp quantities
// contact: Henk Vanhooren
//#################################################################

#ifndef WWTP_QUANTITY
#define WWTP_QUANTITY

// vector CLASSES defined for general purposes
// length == NrOfComponents
// type of elements declared in the class-naming
// e.g MassVector == vector containing masses
CLASS MassVector = Mass[NrOfComponents;];
CLASS MassFluxVector = MassFlux[NrOfComponents;];
CLASS ConcentrationVector = Concentration[NrOfComponents;];
CLASS SpecificVolumeVector = SpecificVolume[NrOfComponents;];
CLASS VelocityVector = Velocity[NrOfComponents;];
CLASS ArealFluxVector = ArealFlux[NrOfComponents;];
CLASS LengthVector = Length[NrOfLayers;];

// vector CLASSES used in the Takacs model
CLASS TakacsMassVector = Mass[NrOfLayers;];
CLASS TakacsConcentrationVector = Concentration[NrOfLayers;];
CLASS TakacsVelocityVector = Velocity[NrOfLayers;];
CLASS TakacsArealFluxVector = ArealFlux[NrOfLayers;];

// Contains class definitions for the WWTP domain quantities.
// This is far more messy than the equivalent quantity definitions
```

```
// for electrical, mechanical, ... domains.
//
// Rather than using concentrations, the generic models
// are expressed in terms of masses and fluxes:

CLASS MassFlux
"Mass per time unit"
SPECIALISES PhysicalQuantityType :=
{:
 quantity  <- "MassFlux";
 unit      <- "g/d";
 interval  <- {: lowerBound <- MIN_INF; upperBound <- PLUS_INF :};
 nature <- "THROUGH";
:};

CLASS ArealFlux
"Mass per unit of surface and per unit of time"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <- "ArealFlux";
 unit     <- "g/(m2*d)";
 interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF :};
:};

// ASM1

// stoichiometric parameters

  CLASS YieldForAutotrophicBiomass
  "A class for YieldForAutotrophicBiomass"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <- "Y_A";
    unit      <- "gCOD/gN";
    interval  <- {: lowerBound <- 0; upperBound <- 4.57 :};
  :};

  CLASS YieldForHeterotrophicBiomass
  "A class for YieldForHeterotrophicBiomass"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <- "Y_H";
    unit      <- "gCOD/gCOD";
    interval  <- {: lowerBound <- 0; upperBound <- 1 :};
  :};

  CLASS FractOfBiomassLeadingToPartProd
  "Fraction of biomass leading to particulate products"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <- "F_P";
    unit      <- "-";
    interval  <- {: lowerBound <- 0; upperBound <- 1 :};
  :};

  CLASS MassOfNitrogenPerMassOfCODInBiomass
  "Mass of N per mass of COD in biomass"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <- "I_XB";
    unit      <- "gN/gCOD";
    interval  <- {: lowerBound <- 0; upperBound <- 0.2 :};
  :};

  CLASS MassOfNitrogenPerMassOfCODInProdFromBiomass
  "Mass of N per mass of COD in products from biomass"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <- "I_XP";
    unit      <- "gN/gCOD";
    interval  <- {: lowerBound <- 0; upperBound <- 0.2 :};
  :};

// kinetic parameters

  CLASS MaxSpecifGrowthRateHetero
  "Maximum specific growth rate for heterotrophic biomass"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <- "Mu_H";
    unit      <- "1/d";
    interval  <- {: lowerBound <- 0; upperBound <- 20 :};
  :};

  CLASS MaxSpecifGrowthRateAutotr
  "Maximum specific growth rate for autotrophic biomass"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <- "Mu_A";
    unit      <- "1/d";
    interval  <- {: lowerBound <- 0; upperBound <- 5 :};
  :};

  CLASS HalfSatCoeff
  "Half-saturation coefficient"
  SPECIALISES PhysicalQuantityType :=
```

```
{:
  quantity   <– "K`";
  unit       <– "gCOD/m3";
  interval   <– {: lowerBound <– 0; upperBound <– 1000000 :};
:};

CLASS HalfSatCoeffForHetero
"Half-saturation coefficient for heterotrophic biomass"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "K`S";
  unit       <– "gCOD/m3";
  interval   <– {: lowerBound <– 0; upperBound <– 100 :};
:};

CLASS OxygenHalfSatCoeffForHetero
"Oxygen half-saturation coeff for heterotrophic biomass"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "K`OH";
  unit       <– "gO2/m3";
  interval   <– {: lowerBound <– 0; upperBound <– 10 :};
:};

CLASS NitrateHalfSatCoeffForDenitrifHetero
"Nitrate half-saturation coeff for denitrifying heterotrophic biomass"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "K`NO";
  unit       <– "gNO3-N/m3g`NO3-N*m^-3";
  interval   <– {: lowerBound <– 0; upperBound <– 2 :};
:};

CLASS OxygenHalfSatCoeffForAutotr
"Oxygen half-saturation coeff for autotrophic biomass"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "K`OA";
  unit       <– "gO2/m3";
  interval   <– {: lowerBound <– 0; upperBound <– 10 :};
:};

CLASS AmmonHalfSatCoeffForAutotr
"Ammonium half saturation coeff for autotrophic biomass"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "K`NH";
  unit       <– "gNH3-N/m3";
  interval   <– {: lowerBound <– 0; upperBound <– 10 :};
:};

CLASS HalfSatCoeffForHydrolSlowBioDegradeSubstr
"Half saturation constant for hydrolysis of slowly biodegradable substrate"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "K`X";
  unit       <– "gCOD/gCOD";
  interval   <– {: lowerBound <– 0; upperBound <– 1 :};
:};

CLASS MaxSpecificHydrolysisRate
"Maximum specific hydrolysis rate"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "K`h";
  unit       <– "gCOD/(gCOD*d)";
  interval   <– {: lowerBound <– 0; upperBound <– 25 :};
:};

CLASS AmmonificationRate
"Ammonification rate"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "K`a";
  unit       <– "m3/(gCOD*d)";
  interval   <– {: lowerBound <– 0; upperBound <– 0.25 :};
:};

CLASS DecayCoeffHeterotr
"Decay coefficient for heterotrophic biomass"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "B`H";
  unit       <– "1/d";
  interval   <– {: lowerBound <– 0; upperBound <– 25 :};
:};

CLASS DecayCoeffAutotr
"Decay coefficient for autotrophic biomass"
SPECIALISES PhysicalQuantityType :=
{:
  quantity   <– "B`A";
  unit       <– "1/d";
  interval   <– {: lowerBound <– 0; upperBound <– 25 :};
:};
```

```
// ASM2

  CLASS DissolvedComponent
  "A class for dissolved components
  change quantity in object to specialize
  e.g. SO2 for dissolved oxygen"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <– "S";
    unit      <– "g/m3";
    interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF :};
    nature <– "ACROSS";
  :};

  CLASS ParticulateComponent
  "A class for particulate component
  same remark as for DissolvedComponent"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <– "X";
    unit      <– "g/m3";
    interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF :};
    nature <– "ACROSS";
  :};

  CLASS ConversionFactor
  "A class for typical conversion factors for continuity equations
   from ASM2"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <– "I";
    unit      <– "g/gCOD";
    interval  <– {: lowerBound <– 0; upperBound <– 1 :};
  :};

  CLASS MaxGrowthRate
  "Maximum growth rate"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <– "Mu";
    unit      <– "1/d";
    interval  <– {: lowerBound <– 0; upperBound <– 20 :};
  :};

  CLASS RateConstant
  "Rate constant"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <– "Q˙or˙B˙or˙K";
    unit      <– "1/d";
    interval  <– {: lowerBound <– 0; upperBound <– 20 :};
  :};

// some important terms || abbreviations || parameters
// used in wwtp

  CLASS ChemOxDemand
  "Chemical oxygen demand is
  the amount of oxygen required to completely oxidize
  organic carbon to CO2 by chemical means"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <– "COD";
    unit      <– "gO2/m3";
  :};

  CLASS BiolOxDemand
  "Biological oxygen demand (BOD˙5˙20):
  amount of oxygen used by non-photosynthetic micro-organisms
  at 20 C to metabolize biologically degradable organic compounds
  measured over a period of 5 days"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <– "BOD˙5˙20";
    unit      <– "gO2/m3";
  :};

  CLASS NitrifOxDemand
  "Nitrification oxygen demand"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <– "NOD";
    unit      <– "gO2/m3";
  :};

  CLASS DissolvedOxygen
  "A class for the ammount of dissolved oxygen"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity     <– "DO";
    unit         <– "gO2/m3";
    displayunit <– "gO2/m3";
    interval     <– {: lowerBound <– 0; upperBound <– 15 :};
  :};

//————————————
```

```
//————————————

CLASS Yield
"A class for Yield"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "Yield";
 unit     <– "-";
 interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF:};
:};

CLASS GrowthRate
"GrowthRate"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "GrowthRate";
 unit     <– "1/d";
 interval  <– {: lowerBound <– 0; upperBound <– 20:};
:};

CLASS Fraction
"Fraction"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "Fraction";
 unit     <– "-";
 interval  <– {: lowerBound <– 0; upperBound <– 1:};
:};

CLASS SaturationCoefficient
"Saturation coefficient"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "K";
 unit     <– "-";
 interval  <– {: lowerBound <– 0; upperBound <– 100:};
:};

CLASS DecayCoefficient
"Decay coefficient"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "B";
 unit     <– "1/d";
 interval  <– {: lowerBound <– 0; upperBound <– 20:};
:};

CLASS CorrectionFactor
"CorrectionFactor"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "eta";
 unit     <– "-";
 interval  <– {: lowerBound <– 0; upperBound <– 1 :};
:};

CLASS ReductionFactor
"Reductionfactor"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "eta";
 unit     <– "-";
 interval  <– {: lowerBound <– 0; upperBound <– 1 :};
:};

CLASS MaxSpecAmmonRate
"Maximum specific ammonification rate"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "MaxSpecAmmonRate";
 unit     <– "m3/(gCOD*d)";
 interval  <– {: lowerBound <– 0; upperBound <– PLUS_INF:};
:};

CLASS OxygenTransferCoefficient
"Oxygen Transfer Coefficient"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "Kla";
 unit     <– "1/d";
 interval  <– {: lowerBound <– 0; upperBound <– 5000 :};
:};

CLASS OxygenUptakeRate
"Oxygen Uptake Rate"
SPECIALISES PhysicalQuantityType :=
{:
 quantity <– "OUR";
 unit     <– "g/(m3.d)";
:};

CLASS ElectricalEnergy
   "A class for electrical energy"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity  <– "Electrical energy";
```

```
    unit        <– "kWh";
    :};

CLASS Dollar
   "dollars"
   SPECIALISES PhysicalQuantityType :=
   {:
    quantity   <– "dollar";
    unit       <– "$";
    :};


// ASM2d classes

#ifndef Class_ASM2d
#define Class_ASM2d

  CLASS MonodTerm
  "A class for Monod-like terms"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity   <– "S / ( K + S)";
    unit       <– "-";
    interval   <– {: lowerBound <– 0; upperBound <– 1:};
  :};

  CLASS InhibitionTerm
  "A class for inhibtion terms of the ASM-models"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity   <– "K / ( K + S)";
    unit       <– "-";
    interval   <– {: lowerBound <– 0; upperBound <– 1:};
  :};

#endif // CLASS_ASM2d

#endif
```

B.5 MSL-USER WWTP Base Library

```
// Description: MSL-USER/WWTP/Base definitions.

//#################################################################
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
// implementation: Hans Vangheluwe, Frederik Decouttere,
//                 Henk Vanhooren, Peter Vanrolleghem
// topic: basic module for wwtp modelbase, extending generic modules
// contact: Hans Vangheluwe, Henk Vanhooren, Peter Vanrolleghem
//#################################################################

#ifndef WWTP_BASE
#define WWTP_BASE

// This library includes non-causal models as well as
// an inheritance hierarchy to separate
// different model aspects and enhance model re-use.

// Contains declarations to describe
// Waste Water Treatment Plants (WWTPs)
//
// With this library as a starting point, the modeller
// only needs to specify relevant biological components
// (e.g., H2O, S_S, X_S, ...) as well as Stochiometric
// and Kinetic conversion information (from IAWQ).
// Also, any number of terminals (physical flow
// of matter in/out of a sub-system, but also
// control and information terminals) may be added to
// the model.
// Once the above are specified, the appropriate model
// will automatically be expanded. Currently, automatic
// expansion only takes into account the hydrological,
// chemical, and biological aspects.
//

 // The following Components TYPE declaration is commented
 // as it will be specified further on.
 // As the order of declarations does not matter in MSL,
 // the actual place of declaration of the Components TYPE
 // does not matter.
 // We already present a TYPE Components definition here
 // as it will make the rest of the generic model
 // easier to understand.

 // In the Components type declaration, the user indicates
 // which components will be used in his/her models.
 // A number of assumptions are made:
 //  1. In one system, ALL the connections between
 //       sub-models pass exactly those biological components
 //       indicated in the Components declaration.
```

```
//  2. In a physical flow, ALL components are explicitly
//       considered: H2O, dissoved gasses, solids in suspension, ...
//       The usual assumption that the concentration of H2O is 1
//       and all the other concentrations are infinite will be put in
//       further. This, to allow modelling of systems where
//       the above assumption is not valid.

// EVERYTHING is deduced from the Components type declaration !
// From this declaration, appropriate models will be expanded
// automatically.
//
// As a convention, the component H2O is always written first
// as it is the "main" component in a wWtp.
// next in line are solubles. followed by particulates

//BeginIllustration
//TYPE Components
//"
// The biological components considered in the WWTP models"
//"
//= ENUM {H2O, S_S, X_S, X_I};
//EndIllustration

OBJ NrOfComponents
"
 The number of biological components considered in the WWTP models
"
: Integer := Cardinality(Components);

// The WWTPTerminal class is a template for the
// quantities which will be passed around the system.
// As with the Component type declaration, this declaration
// may be given at the very end by the user. The appropriate
// model will then be expanded.
// Note however that, as long as we're only dealing with
// biological components flowing around the system (as
// declared in TYPE Components), the WWTPTerminal CLASS
// below is sufficient !
// The following assumptions are made:
// 1. The SAME (WWTPTerminal) terminals are used everywhere in
//      a configuration.
// 2. All terminals of a model have the same cardinality.
//      This is enforced thanks to the way we define Components and
//       WWTPTerminal.
// 3. The number of components in WWTPTerminal is the same as the number of
//      components (columns) in the stoichiometry matrix (which will be
//      defined later on). Again, this is enforced thanks to the way we define
//      Components and WWTPTerminal.
//      Note that components which are transported but do not react
//      (i.e., only hydraulics, no chemistry nor biology)
//      just have a column of zeroes in the stoichiometry matrix.
//      This is easy as by default, when a variable was not given a value,
//      the initial value is 0. Thus, if we don't assign anything to
//      elements of the stoichiometry matrix, it is a matrix of zeroes,
//      which means no chemical/biological reactions take place.

CLASS WWTPTerminal
"
 The variables which are passed between WWTP model building blocks
 Currently, we only consider a flux of biochemical material
"
= MassFlux[NrOfComponents;];


CLASS InWWTPTerminal SPECIALISES WWTPTerminal; //used to indicate inflow
CLASS OutWWTPTerminal SPECIALISES WWTPTerminal; //used to indicage outflow

CLASS WWTPConcTerminal
"
 The variables which are passed between WWTP model building blocks
 Currently, we only consider a flux of biochemical material
"
= Concentration[NrOfComponents;];


CLASS InWWTPConcTerminal SPECIALISES WWTPConcTerminal; //used to indicate inflow concentrations
CLASS OutWWTPConcTerminal SPECIALISES WWTPConcTerminal; //used to indicate outflow concentrations

// These classnames will be used by SelectByType() to determine which terminals
// are inflow and which are outflow. This is necessary to automatically
// generate the volume conservation law for any number (of inflow)
// terminals.


#ifdef ASM1

// some definitions to make the BOD COD transformer work

OBJ NrOfBODCODComponents
"The number of biological components considered in the input of a BODCOD transformer"
: Integer := Cardinality(BODCODComponents);

CLASS BODCODTerminal
"The parameters passed to a BOD COD transformer from the influent file"
```

```
 = MassFlux[NrOfBODCODComponents;];

 CLASS InBODCODTerminal SPECIALISES BODCODTerminal; //used to indicate inflow

OBJ NrOfBODComponents
 "The number of biological components considered in the input of a BODCOD transformer"
 : Integer := Cardinality(BODComponents);

 CLASS BODTerminal
 "The parameters passed to a BOD transformer from the influent file"
 = MassFlux[NrOfBODComponents;];

CLASS InBODTerminal SPECIALISES BODTerminal;

#endif // ASM1

//=================================================================
//=================================GLOBAL VARIABLES=================
//=================================================================

//
// Begin of ALL GLOBAL variables
//

 // Currently, iteration variables are not implicitly
 // declared as being of the type of the range nor are they
 // "expanded" as local (to the loop) variables.
 // Rather, they have to be declared explicitly.

 OBJ Comp_Index "Temporary iteration variable, index of the component" : Integer;
 OBJ Reaction_Index "Temporary iteration variable, index of the reaction" : Integer;
 OBJ In_Comp_Index "Temporary iteration variable, index of the incoming component" : Integer;
 OBJ Out_Comp_Index "Temporary iteration variable, index of the outgoing component" : Integer;
 OBJ Terminal "Temporary iteration variable" : WWTPTerminal;
 OBJ In_Terminal "Temporary iteration variable" : WWTPTerminal;
 OBJ Out_Terminal "Temporary iteration variable" : WWTPTerminal;
#ifdef ASM1
 OBJ In_BOD_COD_Terminal "Temporary iteration variable" : BODCODTerminal;
#endif // ASM1
 OBJ NrOfReactions
 "The number of reactions between biological components considered in the WWTP models"
 : Integer := Cardinality(Reactions);

 //
 // specific to the Takacs model
 //
 OBJ NrOfLayers "The number of layers in the secondary clarifier"
 : Integer := NR_OF_LAYERS;

 OBJ Layer_Index "Temporary iteration variable, index of the layer" : Integer;
 OBJ NrOfLayersButOne
  "Secondary clarifier number of layers - 1" :
  Integer := NR_OF_LAYERS_BUT_ONE;
 OBJ IndexOfFeedLayer
  "The index of the layer where the influent is fed to the clarifier" :
  Integer := INDEX_OF_FEED_LAYER;
 OBJ NrOfLayersPlusOne
  "Secondary clarifier number of layers + 1" :
  Integer := NR_OF_LAYERS_PLUS_ONE;

 //
 // end Takacs
 //

//
// End of ALL GLOBAL variables
//


//=================================================================
//=========================Begin of WWTPAtomicModel hierarchy=======
//=================================================================


//=================================================================
//=========================Begin of components & reactions def.=====
//=================================================================

 // The components definition.
 // Here, the user specifies which components will be considered.
 // This will define the size of the wwtp terminal (through NrOfComponents, see above)
 // Please note that the sequence of elements in the ComponentsVector
 // is of importance !!! This sequence is used in certain models to achieve
 // model re-usability.
 // The sequence (for this modelbase) has to be:
 // H2O
 // S_I S_S .. S_NH S_ALK
 // X_I X_S .. X_P X_ND
 // the safest place to add compenents is in the middle of every subset !!!

 TYPE Components
 "
  The biological components considered in the WWTP models
 "
#ifdef ASM1
  = ENUM {H2O, S_I, S_S, S_O, S_NO, S_ND, S_NH, S_ALK,
```

```
        X_I, X_S, X_BH, X_BA, X_P, X_ND};
#endif // ASM1

#if (defined ASM2 || defined ASM2d)
  = ENUM {H2O, S_I, S_O, S_N2, S_F, S_A, S_NO, S_PO, S_NH, S_ALK,
        X_I, X_S, X_H, X_PAO, X_PP, X_PHA, X_AUT, X_TSS, X_MEOH, X_MEP, X_ND};

 // NOTE: X_ND will always be zero
 // it is not mentioned in ASM2
 // but is here to take care of model re usability between ASM 1 and 2
#endif // ASM2 or ASM2d

 // Here, the user specifies which reactions between biological
 // components will be considered
 // As with Components this is
 // done as an enumerated type so it becomes possible
 // to refer to elements in the Stoichiometry and in the
 // Kinetics matrices by name rather than by number.

 TYPE Reactions
 "
  The reactions between biological components considered in the WWTP models
 "
#ifdef ASM1
 = ENUM {
        AerGrowthHetero,
        AnGrowthHetero,
        AerGrowthAuto,
        DecayOfHetero,
        DecayOfAuto,
        AmmonOfSolOrgN,
        HydrolOfEntrOrg,
        HydrolOfEntrOrgN,
        Aeration,
       };
#endif // ASM1

#ifdef ASM2
 = ENUM {
        AerHydrol,
        AnHydrol,
        AnaerHydrol,
        AerGrowthOnSf,
        AerGrowthOnSa,
        AnGrowthOnSfDenitrif,
        AnGrowthOnSaDenitrif,
        Fermentation,
        LysisOfHetero,
        StorageOfXPHA,
        StorageOfXPP,
        AerGrowthOnXPHA,
        LysisOfXPAO,
        LysisOfXPP,
        LysisOfXPHA,
        GrowthOfAuto,
        LysisOfAuto,
        Precipitation,
        Redissolution,
        Aeration,
       };
#endif // ASM2

#ifdef ASM2d
 = ENUM {
        AerHydrol,
        AnHydrol,
        AnaerHydrol,
        AerGrowthOnSf,
        AerGrowthOnSa,
        AnGrowthOnSfDenitrif,
        AnGrowthOnSaDenitrif,
        Fermentation,
        LysisOfHetero,
        StorageOfXPHA,
         AerStorageOfXPP,
        AerGrowthOnXPHA,
        LysisOfXPAO,
        LysisOfXPP,
        LysisOfXPHA,
        GrowthOfAuto,
        LysisOfAuto,
        Precipitation,
        Redissolution,
        AnStorageOfXPP,
         AnGrowthOnXPHADenitrif,
        Aeration,
       };
#endif // ASM2d

// Note that also the aeration process, a mass transport process,
// is considered to be a conversion process !!!

#ifdef ASM1

 TYPE BODCODComponents
 "The number of biological components considered in the input of a BODCOD transformer"
```

```
  = ENUM {H2O, COD, BOD5, TSS, TKN};

TYPE BODComponents
 "The number of biological components considered in the input of a BOD transformer"
  = ENUM {H2O, BOD5, TSS, TKN};

#endif // ASM1

//=============================================================
//=========================end of defining comp & react=========
//=============================================================

CLASS WWTPAtomicModel
"
 A generic atomic WWTP model.
 Only specifies mass balances (mass variation is
 sum of biological mass fluxes (bioflux, with incoming =
 positive sign, outgoing = negative sign) and a generic
 conversion term (only declared here. Has to be specified
 later).
"
SPECIALISES PhysicalDAEModelType :=
{:


 parameters <–
  {

// Due to the shape of the equations we use,
// it is more appropriate to work with Specific Volume =
// 1/Density (thus, we deal with specific volume = 0 rather than
// with density = infinity) than with density.

// The density (and hence specific volume) of different components
// seems to be global information (i.e., not model instance specific).
// There are however two reasons for NOT declaring
// WWTPSpecificVolume information as a global object.
//  1. WWTPSpecificVolume is a vector of size NrOfComponents.
//       Obvioulsy, filling in values in this vector can only
//       be done once we know which components are used.
//       Example: referring to WWTPSpecificVolume[S_S] if the
//       component S_S is not used is pointless.
//       Thus, it seems more reasonable to put WWTPSpecificVolume
//       in the parameter section of a (generic) model.
//  2. Once MSL-EXEC code is generated, the user
//       currently only has access (from the Experiment Environment)
//       to variables and parameters. Global variables (the logical
//       C equivalent of global MSL objects) are not accessible
//       (and currently not even generated for that matter).
//       We thus HAVE to put WWTPSpecificVolume with the parameters.
//       When it is put there, the user will be able to see(including
//       symbolic information) and even change (though that may not be needed)
//       Specific Volume data.
//       Later, it may be meaningful to include a global
//       constants/parameters section in MSL-EXEC.

    // We only declare WWTPSpecificVolume here.
    // Actual values will be given by the user in the equations of a model.
    // except for  WWTPSpecificVolume[H2O] := 0.000001
    // declared in the initial section
    OBJ WWTPSpecificVolume (* hidden = "1" *)
    "Vector containing the specific volume (= 1/density) for all the components"
    : SpecificVolumeVector;


    //
    // Indexing is done by means of the symbolic indices from the
    // enumerated type Components.
    //
    // WWTPSpecificVolume[H2O] := 0.000001;
    //
    // By default, if no explicit assignment is done, the value is zero.
    // Thus, with the assumption that density of H2O = 1 and all the
    // other densities are infinite, WWTPSpecificVolume[S_S] = 0;
    // etc. must not be written.

  };

 initial <–
  {
   parameters.WWTPSpecificVolume[H2O] := 0.000001;
  };

 independent <– { OBJ t "Time"  : Time; };

 state <–
  {
   OBJ M "Vector containing masses for all the components" : MassVector;
   OBJ FluxPerComponent (* hidden = "1" *)
   "Vector containing fluxes for all the components, the sum of all incoming and outgoing fluxes" : MassFluxVector;
   OBJ InFluxPerComponent (* hidden = "1" *)
    "Vector containing incoming fluxes for all the components": MassFluxVector;
   OBJ ConversionTermPerComponent (* hidden = "1" *)
    "Vector containing conversionterms for all the components": MassFluxVector;
   OBJ Q_In "Influent flow rate" : FlowRate ;
  };
```

```
 equations <–
  {

    // The FluxPerComponent is the sum of all
    // incoming (positive) and outgoing (negative) fluxes

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
      state.FluxPerComponent[Comp_Index] =

    // If not only WWTPTerminal type terminals are present in the interface
    // (e.g., also ControlTerminal), we have to select only
    // those terminals from the interface which are of
    // WWTPTerminal type (or any SUBtype such as InWWTPTerminal of it)
    // as those are the only ones for which the mass balance law holds.

    (SUMOVER In_Terminal IN {SelectByType(interface,InWWTPTerminal)}:
    In_Terminal[Comp_Index])+
    (SUMOVER Out_Terminal IN {SelectByType(interface,OutWWTPTerminal)}:
    Out_Terminal[Comp_Index]);};

    // The mass balance equations.
    // These are composed of a term due to incoming and
    // outgoing fluxes and of a term due to biochemical
    // interactions between components.

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
      DERIV(state.M[Comp_Index],[independent.t]) =
      state.FluxPerComponent[Comp_Index]
      +state.ConversionTermPerComponent[Comp_Index];};

    // for efficiency and because most models need it anyway
    // we calculate Q_In here

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
      state.InFluxPerComponent[Comp_Index] =
          SUMOVER In_Terminal IN {SelectByType(interface,InWWTPTerminal)}:
              (In_Terminal[Comp_Index]);
    };

    {state.Q_In = (parameters.WWTPSpecificVolume[H2O]
                  * state.InFluxPerComponent[H2O]);
    };

// Less general Q_In calculation to avoid algebraic loops in the
// modelling of WWTP's (Algebraic loops for S_I -> X_ND induced
// by Q_In !!!)

   };
 :};


//=====================================================================
//============================WWTPAtomicModelWithoutVolume=========
//=====================================================================

// BE CAREFUL
// IS NOT A SPECIALIZATION OF WWTPATOMICMODEL !!!
// FOR EFFICIENCY REASONS


 CLASS WWTPAtomicModelWithoutVolume
 SPECIALISES PhysicalDAEModelType :=
 {:
  parameters <–
   {
    OBJ WWTPSpecificVolume (* hidden = "1" *)
     "Vector containing the specific volume (= 1/density) for all the components"
     : SpecificVolumeVector;
   };

  initial <–
   {
    parameters.WWTPSpecificVolume[H2O] := 0.000001;
   };

  independent <– { OBJ t "Time" : Time; };

  state <–
   {
    OBJ InFluxPerComponent (* hidden = "1" *)
     "Vector containing incoming fluxes for all components" : MassFluxVector;
    OBJ Q_In "Influent flow rate" : FlowRate ;
   };

   equations <–
   {
     { FOREACH Comp_Index IN {1 .. NrOfComponents}:
        state.InFluxPerComponent[Comp_Index] =
         SUMOVER In_Terminal IN {SelectByType(interface,InWWTPTerminal)}:
         (In_Terminal[Comp_Index]);
     };

     {state.Q_In = (parameters.WWTPSpecificVolume[H2O]
                   * state.InFluxPerComponent[H2O]);
     };
```

```
  };
 :};
//=============================================================
//=========================clarifier===========================
//=============================================================

#include "wwtp.base.clarifier.msl"

//=============================================================
//=========================primary  clarifier==================
//=============================================================

// PointSettler
// PrimaryPointSettler
// OtterpohlAndFreundPrimary
// Tay
// LessardAndBeck

#include "wwtp.base.primary˙clarifier.msl"

//=============================================================
//=========================secondary  clarifier================
//=============================================================


// SecondaryPointSettler
// MarsiliLibelli
// OtterpohlAndFreundSecondary
// Takacs

#include "wwtp.base.secondary˙clarifier.msl"

//=============================================================
//=========================splitters===========================
//=============================================================

// RelTwoSplitter
// AbsTwoSplitter
// TwoCombiner
// RelThreeSplitter
// AbsThreeSplitter
// ThreeCombiner

#include "wwtp.base.splitters˙combiners.msl"

//=============================================================
//=============================WWTPAtomicModelWithVolume=========
//=============================================================

CLASS WWTPAtomicModelWithVolume EXTENDS WWTPAtomicModel WITH
{:

  state <–
   {
     OBJ V "Volume" : Volume;
     OBJ C "Vector containing concentrations for all the components" : ConcentrationVector;
   };

  equations <–
   {
    // volume and conc equations are calculated
    // specific to fixed or variable volume
   };

:};

#include "wwtp.base.buffertanks.msl"

//=============================================================
//=========================WWTPAtomicModelWithVariableVolume===
//=============================================================

CLASS WWTPAtomicModelWithVariableVolume
EXTENDS WWTPAtomicModelWithVolume WITH
{:
 interface <–
 {
   OBJ Inflow (* terminal = "in˙1" *) "Inflow" :
     InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Outflow (* terminal = "out˙1" *)"Outflow" :
     OutWWTPTerminal := {: causality <– "COUT" :};
 };

 parameters <–
 {
   OBJ N "Number of weirs on a tank" : PhysicalQuantityType :=
       {: value <– 100 ;
          interval <– {:lowerBound <– 0; upperBound <– PLUS_INF; :}
       :} ;
   OBJ A "Surface area of the tank" : Area := {: value <– 200; :} ;
   OBJ alfa "Parameter, function of the weir type or width"
       : PhysicalQuantityType := {: value <– 1 :};
   OBJ beta "Parameter, depends on the weir design"
```

```
        : PhysicalQuantityType := {: value <– 1 :};
    OBJ V_Const "Constant tank volume beneath the lowest point of the weir"
        : Volume := {: value <– 1900 :};
  };

  state <–
   {
    OBJ Q_Out "Effluent flow rate" : FlowRate ;
   };

  equations <–
   {
    // Q_Out is stated variable and declared as
    // Q_Out = N*alfa*(V/A^beta)
    // for an explanation of these parameters
    // see the parameter section above

    state.Q_Out = IF (state.V > parameters.V_Const)
      THEN parameters.N * parameters.alfa
           * pow((state.V – parameters.V_Const)/parameters.A, parameters.beta)
      ELSE 0;

    // The total volume is the sum of the volumes of each
    // of the components. The volume of each component
    // is determined by multiplying its mass by its
    // specific volume.

    state.V = SUMOVER Comp_Index IN {1 .. NrOfComponents}:
    (parameters.WWTPSpecificVolume[Comp_Index]*state.M[Comp_Index]);

    // The concentration of each component is just the mass
    // of that component divided by the total volume

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
     state.C[Comp_Index] = state.M[Comp_Index]/state.V;};

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
      interface.Outflow[Comp_Index] =
       – state.C[Comp_Index] * state.Q_Out ;};
   };
:};


// ConversionModel stands for all models where the ConversionTermPerComponent
//                                                T
// takes on the form of Stoichiometry * Kinetics * V
// For each component, the reaction term is the sum
// of products of corresponding (one for each reaction)
// factors from the Stochiometry and the Kinetics matrices.

CLASS VarVolumeConversionModel EXTENDS WWTPAtomicModelWithVariableVolume WITH
{:
#include "wwtp.VolumeConversionModel.body.msl"
:};

// Below is an ASMConversionModel
// Actually, depending on which Components are used
// only a small part of the IAWQ may be needed.
// The result of a reduced  declaration of Components,
// Stoichiometry and Kinetics also become smaller.
// Hence, we only have to (and only can !) refer to
// those Components in Stoichiometry and Kinetics.
// Hence, we will probably build different
// ASM1ConversionModels corresponding to components
// used.
// Perhaps more ELEGANT is the following (which relies
// heavily on symbolic elimination of empty (0=0) equations):
// If we define Components and Reactions so that they contain
// ALL components and reactions from IAWQ, then all appropriate
// equations will be generated automatically. This does require
// us to fill in Stoichiometry and Kinetics. If however
// we want a limited IAWQ model ASM1ConversionModelLimited we leave
// almost all elements in Stoichiometry and Kinetics undefined (and
// thus by default =0). This is equivalent to a limited IAWQ.
// Currently, there is a performance catch. Even though 0=0
// equations may be eliminated, the state vectors still have
// full dimension NrOfComponents. This implies a heavy burden on
// the integrator.
// Conclusion: for the time being, use previous solution.
// Later, not only eliminate 0=0 equations but also variables.
// We need some heuristic for this or perhaps user annotation
// tagging certain variables for removal. E.g, (* ignore_variable =
// TRUE *)

CLASS VarVolumeASMConversionModel EXTENDS VarVolumeConversionModel WITH
{:
#ifdef ASM1
#include "wwtp.VolumeASM1ConversionModel.body.msl"
#endif //ASM1

#ifdef ASM2
#include "wwtp.VolumeASM2ConversionModel.body.msl"
#endif //ASM2

#ifdef ASM2d
```

```
#include "wwtp.VolumeASM2dConversionModel.body.msl"
#endif //ASM2d
 :};

 // A model with one inflow and one outflow
 CLASS VarVolumeInOutIAWQ EXTENDS VarVolumeASMConversionModel WITH
 {:
   initial <-
   {
   };
 :};

 // Below is where we start putting user-specific information
 // This will later be put in a separate file

 // Add the Specific Volume (=1/density) information to the equations

 CLASS VarVolumeASU
 (* class = "activated`sludge`unit"; category = "" *)
 SPECIALISES VarVolumeInOutIAWQ :=
 {:
   comments <- "Model for an activated sludge unit with a variable volume";

   initial <-
   {
   };
 :};

//=============================================================================
//=========================WWTPAtomicModelVariablePumpedVolume========
//=============================================================================

CLASS WWTPAtomicModelWithPumpedVolume
 EXTENDS WWTPAtomicModelWithVolume WITH
 {:
  interface <-
   {
     OBJ Inflow (* terminal = "in`1" *) "Inflow" :
       InWWTPTerminal := {: causality <- "CIN" :};
     OBJ Outflow (* terminal = "out`1" *) "Outflow" :
       OutWWTPTerminal := {: causality <- "COUT" :};
   };

  parameters <-
   {
     OBJ Q_Pump "Desired effluent flow rate" : FlowRate ;
     OBJ V_Max "Maximum volume of the tank" : Volume ;
     OBJ V_Min "Minimum volume of the tank" : Volume ;
   };

  state <-
   {
     OBJ Q_Out "Actual effluent flow rate" : FlowRate ;
   };

  equations <-
   {
    state.V = SUMOVER Comp_Index IN {1 .. NrOfComponents}:
    (parameters.WWTPSpecificVolume[Comp_Index]*state.M[Comp_Index]);

    // The concentration of each component is just the mass
    // of that component divided by the total volume

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
     state.C[Comp_Index] = state.M[Comp_Index]/state.V;};

    state.Q_Out = IF (state.V < parameters.V_Min &&
                      parameters.Q_Pump > state.Q_In)
                  THEN state.Q_In
                  ELSE
                    IF (state.V < parameters.V_Max)
                    THEN parameters.Q_Pump
                    ELSE
                      IF (state.Q_In < parameters.Q_Pump)
                      THEN parameters.Q_Pump
                      ELSE state.Q_In ;

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
      interface.Outflow[Comp_Index] =
       - state.C[Comp_Index] * state.Q_Out ;};
   };
 :};

CLASS PumpedVolumeConversionModel EXTENDS WWTPAtomicModelWithPumpedVolume WITH
 {:
#include "wwtp.VolumeConversionModel.body.msl"
 :};

 CLASS PumpedVolumeASMConversionModel EXTENDS PumpedVolumeConversionModel WITH
 {:
#ifdef ASM1
#include "wwtp.VolumeASM1ConversionModel.body.msl"
#endif //ASM1

#ifdef ASM2
```

```
#include "wwtp.VolumeASM2ConversionModel.body.msl"
#endif //ASM2

#ifdef ASM2d
#include "wwtp.VolumeASM2dConversionModel.body.msl"
#endif //ASM2d
:};

// A model with one inflow and one outflow
CLASS PumpedVolumeInOutIAWQ EXTENDS PumpedVolumeASMConversionModel WITH
{:
  initial <-
  {
  };
:};


CLASS PumpedVolumeASU
(* class = "activated`sludge`unit"; category = "" *)
SPECIALISES PumpedVolumeInOutIAWQ :=
{:
  comments <- "Model for an activated sludge unit with a variable pumped volume";

  initial <-
  {
  };
:};




//=========================================================================
//=======================WWTPAtomicModelWithFixedVolume====================
//=========================================================================

CLASS WWTPAtomicModelWithFixedVolume EXTENDS WWTPAtomicModelWithVolume WITH
{:
 interface <-
  {
    OBJ Inflow (* terminal = "in`1" *) "Inflow" :
      InWWTPTerminal := {: causality <- "CIN" :};
    OBJ Outflow (* terminal = "out`1" *)"Outflow" :
      OutWWTPTerminal := {: causality <- "COUT" :};
  };

 state <-
  {
//      OBJ Q_Out "Effluent flow rate" : FlowRate ;
  };

 equations <-
  {
   // because of a fixed volume ...
   // state.Q_Out = state.Q_In; anyway
   // so skip it

    // The total volume is the sum of the volumes of each
   // of the components. The volume of each component
   // is determined by multiplying its mass by its
   // specific volume.

   state.V = SUMOVER Comp_Index IN {1 .. NrOfComponents}:
   (parameters.WWTPSpecificVolume[Comp_Index]*state.M[Comp_Index]);

   // The concentration of each component is just the mass
   // of that component divided by the total volume

   {FOREACH Comp_Index IN {1 .. NrOfComponents}:
    state.C[Comp_Index] = state.M[Comp_Index]/state.V;};

  {FOREACH Comp_Index IN {1 .. NrOfComponents}:
     interface.Outflow[Comp_Index] =
      - state.C[Comp_Index] * state.Q_In ;};
  };
:};


CLASS FixVolumeConversionModel EXTENDS WWTPAtomicModelWithFixedVolume WITH
{:
#include "wwtp.VolumeConversionModel.body.msl"
:};

CLASS FixVolumeASMConversionModel EXTENDS FixVolumeConversionModel WITH
{:
#ifdef ASM1
#include "wwtp.VolumeASM1ConversionModel.body.msl"
#endif //ASM1

#ifdef ASM2
#include "wwtp.VolumeASM2ConversionModel.body.msl"
#endif //ASM2

#ifdef ASM2d
#include "wwtp.VolumeASM2dConversionModel.body.msl"
#endif //ASM2d
```

```
:};

// A model with one inflow and one outflow
CLASS FixVolumeInOutIAWQ EXTENDS FixVolumeASMConversionModel WITH
{:
  initial <-
  {
  };
:};

// Below is where we start putting user-specific information
// This will later be put in a separate file

// Add the Specific Volume (=1/density) information to the equations
CLASS FixVolumeASU
(* class = "activated`sludge`unit"; category = "" *)
SPECIALISES FixVolumeInOutIAWQ :=
{:
  comments <- "Model for an activated sludge unit with a fixed volume";

  initial <-
  {
  };
:};


//=====================================================================
//=================End of WWTPAtomicModel hierarchy=====================
//=====================================================================


//
// End of WWTPAtomicModel hierarchy
//

//
// From this point on, non-WWTPAtomicModels are considered
// like Takacs model, sensors, controllers, ...
// they are mostly non-WWTP specializations of PhysicalDAEModelType
//


//
// Begin of non-WWTPAtomicModels
//

//=============================================================
//===================================sensors===================
//=============================================================

// FlowSensor
// DOSensor
// NH4Sensor
// NO3Sensor
// XSensor

#include "wwtp.base.sensors.msl"

//=============================================================
//=============================controllers=====================
//=============================================================

// P
// PI
// PID
// OnOff
// Backlash
// Saturation
// RateLimiter
// DeadZone
// CoulombFriction

#include "wwtp.base.controllers.msl"

//=============================================================
//================================timers=======================
//=============================================================

// timer21
// timer22
// timer31
// timer32
// timer41
// timer42
// timer51
// timer52

#include "wwtp.base.timers.msl"

//=============================================================
//======================convertors=============================
//=============================================================

// CtoF
// FtoC
```

```
// Waste

#include "wwtp.base.convertors.msl"

//=========================================================
//=======================transformer=======================
//=========================================================

// BOD/COD tranformer

#include "wwtp.base.transformers.msl"

//=========================================================
//=======================generators========================
//=========================================================

// sinus
// double sinus
// block

#include "wwtp.base.generators.msl"

//=========================================================
//===============loopbreaker models=======================
//=========================================================

#include "wwtp.base.loopbreaker.msl"

//=========================================================
//====================trickling filter=====================
//=========================================================

#include "wwtp.base.trickling˙filter.msl"

//=========================================================
//====================river models=========================
//=========================================================

// Bulk_Benthic_River
// BenthicRiver

// #include “river.base.msl”

//=========================================================
//================anaerobic digestion======================
//=========================================================

// anaerobic digestion models

//#include “wwtp.base.anaerobic_digestion.msl”

//=========================================================

#endif // WWTP_BASE
```

B.6 MSL-USER WWTP ASM Conversion Library

```
// Description: WWTP/VolumeConsersionModel body.

//#################################################################
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
// implementation: Frederik Decouttere
// topic: general stoichiometry, kinetics, dataflow
// contact: Jurgen Meirlaen
//#################################################################

// body for variable & fixed VolumeConversionModel

  interface <–
  {
    OBJ OUR_ASU (* terminal = "out˙2" *)"OUR measurement data" :
      OxygenUptakeRate := {: causality <– "COUT" :};
    OBJ Kla_ASU (* terminal = "out˙2" *)"Kla measurement data" :
      OxygenTransferCoefficient := {: causality <– "COUT" :};
    OBJ V_ASU (* terminal = "out˙2" *)"Volume measurement data" :
      Volume := {: causality <– "COUT" :};
  };

  parameters <–
  {
    OBJ Stoichiometry (* hidden = "1" *)
    "A matrix structure containing stoichiometry" : QuantityType[NrOfReactions;][NrOfComponents,];
  };

  state <–
  {
    OBJ Kinetics (* hidden = "1" *) "A vector containing kinetics for all reactions" : QuantityType[NrOfReactions;];
    OBJ OUR (* fixed = "1" *) "The OUR == oxygen uptake rate" : OxygenUptakeRate;
  };
```

```
equations <-
{
 {FOREACH Comp_Index IN {1 .. NrOfComponents}:
   state.ConversionTermPerComponent[Comp_Index] =
    SUMOVER Reaction_Index IN {1 .. NrOfReactions}:
     (parameters.Stoichiometry[Reaction_Index][Comp_Index]
     *state.Kinetics[Reaction_Index])
    *state.V;};

 state.OUR =
  - (state.ConversionTermPerComponent[S_O] / state.V)
  + parameters.Stoichiometry[Aeration][S_O] * state.Kinetics[Aeration];

 interface.OUR_ASU = state.OUR;
 interface.Kla_ASU = parameters.Kla;
 interface.V_ASU = state.V;
};
```

B.7 MSL-USER WWTP ASM1 Library

```
// Description: MSL-USER/WWTP/VolumeASM1ConversionModel body.

//#####################################################################
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
// implementation: Frederik Decouttere
// topic: ASM1 model
// contact: Jurgen Meirlaen
//#####################################################################

// body for variable & fixed VolumeASM1ConversionModel

 parameters <-
 {
   //Basic biological parameters

   OBJ Y_H       "Yield For Heterotrophic Biomass" : YieldForHeterotrophicBiomass :=
        {:
         value <- 0.67;
         interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF; :};
        :};
   OBJ i_X_B     "Mass Of Nitrogen Per Mass Of COD In Biomass"
    : MassOfNitrogenPerMassOfCODInBiomass := {:value <- 0.086:};
   OBJ Y_A       "Yield For Autotrophic Biomass"
    : YieldForAutotrophicBiomass := {:value <- 0.24:};
   OBJ f_P       "Fraction Of Biomass Converted To Inert Matter"
    : FractOfBiomassLeadingToPartProd := {:value<- 0.08:};
   OBJ i_X_P     "Mass Of Nitrogen Per Mass Of COD In Products Formed"
    : MassOfNitrogenPerMassOfCODInProdFromBiomass := {:value <- 0.06:};
   OBJ mu_H      "Maximum Specific Growth Rate For Heterotrophic Biomass"
    : MaxSpecifGrowthRateHetero := {:value <- 4.00:};
   OBJ mu_A      "Maximum Specific Growth Rate For Autotrophic Biomass"
    : MaxSpecifGrowthRateAutotr := {:value <- 0.55:};
   OBJ K_S       "Half-Saturation Coefficient For Heterotrophic Biomass"
    : HalfSatCoeffForHetero :={:value <- 20.00:};
   OBJ K_OH      "Oxygen Half-Saturation Coefficient For Heterotrophic Biomass"
    : OxygenHalfSatCoeffForHetero := {:value <- 0.2:};
   OBJ K_X       "Half Saturation Coefficient For Hydrolysis Of Slowly Biodegradable Substrate"
    : HalfSatCoeffForHydrolSlowBioDegradeSubstr := {:value <- 0.02:};
   OBJ b_H       "Decay Coefficient For Heterotrophic Biomass"
    : DecayCoeffHeterotr := {:value <- 0.40:};
   OBJ b_A       "Decay Coefficient For Autotrophic Biomass"
    : DecayCoeffAutotr := {:value <- 0.01:};
   OBJ n_h       "Correction Factor For Anoxic Hydrolysis"
    : CorrectionFactor := {:value <-  0.4:};
   OBJ n_g       "Correction Factor For Anoxic Growth Of Heteritrophs"
    : CorrectionFactor := {:value <- 0.8:};
   OBJ k_a       "Maximum Specific Ammonification Rate"
    : AmmonificationRate := {:value <- 0.06:};
   OBJ k_h       "Maximum Specific Hydrolysis Rate"
    : MaxSpecificHydrolysisRate :=  {:value <- 2.00:};
   OBJ K_NO      "Nitrate Half-Saturation Coefficient For Denitrifying Heterotrophic Biomass"
    : NitrateHalfSatCoeffForDenitrifHetero := {:value <- 0.50:};
   OBJ K_NH      "Ammonia Half-Saturation Coefficient For Autotrophic Biomass"
    : AmmonHalfSatCoeffForAutotr := {:value <- 1.00:};
   OBJ K_OA      "Oxygen Half-Saturation Coefficient For Autotrophic Biomass"
    : OxygenHalfSatCoeffForAutotr := {:value <- 0.4:};
   OBJ Kla       "Oxygen transfer coefficient"
    : OxygenTransferCoefficient := {:value <- 50:};
   OBJ S_O_Sat   "Oxygen saturation concentration"
    : Concentration := {:value <- 8:};

   OBJ F_COD_TSS "Fraction COD/TSS" : Fraction := {: value <- 0.75 :};
 };

 state <-
 {
   OBJ X_TSS "Total suspended solids" : Concentration;
 };
```

```
initial <-
{
  parameters.Stoichiometry[AerGrowthHetero][S_S]
   := - 1/(parameters.Y_H);
  parameters.Stoichiometry[AerGrowthHetero][X_BH]
   := 1;
  parameters.Stoichiometry[AerGrowthHetero][S_O]
   := - (1-parameters.Y_H)/parameters.Y_H;
  parameters.Stoichiometry[AerGrowthHetero][S_NH]
   := - parameters.i_X_B;
  parameters.Stoichiometry[AerGrowthHetero][S_ALK]
   := - parameters.i_X_B/14;
  parameters.Stoichiometry[AnGrowthHetero][S_S]
   := - 1/parameters.Y_H;
  parameters.Stoichiometry[AnGrowthHetero][X_BH]
   := 1;
  parameters.Stoichiometry[AnGrowthHetero][S_NO]
   := - (1-parameters.Y_H)/(2.86*parameters.Y_H);
  parameters.Stoichiometry[AnGrowthHetero][S_NH]
   := - parameters.i_X_B;
  parameters.Stoichiometry[AnGrowthHetero][S_ALK]
   := ((1-parameters.Y_H)/(14*2.86*parameters.Y_H))-(parameters.i_X_B/14);
  parameters.Stoichiometry[AerGrowthAuto][X_BA]
   := 1;
  parameters.Stoichiometry[AerGrowthAuto][S_O]
   := -(4.57-parameters.Y_A)/parameters.Y_A;
  parameters.Stoichiometry[AerGrowthAuto][S_NO]
   := 1/parameters.Y_A;
  parameters.Stoichiometry[AerGrowthAuto][S_NH]
   := -parameters.i_X_B-1/parameters.Y_A;
  parameters.Stoichiometry[AerGrowthAuto][S_ALK]
   := -(parameters.i_X_B/14)-(1/(7*parameters.Y_A));
  parameters.Stoichiometry[DecayOfHetero][X_S]
   := 1-parameters.f_P;
  parameters.Stoichiometry[DecayOfHetero][X_BH]
   := -1;
  parameters.Stoichiometry[DecayOfHetero][X_P]
   := parameters.f_P;
  parameters.Stoichiometry[DecayOfHetero][X_ND]
   := parameters.i_X_B-parameters.f_P*parameters.i_X_P;
  parameters.Stoichiometry[DecayOfAuto][X_S]
   := 1-parameters.f_P;
  parameters.Stoichiometry[DecayOfAuto][X_BA]
   := -1;
  parameters.Stoichiometry[DecayOfAuto][X_P]
   := parameters.f_P;
  parameters.Stoichiometry[DecayOfAuto][X_ND]
   := parameters.i_X_B-parameters.f_P*parameters.i_X_P;
  parameters.Stoichiometry[AmmonOfSolOrgN][S_NH]
   := 1;
  parameters.Stoichiometry[AmmonOfSolOrgN][S_ND]
   := -1;
  parameters.Stoichiometry[AmmonOfSolOrgN][S_ALK]
   := 1.0/14.0;
  parameters.Stoichiometry[HydrolOfEntrOrg][S_S]
   := 1;
  parameters.Stoichiometry[HydrolOfEntrOrg][X_S]
   := -1;
  parameters.Stoichiometry[HydrolOfEntrOrgN][S_ND]
   := 1;
  parameters.Stoichiometry[HydrolOfEntrOrgN][X_ND]
   := -1;
  parameters.Stoichiometry[Aeration][S_O]
   := 1;
};

equations <-
{
  // From here, we assume the
  // Specific Volumes to be known

  state.Kinetics[AerGrowthHetero]
   := parameters.mu_H*
      (state.C[S_S]/(parameters.K_S+state.C[S_S]))*
      (state.C[S_O]/(parameters.K_OH+state.C[S_O]))*
      state.C[X_BH];
  state.Kinetics[AnGrowthHetero]
   := parameters.mu_H*
      (state.C[S_S]/(parameters.K_S+state.C[S_S]))*
      (parameters.K_OH/(parameters.K_OH+state.C[S_O]))*
      (state.C[S_NO]/(parameters.K_NO+state.C[S_NO]))*
      parameters.n_g*state.C[X_BH];
  state.Kinetics[AerGrowthAuto]
   := parameters.mu_A*
      (state.C[S_NH]/(parameters.K_NH+state.C[S_NH]))*
      (state.C[S_O]/(parameters.K_OA+state.C[S_O]))*
      state.C[X_BA];
  state.Kinetics[DecayOfHetero]
   := parameters.b_H*state.C[X_BH];
  state.Kinetics[DecayOfAuto]
   := parameters.b_A*state.C[X_BA];
  state.Kinetics[AmmonOfSolOrgN]
   := parameters.k_a*state.C[S_ND]*state.C[X_BH];
  state.Kinetics[HydrolOfEntrOrg]
   := parameters.k_h*
      (state.C[X_S]/state.C[X_BH])/(parameters.K_X+
```

```
        (state.C[X_S]/state.C[X_BH]))*
        ((state.C[S_O]/(parameters.K_OH+state.C[S_O]))+
        parameters.n_h*(parameters.K_OH/(parameters.K_OH+state.C[S_O]))*
        (state.C[S_NO]/(parameters.K_NO+state.C[S_NO])))*
        state.C[X_BH];
    state.Kinetics[HydrolOfEntrOrgN]
     := (parameters.k_h*(state.C[X_S]/state.C[X_BH])/(parameters.K_X+
        (state.C[X_S]/state.C[X_BH]))*((state.C[S_O]/(parameters.K_OH+
        state.C[S_O]))+parameters.n_h*(parameters.K_OH/(parameters.K_OH+
        state.C[S_O]))*(state.C[S_NO]/(parameters.K_NO+state.C[S_NO])))*
        state.C[X_BH])*(state.C[X_ND]/state.C[X_S]);
    state.Kinetics[Aeration]
     := parameters.Kla*(parameters.S_O_Sat-state.C[S_O]);

    // calculate X_TSS ...
    state.X_TSS :=
        (state.C[X_BH] + state.C[X_BA] + state.C[X_I] + state.C[X_S] + state.C[X_P])/ parameters.F_COD_TSS;

  };
```

B.8 MSL-USER WWTP Clarifier Models Library

```
// Description: MSL-USER/WWTP/Base/Primary_clarifier definitions.

//#################################################################
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
// implementation: Frederik Decouttere, Henk Vanhooren
// topic: primary clarifiers
// contact: Henk Vanhooren
//#################################################################

#ifndef WWTP_BASE_CLARIFIER
#define WWTP_BASE_CLARIFIER


 CLASS PointSettler
 "Pointsettler"

// The modelling of a settler by means of a pointsettler is a large
// simplification of the actual process. The settler is only a phase
// separator, and has no real volume. Hence, the model does not take into
// account the retention time in the settler. It is not a dynamical model
// but only based on mass balances.
//
// The effluent particulate concentration is calculated as a fraction of the
// influent concentration to the settler. The central equation is : X_Out = f_ns * X_i
// To calculate the underflow
// concentration a mass balance over the settler is solved.

 EXTENDS WWTPAtomicModelWithoutVolume WITH
 {:
  interface <-
   {
    OBJ Inflow (* terminal = "in`1" *) "Inflow" :
        InWWTPTerminal := {: causality <- "CIN" :};
    OBJ Outflow (* terminal = "out`1" *) "Overflow" :
        OutWWTPTerminal := {: causality <- "COUT" :};
    OBJ Underflow (* terminal = "out`2" *) "Underflow" :
        OutWWTPTerminal := {: causality <- "COUT" :};
   };

  parameters <-
   {
    OBJ f_ns "Non-settleable fraction of suspended solids" :
        Fraction := {: value <- 0.005 :};
    OBJ Q_Under "Underflow rate" : FlowRate := {: value <- 10 :};
    OBJ F_COD_TSS "Fraction COD/TSS 0.75" :                    Fraction := {: value <- 0.75 :};
   };

  state <-
   {
    OBJ f_Out (* hidden = "1" *) "Fraction of the influent flux going to the overflow" : Fraction ;
    OBJ f_Under (* hidden = "1" *) "Fraction of the influent flux going to the underflow" : Fraction ;
    OBJ X_Out "Effluent suspended solids concentration" : Concentration ;
    OBJ X_Under "Underflow suspended solids concentration" : Concentration ;
   };

  equations <-
   {
     // The underflow rate is a parameter, so the effluent flow rate has to
     // be calculated as a state variable

     // Soluble components (including water itself) are split into the two
     // streams (effluent and underflow) according to the ratio between
     // the flow rates.

    state.f_Out := (state.Q_In - parameters.Q_Under) / state.Q_In ;
    state.f_Under := parameters.Q_Under/state.Q_In ;

    {FOREACH Comp_Index IN {H2O .. S_ALK}:
```

```
      interface.Outflow[Comp_Index] =
       – state.InFluxPerComponent[Comp_Index] * state.f_Out ;};

    interface.Underflow[H2O] = – parameters.Q_Under /
          parameters.WWTPSpecificVolume[H2O] ;

    {FOREACH Comp_Index IN {S_I .. S_ALK}:
     interface.Underflow[Comp_Index] =
     – state.InFluxPerComponent[Comp_Index]* state.f_Under ;};

     // Particulate components are split according to the non settleable
     // fraction f_ns.
     // X_Out = f_ns * X_i has to be transformed to an equation using fluxes !!
     // Outflow = Q_Out * X_Out = a flux
     //         = Q_Out * f_ns * X_i
     //         = Q_Out * f_ns * Inflow / Q_In
     //         = Inflow * Q_Out / Q_In * f_ns  ==> see equations below
     // Underflow = Inflow - Outflow


    {FOREACH Comp_Index IN {X_I .. X_ND}:
      interface.Outflow[Comp_Index] =
       – state.InFluxPerComponent[Comp_Index] * state.f_Out *
         parameters.f_ns ;};

    {FOREACH Comp_Index IN {X_I .. X_ND}:
     interface.Underflow[Comp_Index] = – (state.InFluxPerComponent[Comp_Index]
     + interface.Outflow[Comp_Index]);};


#ifdef ASM1
    {state.X_Out =
      (SUMOVER Comp_Index IN {X_I .. X_P}:
      (– interface.Outflow[Comp_Index])) / ((state.Q_In – parameters.Q_Under)*parameters.F_COD_TSS) ;};

    {state.X_Under =
      (SUMOVER Comp_Index IN {X_I .. X_P}:
      (– interface.Underflow[Comp_Index])) / (parameters.Q_Under * parameters.F_COD_TSS) ;};


#endif // ASM1

#if (defined ASM2 || defined ASM2d)
    state.X_Out = interface.Outflow[X_TSS] / (state.Q_In – parameters.Q_Under);
    state.X_Under = interface.Underflow[X_TSS] / (parameters.Q_Under);
#endif // ASM2 or ASM2d
   };
 :};

#endif // WWTP_BASE_CLARIFIER
```

B.9 MSL-USER WWTP Splitter and Combiners Library

```
// Description: MSL-USER/WWTP/Base/Splitters_combiners definitions.

//#################################################################
// Ghent University
// Department of Applied Mathematics, Biometrics and Process Control
// implementation: Frederik Decouttere, Henk Vanhooren
// topic: splitters, combiners
// contact: Henk Vanhooren
//#################################################################

#ifndef WWTP_BASE_SPLITTERS_COMBINERS
#define WWTP_BASE_SPLITTERS_COMBINERS
CLASS RelTwoSplitter
 (* class = "two`splitter"; category = "" *)
 "relative splitter"

// Dividing a flow in two fraction, based on the flow fraction parameter.

 EXTENDS WWTPAtomicModelWithoutVolume WITH
 {:
  comments <– "A model for a relative splitter into two flows";
  interface <–
   {
    OBJ Inflow (* terminal = "in`1" *) "Inflow" :
        InWWTPTerminal := {: causality <– "CIN" :};
    OBJ Outflow1 (* terminal = "out`1" *) "Outflow1" :
        OutWWTPTerminal := {: causality <– "COUT" :};
    OBJ Outflow2 (* terminal = "out`2" *) "Outflow2" :
        OutWWTPTerminal := {: causality <– "COUT" :};
   };

  parameters <–
   {
    OBJ f_Out2 "Fraction of the fluxes going to outflow2" : Fraction := {: value <– 0.9 :};
   };

  equations <–
   {
    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
```

```
    interface.Outflow1[Comp_Index] =
    – state.InFluxPerComponent[Comp_Index] * (1–parameters.f_Out2);};

  {FOREACH Comp_Index IN {1 .. NrOfComponents}:
    interface.Outflow2[Comp_Index] =
    – state.InFluxPerComponent[Comp_Index] *
    (parameters.f_Out2);};

 };

:};


CLASS AbsTwoSplitter
(* class = "two`splitter"; category = "" *)
"absolute two way splitter"

// Dividing a flow in two flows.
// Attention should be given to the possibility that in case of an
// absolute splitter the flows never go negative.
// Attention is given in the model

EXTENDS WWTPAtomicModelWithoutVolume WITH
{:
 comments <– "A model for an absolute splitter into two flows";
 interface <–
  {
   OBJ Inflow (* terminal = "in`1" *) "Inflow" :
       InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Outflow1 (* terminal = "out`1" *) "Outflow1" :
       OutWWTPTerminal := {: causality <– "COUT" :};
   OBJ Outflow2 (* terminal = "out`2" *) "Outflow2" :
       OutWWTPTerminal := {: causality <– "COUT" :};
  };
 parameters <–
  {
   OBJ Q_Out2 "Outflow2 rate" : FlowRate := {: value <– 50:};
  };

 state <–
  {
   OBJ Q_Out1 "Outflow1 rate" : FlowRate ;
   OBJ f_Out2 (* hidden = "1" *) "Fraction of the influent flux going to outflow2" : Fraction ;
   OBJ Q_Out2_Help (* hidden = "1" *) "Help variable for outflow2 rate" : FlowRate ;
  };

 equations <–
  {
   state.Q_Out1 = IF(parameters.Q_Out2 > state.Q_In)
                  THEN 0
                  ELSE state.Q_In – state.Q_Out2_Help ;

   state.Q_Out2_Help = IF(parameters.Q_Out2 > state.Q_In)
                       THEN state.Q_In
                       ELSE parameters.Q_Out2;

   state.f_Out2 = state.Q_Out2_Help / state.Q_In ;

   interface.Outflow1[H2O] = – state.Q_Out1
     / parameters.WWTPSpecificVolume[H2O] ;
   interface.Outflow2[H2O] = – state.Q_Out2_Help
     / parameters.WWTPSpecificVolume[H2O] ;

   {FOREACH Comp_Index IN {2 .. NrOfComponents}:
     interface.Outflow1[Comp_Index] =
     – state.InFluxPerComponent[Comp_Index] * (1 – state.f_Out2) ;};

   {FOREACH Comp_Index IN {2 .. NrOfComponents}:
     interface.Outflow2[Comp_Index] =
     – state.InFluxPerComponent[Comp_Index] * state.f_Out2 ;};

  };
:};

CLASS TwoCombiner
(* class = "two`combiner"; category = "" *)
"two combiner"
EXTENDS WWTPAtomicModelWithoutVolume WITH
{:
 comments <– "A model for a combiner of two flows";
 interface <–
  {
   OBJ Inflow1 (* terminal = "in`1" *) "Inflow1" :
       InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Inflow2 (* terminal = "in`2" *) "Inflow2" :
       InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Outflow (* terminal = "out`1" *) "Outflow" :
       OutWWTPTerminal := {: causality <– "COUT" :};
  };

 equations <–
  {
   {FOREACH Comp_Index IN {1 .. NrOfComponents}:
     interface.Outflow[Comp_Index] =
     – state.InFluxPerComponent[Comp_Index];};
  };
```

```
:};

CLASS RelThreeSplitter
(* class = "three˙splitter"; category = "" *)
"relative three splitter"

// Dividing a flow in three fractions, based on the flow fraction parameter.

EXTENDS WWTPAtomicModelWithoutVolume WITH
{:
 comments <– "A model for a relative splitter into three flows";
 interface <–
  {
   OBJ Inflow (* terminal = "in˙1" *) "Inflow" :
       InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Outflow1 (* terminal = "out˙1" *) "Outflow1" :
       OutWWTPTerminal := {: causality <– "COUT" :};
   OBJ Outflow2 (* terminal = "out˙2" *) "Outflow2" :
       OutWWTPTerminal := {: causality <– "COUT" :};
   OBJ Outflow3 (* terminal = "out˙3" *) "Outflow3" :
       OutWWTPTerminal := {: causality <– "COUT" :};
  };

 parameters <–
  {
   OBJ f_Out2 "Fraction of the fluxes going to outflow2" : Fraction := {: value <– 0.1 :};
   OBJ f_Out3 "Fraction of the fluxes going to outflow3" : Fraction := {: value <– 0.8 :};
  };

 equations <–
  {
    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
     interface.Outflow2[Comp_Index] =
     – state.InFluxPerComponent[Comp_Index] *
     (parameters.f_Out2);};

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
     interface.Outflow3[Comp_Index] =
     – state.InFluxPerComponent[Comp_Index] *
     (parameters.f_Out3);};

    {FOREACH Comp_Index IN {1 .. NrOfComponents}:
     interface.Outflow1[Comp_Index] =
   – state.InFluxPerComponent[Comp_Index] * (1 – (parameters.f_Out2 + parameters.f_Out3) );};
  };

:};


CLASS AbsThreeSplitter
(* class = "three˙splitter"; category = "" *)
"absolute three way splitter"

// Dividing a flow in three flows.
// Attention should be given to the possibility that in case of an
// absolute splitter the flows never go negative.

EXTENDS WWTPAtomicModelWithoutVolume WITH
{:
 comments <– "A model for an absolute splitter into three flows";
 interface <–
  {
   OBJ Inflow (* terminal = "in˙1" *) "Inflow" :
       InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Outflow1 (* terminal = "out˙1" *) "Outflow1" :
       OutWWTPTerminal := {: causality <– "COUT" :};
   OBJ Outflow2 (* terminal = "out˙2" *) "Outflow2" :
       OutWWTPTerminal := {: causality <– "COUT" :};
   OBJ Outflow3 (* terminal = "out˙3" *) "Outflow3" :
       OutWWTPTerminal := {: causality <– "COUT" :};
  };

 parameters <–
  {
   OBJ Q_Out2 "Outflow2 rate" : FlowRate := {: value <– 50:};
   OBJ Q_Out3 "Outflow3 rate" : FlowRate := {: value <– 50:};
  };

 state <–
  {
   OBJ Q_Out1 "Outflow1 rate" : FlowRate ;
   OBJ f_Out1 (* hidden = "1" *) "Fraction of the influent flux going to outflow1" : Fraction ;
   OBJ f_Out2 (* hidden = "1" *) "Fraction of the influent flux going to outflow2" : Fraction ;
   OBJ f_Out3 (* hidden = "1" *) "Fraction of the influent flux going to outflow3" : Fraction ;
   OBJ Q_Out2_Help (* hidden = "1" *) "Help variable for outflow2 rate" : FlowRate ;
   OBJ Q_Out3_Help (* hidden = "1" *) "Help variable for outflow3 rate" : FlowRate ;
  };

 equations <–
  {
//creating zero outflow is potentially dangerous because the Q_In in the next block is zero,
// so when something is divided by Q_In it gives NaN (luckily caught by the solver now) !!!

   state.Q_Out1 = state.Q_In – (state.Q_Out2_Help + state.Q_Out3_Help);

   state.Q_Out2_Help = IF( parameters.Q_Out2 > state.Q_In)
```

```
                    THEN state.Q_In
                    ELSE parameters.Q_Out2;

  state.Q_Out3_Help = IF( (parameters.Q_Out2+parameters.Q_Out3) > state.Q_In)
                    THEN state.Q_In – state.Q_Out2_Help
                    ELSE parameters.Q_Out3;

  state.f_Out1 = 1 – (state.f_Out2 + state.f_Out3) ;
  state.f_Out2 = state.Q_Out2_Help /state.Q_In ;
  state.f_Out3 = state.Q_Out3_Help / state.Q_In ;

  interface.Outflow1[H2O] = – state.Q_Out1
    / parameters.WWTPSpecificVolume[H2O] ;
  interface.Outflow2[H2O] = – state.Q_Out2_Help
    / parameters.WWTPSpecificVolume[H2O] ;
  interface.Outflow3[H2O] = – state.Q_Out3_Help
    / parameters.WWTPSpecificVolume[H2O] ;

  {FOREACH Comp_Index IN {1 .. NrOfComponents}:
    interface.Outflow1[Comp_Index] =
    – state.InFluxPerComponent[Comp_Index] * state.f_Out1 ;};

  {FOREACH Comp_Index IN {1 .. NrOfComponents}:
    interface.Outflow2[Comp_Index] =
    – state.InFluxPerComponent[Comp_Index] * state.f_Out2 ;};

  {FOREACH Comp_Index IN {1 .. NrOfComponents}:
    interface.Outflow3[Comp_Index] =
    – state.InFluxPerComponent[Comp_Index] * state.f_Out3 ;};
  };
:};

CLASS ThreeCombiner
(* class = "three`combiner"; category = "" *)
"three combiner"
EXTENDS WWTPAtomicModelWithoutVolume WITH
{:
 comments <– "A model for a combiner of three flows";
 interface <–
  {
   OBJ Inflow1 (* terminal = "in`1" *) "Inflow1" :
       InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Inflow2 (* terminal = "in`2" *) "Inflow2" :
       InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Inflow3 (* terminal = "in`3" *) "Inflow3" :
       InWWTPTerminal := {: causality <– "CIN" :};
   OBJ Outflow (* terminal = "out`1" *) "Outflow" :
       OutWWTPTerminal := {: causality <– "COUT" :};
  };

 equations <–
  {
   {FOREACH Comp_Index IN {1 .. NrOfComponents}:
    interface.Outflow[Comp_Index] =
    – state.InFluxPerComponent[Comp_Index];};
  };
:};

#endif // WWTP_BASE_SPLITTERS
```

C

WWTP PDE Simulation

C.1 Discretization result: continuous case, effluent overflow

```
OBJ Iiterator: Integer;
OBJ Kiterator: Integer;
CLASS pdeClass SPECIALISES PhysicalDAEModelType :=
{:
//
// PDE model before discretisation: cont.msl
//
// //
// // cont.msl
// //
// // Acceptance test for the MSL compiler:
// //  a sedimentation PDE model
// //  The Dirac deltas have been concretised by
// //  a square pulse of width 2*sigma
// //  This should NOT be seen as the width of the inlet
// //  which is abstracted as a point in this 1-D model
// //
// // HV 23/ 3/1999
// //
//
// OBJ pdeClass "acceptance test of PDE" :
//  SET Generic END_SET:=
// {
//
//  // The declarations
//  //
//
//  // constants
//  //
//  OBJ z_f "inlet position" : PhysicalQuantityType :=
//   {OBJ value: Real:=0.35,
//    OBJ variability : Variability:= constant},
//
//  OBJ sigma "half width of the Dirac delta" : PhysicalQuantityType :=
//   {OBJ value: Real:=0.25,
//    OBJ variability: Variability:= constant},
//
//  // 1-D independent variables
```

```
//  //
//  OBJ t "time"  : PhysicalQuantityType :=
//   {OBJ variability: Variability:= indep_time_var},
//  OBJ z "space" : PhysicalQuantityType :=
//   {OBJ variability: Variability:= indep_space_var,
//    OBJ DIM1: Integer := 999},
//
//  // inputs
//  //
//  OBJ Q_f "influent flow" : PhysicalQuantityType :=
//   {OBJ variability: Variability:= input_var},
//  OBJ X_f "influent concentration" : PhysicalQuantityType :=
//   {OBJ variability: Variability:= input_var},
//  OBJ Q_u "underflow" : PhysicalQuantityType :=
//   {OBJ variability: Variability:= input_var},
//
//  // outputs (just for testing, not needed)
//  //
//  OBJ Q_e "effluent flow" : PhysicalQuantityType :=
//   {OBJ variability: Variability:= output_var},
//
//  // physical parameters
//  //
//  OBJ A "cross section area" : PhysicalQuantityType :=
//   {OBJ value: Real:=700.0,
//    OBJ variability: Variability:= parameter},
//  OBJ L "height of settler"  : PhysicalQuantityType :=
//   {OBJ value: Real:=5.0,
//    OBJ variability: Variability:= constant},
//
//  OBJ D_0 "diffusion constant"  : PhysicalQuantityType :=
//   {OBJ value: Real:=0.165,
//    OBJ variability: Variability:= parameter},
//  OBJ n "Vesilind settling" : PhysicalQuantityType :=
//   {OBJ value: Real:=0.306,
//    OBJ variability: Variability:= parameter},
//  OBJ v_0 "Vesilind settling" : PhysicalQuantityType :=
//   {OBJ value: Real:=7.625,
//    OBJ variability: Variability:= parameter},
//
//  // dependent variables
//  //
//  OBJ X "concentration X(z,t)" : PhysicalQuantityType :=
//   {OBJ variability: Variability:= algode_var},
//
//  // The PDE (after function substitution and Dirac approximation)
//  // HV: function substitution is not necessary
//  //     when (future) we allow a SET of equations as
//  //     3rd argument of foreach()
//  //
//  // Equations and Boundary Conditions are given "over" a domain
//  // The domain may be a range or a single point
//  //
//  //  foreach(<variable id>, <domainrange>, <equation>)
//
//  // The equations
//  //
//  foreach(z, range(-(0),-(1.5)),
//   DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A )*DERIV(X, [z,])
//                    + D_0*DERIV(X, [z,z])),
//  foreach(z, range(-(1.5),-(2.0)),
//   DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A)*DERIV(X, [z,])
//                      + X_f*Q_f/(2.0*sigma*A)
```

```
//                     + D_0*DERIV(X, [z,z])),
//  foreach(z, range(-(2.0),+(5.0)),
//   DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) + Q_u/A)*DERIV(X, [z,])
//                    + D_0*DERIV(X, [z,z])),
//
//  // The Boundary Conditions
//  // convention in BCs only:
//  //  write C + X*... + DX*... + D2X*...
//  //  i.e., the X-term up front in a product
//  //
//  // Note how the factors of the X-terms may not
//  // contain X (except in the condition of and ifThenElse())
//  // If the factors do contain X, the matrix approach
//  // can no longer be used which results in a consideble
//  // slowdown, less accuracy, ... It is better to employ
//  // a suitable piecewise linearisation of the F(X)
//  // factor. (see the batch case example)
//  //
//  foreach(z, range(+(0),-(0)), DERIV(X, [z,]) = 0.0),
//  foreach(z, range(+(5),-(5)), DERIV(X, [z,]) = 0.0),
//
//  // Later: discretisation information here too ?
//  // Thus: EXTEND PDE model with different
//  // discretisation schemes
// }
// // End of cont.msl
//
// -------------------------------------------
// Discretisation file: cont.inf
//
// 3
// 0
// 1 0 0 1.5
// 1 0 0 2.0
// 2 0 0 5.0
//
// -------------------------------------------
// Discretised model
//
 interface <-
{ OBJ Q_f "influent flow" :PhysicalQuantityType := {:causality <- CIN:};
OBJ X_f "influent concentration" :PhysicalQuantityType := {:causality <- CIN:};
OBJ Q_u "underflow" :PhysicalQuantityType := {:causality <- CIN:};
OBJ Q_e "effluent flow" :PhysicalQuantityType := {:causality <- COUT:}}
;
 parameters <-
{ OBJ Qinv (* hidden = "1" *) "Qinv matrix = inv(Q)" :PhysicalQuantityType [4;][4,];
OBJ C (* hidden = "1" *) "C matrix of constants (not X dependent)" :PhysicalQuantityType [4;];
OBJ Q (* hidden = "1" *) "Q matrix of U coefficients" :PhysicalQuantityType [4;][4,];
OBJ P (* hidden = "1" *) "P matrix of W coefficients" :PhysicalQuantityType [4;][4,];
OBJ A1 (* hidden = "1" *) "collocation matrix A1" :PhysicalQuantityType [3;][3,]:= [
 [ {: value <- -3.0:}, {: value <- 4.0:}, {: value <- -1.0:}];
 [ {: value <- -1.0:}, {: value <- 0.0:}, {: value <- 1.0:}];
 [ {: value <- 1.0:}, {: value <- -4.0:}, {: value <- 3.0:}]
]
;
OBJ B1 (* hidden = "1" *) "collocation matrix B1" :PhysicalQuantityType [3;][3,]:= [
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}];
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}];
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}]
]
;
OBJ A2 (* hidden = "1" *) "collocation matrix A2" :PhysicalQuantityType [3;][3,]:= [
```

```
 [ {: value <- -3.0:}, {: value <- 4.0:}, {: value <- -1.0:}];
 [ {: value <- -1.0:}, {: value <- 0.0:}, {: value <- 1.0:}];
 [ {: value <- 1.0:}, {: value <- -4.0:}, {: value <- 3.0:}]
]
;
OBJ B2 (* hidden = "1" *) "collocation matrix B2" :PhysicalQuantityType [3;][3,]:= [
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}];
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}];
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}]
]
;
OBJ A3 (* hidden = "1" *) "collocation matrix A3" :PhysicalQuantityType [4;][4,]:= [
 [ {: value <- -7.0:}, {: value <- 8.196152422707:}, {: value <- -2.196152422707:},
  {: value <- 1.0:}];
 [ {: value <- -2.732050807569:}, {: value <- 1.732050807569:}, {: value <- 1.732050807569:},
  {: value <- -0.7320508075689:}];
 [ {: value <- 0.7320508075689:}, {: value <- -1.732050807569:}, {: value <- -1.732050807569:},
  {: value <- 2.732050807569:}];
 [ {: value <- -1.0:}, {: value <- 2.196152422707:}, {: value <- -8.196152422707:},
  {: value <- 7.0:}]
]
;
OBJ B3 (* hidden = "1" *) "collocation matrix B3" :PhysicalQuantityType [4;][4,]:= [
 [ {: value <- 24:}, {: value <- -37.17691453624:}, {: value <- 25.17691453624:},
  {: value <- -12.0:}];
 [ {: value <- 16.39230484541:}, {: value <- -24.0:}, {: value <- 12.0:},
  {: value <- -4.392304845413:}];
 [ {: value <- -4.392304845413:}, {: value <- 12:}, {: value <- -24.0:},
  {: value <- 16.39230484541:}];
 [ {: value <- -12.0:}, {: value <- 25.17691453624:}, {: value <- -37.17691453624:},
  {: value <- 24:}]
]
;
OBJ z "space" :PhysicalQuantityType [8,]:=  [ {: value <- 0.0:}, {: value <- 0.75:},
 {: value <- 1.5:}, {: value <- 1.75:}, {: value <- 2.0:}, {: value <- 2.633974596216:},
 {: value <- 4.366025403784:}, {: value <- 5.0:}];
OBJ z_f "inlet position" :PhysicalQuantityType := {: value <- 0.35:};
OBJ sigma "half width of the Dirac delta" :PhysicalQuantityType := {: value <- 0.25:};
OBJ A "cross section area" :PhysicalQuantityType := {: value <- 700.0:};
OBJ L "height of settler" :PhysicalQuantityType := {: value <- 5.0:};
OBJ D_0 "diffusion constant" :PhysicalQuantityType := {: value <- 0.165:};
OBJ n "Vesilind settling" :PhysicalQuantityType := {: value <- 0.306:};
OBJ v_0 "Vesilind settling" :PhysicalQuantityType := {: value <- 7.625:}}
;
 independent <-
{ OBJ t "time" :PhysicalQuantityType }
;
 state <-
{ OBJ R (* hidden = "1" *) "R matrix = P*W+C" :PhysicalQuantityType [4;];
OBJ U (* hidden = "1" *) "X(z,t) at Finite Element Boundaries" :PhysicalQuantityType [4,]:=
 [ {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}];
OBJ W "X(z,t) at Internal Collocation Points" :PhysicalQuantityType [4,]:=
 [ {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}];
OBJ X "concentration X(z,t)" :PhysicalQuantityType [8,]:=
 [ {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:},
  {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}]}
;
 initial <-
{ parameters.Q[4][3]:= ((1/3.0)*parameters.A3[4][1]);
parameters.Q[4][4]:= ((1/3.0)*parameters.A3[4][4]);
parameters.Q[1][1]:= ((1/1.5)*parameters.A1[1][1]);
parameters.Q[1][2]:= ((1/1.5)*parameters.A1[1][3]);
parameters.Q[2][1]:= ((1/1.5)*parameters.A1[3][1]);
```

```
parameters.Q[2][2]:= (-(((1/0.5)*parameters.A2[1][1])) + ((1/1.5)*parameters.A1[3][3]));
parameters.Q[2][3]:= -(((1/0.5)*parameters.A2[1][3]));
parameters.Q[3][2]:= ((1/0.5)*parameters.A2[3][1]);
parameters.Q[3][3]:= (-(((1/3.0)*parameters.A3[1][1])) + ((1/0.5)*parameters.A2[3][3]));
parameters.Q[3][4]:= -(((1/3.0)*parameters.A3[1][4]));
parameters.Qinv[1][1]:= invert_td(ref(parameters.Qinv[1][1]) , ref(parameters.Q[1][1]) , 4)
;
parameters.C[1][1]:= -(0.0);
parameters.C[4][1]:= -(0.0);
parameters.P[1][1]:= ((1/1.5)*parameters.A1[1][2]);
parameters.P[2][1]:= ((1/1.5)*parameters.A1[3][2]);
parameters.P[2][2]:= -(((1/0.5)*parameters.A2[1][2]));
parameters.P[3][2]:= ((1/0.5)*parameters.A2[3][2]);
parameters.P[3][3]:= -(((1/3.0)*parameters.A3[1][2]));
parameters.P[3][4]:= -(((1/3.0)*parameters.A3[1][3]));
parameters.P[4][3]:= ((1/3.0)*parameters.A3[4][2]);
parameters.P[4][4]:= ((1/3.0)*parameters.A3[4][3])}
;
 equations <-
{
{FOREACH Iiterator IN {1 .. 4}
:
  state.U[Iiterator]:= -((SUMOVER Kiterator IN {1 .. 4}
:(parameters.Qinv[Iiterator][Kiterator]*state.R[Kiterator]));}
;

{FOREACH Iiterator IN {1 .. 4}
:
  state.R[Iiterator]:= ((SUMOVER Kiterator IN {1 .. 4}
:(parameters.P[Iiterator][Kiterator]*state.W[Kiterator])) + parameters.C[Iiterator]);}
;
DERIV(state.W[4],[independent.t])
= ((-(((((1 - (parameters.n*state.W[4]))*parameters.v_0)*exp((-(parameters.n)*state.W[4]))
) + (interface.Q_u/parameters.A)))*((state.U[4]*((1/3.0)*parameters.A3[3][4]))
+ ((state.W[4]*((1/3.0)*parameters.A3[3][3])) + ((state.W[3]*((1/3.0)*parameters.A3[3][2]))
+ (state.U[3]*((1/3.0)*parameters.A3[3][1])))))))
+ (parameters.D_0*((state.U[4]*(((1/3.0)*(1/3.0))*parameters.B3[3][4]))
+ ((state.W[4]*(((1/3.0)*(1/3.0))*parameters.B3[3][3]))
+ ((state.W[3]*(((1/3.0)*(1/3.0))*parameters.B3[3][2]))
+ (state.U[3]*(((1/3.0)*(1/3.0))*parameters.B3[3][1])))))));
DERIV(state.W[3],[independent.t])
= ((-(((((1 - (parameters.n*state.W[3]))*parameters.v_0)*exp((-(parameters.n)*state.W[3]))
) + (interface.Q_u/parameters.A)))*((state.U[4]*((1/3.0)*parameters.A3[2][4]))
+ ((state.W[4]*((1/3.0)*parameters.A3[2][3])) + ((state.W[3]*((1/3.0)*parameters.A3[2][2]))
+ (state.U[3]*((1/3.0)*parameters.A3[2][1])))))))
+ (parameters.D_0*((state.U[4]*(((1/3.0)*(1/3.0))*parameters.B3[2][4]))
+ ((state.W[4]*(((1/3.0)*(1/3.0))*parameters.B3[2][3]))
+ ((state.W[3]*(((1/3.0)*(1/3.0))*parameters.B3[2][2]))
+ (state.U[3]*(((1/3.0)*(1/3.0))*parameters.B3[2][1])))))));
DERIV(state.W[2],[independent.t])
= (((-(((((1 - (parameters.n*state.W[2]))*parameters.v_0)*exp((-(parameters.n)*state.W[2]))
) + (interface.Q_u/parameters.A)))*((state.U[3]*((1/0.5)*parameters.A2[2][3]))
+ ((state.W[2]*((1/0.5)*parameters.A2[2][2])) + (state.U[2]*((1/0.5)*parameters.A2[2][1])))))
+ ((interface.X_f*interface.Q_f)/((2.0*parameters.sigma)*parameters.A)))
+ (parameters.D_0*((state.U[3]*(((1/0.5)*(1/0.5))*parameters.B2[2][3]))
+ ((state.W[2]*(((1/0.5)*(1/0.5))*parameters.B2[2][2]))
+ (state.U[2]*(((1/0.5)*(1/0.5))*parameters.B2[2][1])))))));
DERIV(state.W[1],[independent.t])
= ((-(((((1 - (parameters.n*state.W[1]))*parameters.v_0)*exp((-(parameters.n)*state.W[1]))
) + (interface.Q_u/parameters.A)))*((state.U[2]*((1/1.5)*parameters.A1[2][3]))
+ ((state.W[1]*((1/1.5)*parameters.A1[2][2])) + (state.U[1]*((1/1.5)*parameters.A1[2][1])))))
+ (parameters.D_0*((state.U[2]*(((1/1.5)*(1/1.5))*parameters.B1[2][3]))
+ ((state.W[1]*(((1/1.5)*(1/1.5))*parameters.B1[2][2]))
```

```
+ (state.U[1]*(((1/1.5)*(1/1.5))*parameters.B1[2][1])))));
state.X[1]:= state.U[1];
state.X[2]:= state.W[1];
state.X[3]:= state.U[2];
state.X[4]:= state.W[2];
state.X[5]:= state.U[3];
state.X[6]:= state.W[3];
state.X[7]:= state.W[4];
state.X[8]:= state.U[4]}
;
:};
OBJ pde "pde. ": pdeClass; // (* Mod: remove *)
```

C.2 Discretization result: continuous case, effluent pumped

```
OBJ Iiterator: Integer;
OBJ Kiterator: Integer;
CLASS pdeClass SPECIALISES PhysicalDAEModelType :=
{:
//
// PDE model before discretisation: cont_theta.msl
//
// //
// // PDE model before discretisation: cont_theta.msl
// //
// // //
// // // cont_theta.msl
// // //
// // // Acceptance test for the MSL compiler:
// // //  a sedimentation PDE model
// // //  The Dirac deltas have been concretised by
// // //  a square pulse of width 2*sigma
// // //  This should NOT be seen as the width of the inlet
// // //  which is abstracted as a point in this 1-D model
// // //
// // // HV 23/ 3/1999
// // //
//
//  OBJ pdeClass "acceptance test of PDE" :
//   SET Generic END_SET:=
//  {
// //
// //  // The declarations
// //  //
// //
// //  // constants
// //  //
//   OBJ z_f "inlet position" : PhysicalQuantityType :=
//    {OBJ value: Real:=0.35,
//     OBJ variability : Variability:= constant},
//
//   OBJ sigma "half width of the Dirac delta" : PhysicalQuantityType :=
//    {OBJ value: Real:=0.5,
//     OBJ variability: Variability:= constant},
//
//   // 1-D independent variables
//   //
//   OBJ t "time"  : PhysicalQuantityType :=
//    {OBJ variability: Variability:= indep_time_var},
//   OBJ z "space" : PhysicalQuantityType :=
//    {OBJ variability: Variability:= indep_space_var,
//     OBJ DIM1: Integer := 999},
```

```
//
// //  // inputs
// //  //
// // // OBJ Q_f "influent flow" : PhysicalQuantityType :=
// // //  {OBJ variability: Variability:= input_var},
// // // OBJ X_f "influent concentration" : PhysicalQuantityType :=
// // //  {OBJ variability: Variability:= input_var},
// // // OBJ Q_u "underflow" : PhysicalQuantityType :=
// // //  {OBJ variability: Variability:= input_var},
// //
// //  // outputs (just for testing, not needed)
// //  //
// // // OBJ Q_e "effluent flow" : PhysicalQuantityType :=
// // //  {OBJ variability: Variability:= output_var},
// //
// //  // physical parameters
// //  //
//   OBJ A "cross section area" : PhysicalQuantityType :=
//    {OBJ value: Real:= 700.0,
//     OBJ variability: Variability:= parameter},
//   OBJ L "height of settler"  : PhysicalQuantityType :=
//    {OBJ value: Real:=5.0,
//     OBJ variability: Variability:= constant},
//
//   OBJ D_0 "diffusion constant"  : PhysicalQuantityType :=
//    {OBJ value: Real:= 0.165,
//     OBJ variability: Variability:= parameter},
//   OBJ n "Vesilind settling" : PhysicalQuantityType :=
//    {OBJ value: Real:= 0.306,
//     OBJ variability: Variability:= parameter},
//   OBJ v_0 "Vesilind settling" : PhysicalQuantityType :=
//   {OBJ value: Real:= 7.625,
//     OBJ variability: Variability:= parameter},
//
//   OBJ Q_f "influent flow" : PhysicalQuantityType :=
//    {OBJ value: Real:= 872.0,
//    OBJ variability: Variability:= parameter },
//   OBJ X_f "influent concentration" : PhysicalQuantityType :=
//    {OBJ value: Real:= 4.0,
//    OBJ variability: Variability:= parameter},
//   OBJ Q_u "underflow" : PhysicalQuantityType :=
//    {OBJ value: Real:= 397.0,
//   OBJ variability: Variability:= parameter},
//
//   OBJ dQ_f "influent flow step" : PhysicalQuantityType :=
//   {OBJ value: Real:= 400.0,
//    OBJ variability: Variability:= parameter },
//   OBJ dX_f "influent concentration step" : PhysicalQuantityType :=
//    {OBJ value: Real:= 3.0,
//    OBJ variability: Variability:= parameter},
//   OBJ dQ_u "underflow step" : PhysicalQuantityType :=
//    {OBJ value: Real:= 100.0,
//    OBJ variability: Variability:= parameter},
//
//   OBJ tQ_f1 "influent flow step start time" : PhysicalQuantityType :=
//    {OBJ value: Real:= 3.0,
//   OBJ variability: Variability:= parameter },
//   OBJ tQ_f2 "influent flow step end time" : PhysicalQuantityType :=
//    {OBJ value: Real:= 6.0,
//    OBJ variability: Variability:= parameter },
//
//   OBJ tX_f1 "influent concentration step start time" : PhysicalQuantityType :=
//    {OBJ value: Real:= 2.0,
```

```
//    OBJ variability: Variability:= parameter},
//   OBJ tX_f2 "influent concentration step end time" : PhysicalQuantityType :=
//    {OBJ value: Real:= 5.0,
//    OBJ variability: Variability:= parameter},
//
//   OBJ tQ_u1 "underflow step start time" : PhysicalQuantityType :=
//    {OBJ value: Real:= 10.0,
//    OBJ variability: Variability:= parameter},
//   OBJ tQ_u2 "underflow step end time" : PhysicalQuantityType :=
//    {OBJ value: Real:= 12.0,
//    OBJ variability: Variability:= parameter},
//
// // dependent variables
// //  //
//   OBJ X "concentration X(z,t)" : PhysicalQuantityType :=
//    {OBJ variability: Variability:= algode_var},
//
// // The PDE (after function substitution and Dirac approximation)
// //  // HV: function substitution is not necessary
// //  //     when (future) we allow a SET of equations as
// //  //     3rd argument of foreach()
// //  //
// //  // Equations and Boundary Conditions are given "over" a domain
// //  // The domain may be a range or a single point
// //  //
// //  //  foreach(<variable id>, <domainrange>, <equation>)
// //
// //  // The equations
// //  //
//   foreach(z, range(-(0),-(1.5)),
//    DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) +
//                       Q_u/A + dQ_u/A*(theta(t-tQ_u1) - theta(t-tQ_u2)))
//                       *DERIV(X, [z,]) + D_0*DERIV(X, [z,z])),
//   foreach(z, range(-(1.5),-(2.0)),
//    DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) +
//                      Q_u/A + dQ_u/A*(theta(t-tQ_u1) - theta(t-tQ_u2)))
//                       *DERIV(X, [z,])
//                        + (X_f+dX_f*(theta(t-tX_f1)-theta(t-tX_f2)))
//                        * (Q_f+dQ_f*(theta(t-tQ_f1)-theta(t-tQ_f2)))
//                        /(2.0*sigma*A) + D_0*DERIV(X, [z,z])),
//   foreach(z, range(-(2.0),+(5.0)),
//    DERIV(X, [t,]) = - ((1-n*X)*v_0*exp(-n*X) +
//                      Q_u/A + dQ_u/A*(theta(t-tQ_u1) - theta(t-tQ_u2)))
//                       *DERIV(X, [z,])
//                      + D_0*DERIV(X, [z,z])),
// //
// //  // The Boundary Conditions
// //  // convention in BCs only:
// //  //  write C + X*... + DX*... + D2X*...
// //  //  i.e., the X-term up front in a product
// //  //
// //  // Note how the factors of the X-terms may not
// //  // contain X (except in the condition of and ifThenElse())
// //  // If the factors do contain X, the matrix approach
// //  // can no longer be used which results in a consideble
// //  // slowdown, less accuracy, ... It is better to employ
// //  // a suitable piecewise linearisation of the F(X)
// //  // factor. (see the batch case example)
// //  //
//   foreach(z, range(+(0),-(0)), DERIV(X, [z,]) = 0.0),
//   foreach(z, range(+(5),-(5)), DERIV(X, [z,]) = 0.0),
// //
// //  // Later: discretisation information here too ?
```

```
// //  // Thus: EXTEND PDE model with different
// //  // discretisation schemes
//  }
// // // End of cont_theta.msl
//
// -------------------------------------------
// Discretisation file: cont_theta.inf
//
// 3
// 0
// 1 0 0 1.5
// 1 0 0 2.0
// 2 0 0 5.0
//
// -------------------------------------------
// Discretised model
//
 interface <-
{ }
;
 parameters <-
{ OBJ Qinv (* hidden = "1" *) "Qinv matrix = inv(Q)" :PhysicalQuantityType [4;][4,];
OBJ C (* hidden = "1" *) "C matrix of constants (not X dependent)" :PhysicalQuantityType [4;];
OBJ Q (* hidden = "1" *) "Q matrix of U coefficients" :PhysicalQuantityType [4;][4,];
OBJ P (* hidden = "1" *) "P matrix of W coefficients" :PhysicalQuantityType [4;][4,];
OBJ A1 (* hidden = "1" *) "collocation matrix A1" :PhysicalQuantityType [3;][3,]:= [
 [ {: value <- -3.0:}, {: value <- 4.0:}, {: value <- -1.0:}];
 [ {: value <- -1.0:}, {: value <- 0.0:}, {: value <- 1.0:}];
 [ {: value <- 1.0:}, {: value <- -4.0:}, {: value <- 3.0:}]
]
;
OBJ B1 (* hidden = "1" *) "collocation matrix B1" :PhysicalQuantityType [3;][3,]:= [
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}];
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}];
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}]
]
;
OBJ A2 (* hidden = "1" *) "collocation matrix A2" :PhysicalQuantityType [3;][3,]:= [
 [ {: value <- -3.0:}, {: value <- 4.0:}, {: value <- -1.0:}];
 [ {: value <- -1.0:}, {: value <- 0.0:}, {: value <- 1.0:}];
 [ {: value <- 1.0:}, {: value <- -4.0:}, {: value <- 3.0:}]
]
;
OBJ B2 (* hidden = "1" *) "collocation matrix B2" :PhysicalQuantityType [3;][3,]:= [
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}];
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}];
 [ {: value <- 4.0:}, {: value <- -8.0:}, {: value <- 4.0:}]
]
;
OBJ A3 (* hidden = "1" *) "collocation matrix A3" :PhysicalQuantityType [4;][4,]:= [
 [ {: value <- -7.0:}, {: value <- 8.196152422707:}, {: value <- -2.196152422707:},
  {: value <- 1.0:}];
 [ {: value <- -2.732050807569:}, {: value <- 1.732050807569:}, {: value <- 1.732050807569:},
  {: value <- -0.7320508075689:}];
 [ {: value <- 0.7320508075689:}, {: value <- -1.732050807569:}, {: value <- -1.732050807569:},
  {: value <- 2.732050807569:}];
 [ {: value <- -1.0:}, {: value <- 2.196152422707:}, {: value <- -8.196152422707:},
  {: value <- 7.0:}]
]
;
OBJ B3 (* hidden = "1" *) "collocation matrix B3" :PhysicalQuantityType [4;][4,]:= [
 [ {: value <- 24:}, {: value <- -37.17691453624:}, {: value <- 25.17691453624:},
  {: value <- -12.0:}];
```

```
 [ {: value <- 16.39230484541:}, {: value <- -24.0:}, {: value <- 12.0:},
  {: value <- -4.392304845413:}];
 [ {: value <- -4.392304845413:}, {: value <- 12:}, {: value <- -24.0:},
  {: value <- 16.39230484541:}];
 [ {: value <- -12.0:}, {: value <- 25.17691453624:}, {: value <- -37.17691453624:},
  {: value <- 24:}]
]
;
OBJ z "space" :PhysicalQuantityType [8,]:=  [ {: value <- 0.0:}, {: value <- 0.75:},
 {: value <- 1.5:}, {: value <- 1.75:}, {: value <- 2.0:}, {: value <- 2.633974596216:},
 {: value <- 4.366025403784:}, {: value <- 5.0:}];
OBJ z_f "inlet position" :PhysicalQuantityType := {: value <- 0.35:};
OBJ sigma "half width of the Dirac delta" :PhysicalQuantityType := {: value <- 0.5:};
OBJ A "cross section area" :PhysicalQuantityType := {: value <- 700.0:};
OBJ L "height of settler" :PhysicalQuantityType := {: value <- 5.0:};
OBJ D_0 "diffusion constant" :PhysicalQuantityType := {: value <- 0.165:};
OBJ n "Vesilind settling" :PhysicalQuantityType := {: value <- 0.306:};
OBJ v_0 "Vesilind settling" :PhysicalQuantityType := {: value <- 7.625:};
OBJ Q_f "influent flow" :PhysicalQuantityType := {: value <- 872.0:};
OBJ X_f "influent concentration" :PhysicalQuantityType := {: value <- 4.0:};
OBJ Q_u "underflow" :PhysicalQuantityType := {: value <- 397.0:};
OBJ dQ_f "influent flow step" :PhysicalQuantityType := {: value <- 400.0:};
OBJ dX_f "influent concentration step" :PhysicalQuantityType := {: value <- 3.0:};
OBJ dQ_u "underflow step" :PhysicalQuantityType := {: value <- 100.0:};
OBJ tQ_f1 "influent flow step start time" :PhysicalQuantityType := {: value <- 3.0:};
OBJ tQ_f2 "influent flow step end time" :PhysicalQuantityType := {: value <- 6.0:};
OBJ tX_f1 "influent concentration step start time" :PhysicalQuantityType := {: value <- 2.0:};
OBJ tX_f2 "influent concentration step end time" :PhysicalQuantityType := {: value <- 5.0:};
OBJ tQ_u1 "underflow step start time" :PhysicalQuantityType := {: value <- 10.0:};
OBJ tQ_u2 "underflow step end time" :PhysicalQuantityType := {: value <- 12.0:}}
;
 independent <-
{ OBJ t "time" :PhysicalQuantityType }
;
 state <-
{ OBJ R (* hidden = "1" *) "R matrix = P*W+C" :PhysicalQuantityType [4;];
OBJ U (* hidden = "1" *) "X(z,t) at Finite Element Boundaries" :PhysicalQuantityType [4,]:=
 [ {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}];
OBJ W "X(z,t) at Internal Collocation Points" :PhysicalQuantityType [4,]:=
 [ {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}];
OBJ X "concentration X(z,t)" :PhysicalQuantityType [8,]:=
 [ {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:},
  {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}, {: value <- 1.0:}]}
;
 initial <-
{ parameters.Q[4][3]:= ((1/3.0)*parameters.A3[4][1]);
parameters.Q[4][4]:= ((1/3.0)*parameters.A3[4][4]);
parameters.Q[1][1]:= ((1/1.5)*parameters.A1[1][1]);
parameters.Q[1][2]:= ((1/1.5)*parameters.A1[1][3]);
parameters.Q[2][1]:= ((1/1.5)*parameters.A1[3][1]);
parameters.Q[2][2]:= (-(((1/0.5)*parameters.A2[1][1])) + ((1/1.5)*parameters.A1[3][3]));
parameters.Q[2][3]:= -(((1/0.5)*parameters.A2[1][3]));
parameters.Q[3][2]:= ((1/0.5)*parameters.A2[3][1]);
parameters.Q[3][3]:= (-(((1/3.0)*parameters.A3[1][1])) + ((1/0.5)*parameters.A2[3][3]));
parameters.Q[3][4]:= -(((1/3.0)*parameters.A3[1][4]));
parameters.Qinv[1][1]:= invert_td(ref(parameters.Qinv[1][1]) , ref(parameters.Q[1][1]) , 4)
;
parameters.C[1][1]:= -(0.0);
parameters.C[4][1]:= -(0.0);
parameters.P[1][1]:= ((1/1.5)*parameters.A1[1][2]);
parameters.P[2][1]:= ((1/1.5)*parameters.A1[3][2]);
parameters.P[2][2]:= -(((1/0.5)*parameters.A2[1][2]));
parameters.P[3][2]:= ((1/0.5)*parameters.A2[3][2]);
```

```
parameters.P[3][3]:= -(((1/3.0)*parameters.A3[1][2]));
parameters.P[3][4]:= -(((1/3.0)*parameters.A3[1][3]));
parameters.P[4][3]:= ((1/3.0)*parameters.A3[4][2]);
parameters.P[4][4]:= ((1/3.0)*parameters.A3[4][3])}
;
 equations <-
{
{FOREACH Iiterator IN {1 .. 4}
:
  state.U[Iiterator]:= -((SUMOVER Kiterator IN {1 .. 4}
:(parameters.Qinv[Iiterator][Kiterator]*state.R[Kiterator]));}
;

{FOREACH Iiterator IN {1 .. 4}
:
  state.R[Iiterator]:= ((SUMOVER Kiterator IN {1 .. 4}
:(parameters.P[Iiterator][Kiterator]*state.W[Kiterator])) + parameters.C[Iiterator]);}
;
DERIV(state.W[4],[independent.t])
= ((-((((((1 - (parameters.n*state.W[4]))*parameters.v_0)*exp((-(parameters.n)*state.W[4]))
) + (parameters.Q_u/parameters.A)) + ((parameters.dQ_u/parameters.A)*(theta((independent.t
- parameters.tQ_u1)) - theta((independent.t - parameters.tQ_u2))
))))*((state.U[4]*((1/3.0)*parameters.A3[3][4])) + ((state.W[4]*((1/3.0)*parameters.A3[3][3]))
+ ((state.W[3]*((1/3.0)*parameters.A3[3][2])) + (state.U[3]*((1/3.0)*parameters.A3[3][1])))))))
+ (parameters.D_0*((state.U[4]*(((1/3.0)*(1/3.0))*parameters.B3[3][4]))
+ ((state.W[4]*(((1/3.0)*(1/3.0))*parameters.B3[3][3]))
+ ((state.W[3]*(((1/3.0)*(1/3.0))*parameters.B3[3][2]))
+ (state.U[3]*(((1/3.0)*(1/3.0))*parameters.B3[3][1])))))));
DERIV(state.W[3],[independent.t])
= ((-((((((1 - (parameters.n*state.W[3]))*parameters.v_0)*exp((-(parameters.n)*state.W[3]))
) + (parameters.Q_u/parameters.A)) + ((parameters.dQ_u/parameters.A)*(theta((independent.t
- parameters.tQ_u1)) - theta((independent.t - parameters.tQ_u2))
))))*((state.U[4]*((1/3.0)*parameters.A3[2][4])) + ((state.W[4]*((1/3.0)*parameters.A3[2][3]))
+ ((state.W[3]*((1/3.0)*parameters.A3[2][2])) + (state.U[3]*((1/3.0)*parameters.A3[2][1])))))))
+ (parameters.D_0*((state.U[4]*(((1/3.0)*(1/3.0))*parameters.B3[2][4]))
+ ((state.W[4]*(((1/3.0)*(1/3.0))*parameters.B3[2][3]))
+ ((state.W[3]*(((1/3.0)*(1/3.0))*parameters.B3[2][2]))
+ (state.U[3]*(((1/3.0)*(1/3.0))*parameters.B3[2][1])))))));
DERIV(state.W[2],[independent.t])
= (((-((((((1 - (parameters.n*state.W[2]))*parameters.v_0)*exp((-(parameters.n)*state.W[2]))
) + (parameters.Q_u/parameters.A)) + ((parameters.dQ_u/parameters.A)*(theta((independent.t
- parameters.tQ_u1)) - theta((independent.t - parameters.tQ_u2))
))))*((state.U[3]*((1/0.5)*parameters.A2[2][3])) + ((state.W[2]*((1/0.5)*parameters.A2[2][2]))
+ (state.U[2]*((1/0.5)*parameters.A2[2][1])))))
+ (((parameters.X_f + (parameters.dX_f*(theta((independent.t - parameters.tX_f1))
 - theta((independent.t - parameters.tX_f2))
)))*(parameters.Q_f + (parameters.dQ_f*(theta((independent.t - parameters.tQ_f1))
 - theta((independent.t - parameters.tQ_f2))
))))/((2.0*parameters.sigma)*parameters.A)))
+ (parameters.D_0*((state.U[3]*(((1/0.5)*(1/0.5))*parameters.B2[2][3]))
+ ((state.W[2]*(((1/0.5)*(1/0.5))*parameters.B2[2][2]))
+ (state.U[2]*(((1/0.5)*(1/0.5))*parameters.B2[2][1])))))));
DERIV(state.W[1],[independent.t])
= ((-((((((1 - (parameters.n*state.W[1]))*parameters.v_0)*exp((-(parameters.n)*state.W[1]))
) + (parameters.Q_u/parameters.A))
+ ((parameters.dQ_u/parameters.A)*(theta((independent.t - parameters.tQ_u1))
 - theta((independent.t - parameters.tQ_u2))
))))*((state.U[2]*((1/1.5)*parameters.A1[2][3]))
+ ((state.W[1]*((1/1.5)*parameters.A1[2][2])) + (state.U[1]*((1/1.5)*parameters.A1[2][1]))))))
+ (parameters.D_0*((state.U[2]*(((1/1.5)*(1/1.5))*parameters.B1[2][3]))
+ ((state.W[1]*(((1/1.5)*(1/1.5))*parameters.B1[2][2]))
+ (state.U[1]*(((1/1.5)*(1/1.5))*parameters.B1[2][1])))))));
state.X[1]:= state.U[1];
```

```
state.X[2]:= state.W[1];
state.X[3]:= state.U[2];
state.X[4]:= state.W[2];
state.X[5]:= state.U[3];
state.X[6]:= state.W[3];
state.X[7]:= state.W[4];
state.X[8]:= state.U[4]}
;
:};
OBJ pde "pde. ": pdeClass; // (* Mod: remove *)
```

C.3 Simulation results: continuous case

In addition to the values above, the initial concentrations at the four interior collocation points, $\{w_1^0, w_2^0, w_3^0, w_4^0\}$, have to be specified. Three different initial concentration profiles were considered: a uniform concentration profile, with $(w_1^0 = w_2^0 = w_3^0 = w_4^0)$, and with actual values $(9,9,9,9)kg/m^3$; a positive concentration gradient upward, with $(w_1^0 > w_2^0 > w_3^0 > w_4^0)$, with values $(11,8,6,5)kg/m^3$; and a positive concentration gradient downward, corresponding to $(w_1^0 < w_2^0 < w_3^0 < w_4^0)$, with values $(5,6,8,11)kg/m^3$.

One set of simulations (Set I) was performed with the Watts default parameters above. In addition, different *limits* of the PDEs were examined in order to verify that the implementation gives physically meaningful results. These other sets of simulations can be treated as further test cases for the implementation of PDEs within WEST++ using orthogonal collocation.

These different limits were:

- Set II: pure diffusion, with $D_0 = 1$, and $v_0 = Q_f = Q_u = 0$ (in appropriate units);
- Set III: diffusion with the source present, with $D_0 = 1$, $Q_f = 100$, $v_0 = Q_u = 0$;
- Set IV: only gravitational settling, with $D_0 = Q_f = Q_u = 0$, and with the Watts values for the Vesilind parameters;
- Set V: settling velocity with diffusion, with $Q_f = Q_u = 0$, and D_0, v_0, n with the Watts default values.

For all sets, the three different initial concentration profiles described above were used.

Simulations corresponding to Sets I and III above were performed for both cases of effluent overflow and effluent pumping, where their difference would be apparent. The other sets would yield the same results for both, and therefore have been performed for the case with effluent overflow only.

In the following graphs, the concentrations at the nodes and interior collocation points, $X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8$, are plotted as a function of time. That is, these concentrations are along the $y-$ axis, with time along the $x-$ axis. Units are already mentioned above.

C.3.1 Effluent overflow

1. Set I: Full PDE:

The three figures below show that there is indeed some settling achieved, for all three initial concentration profiles. These results are only qualitatively correct, since the asymptotic concentration values are all within an order of magnitude of the initial concentration values. (The asymptotic concentrations are actually in the order $(X_1 < X_2 < X_3 < X_4 < X_5)$, with $(X_5 = X_6 = X_7 = X_8)$, for the cases of uniform concentration, and a downward positive concentration gradient).

2. Set II: Pure Diffusion:

The three figures below show the familiar results expected from the simple diffusion equation. Notice that when there is no initial gradient, the concentration does not evolve in time. When there is an initial gradient, however, diffusion ensures that a uniform concentration is achieved across the clarifier, corresponding to the average initial concentration.

3. Set III: Diffusion with source present:

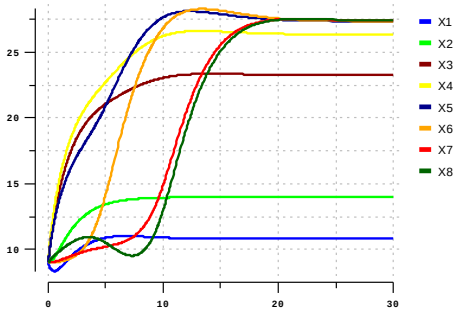

Figure C.1: Effluent overflow: Full PDE: Set I: uniform concentration

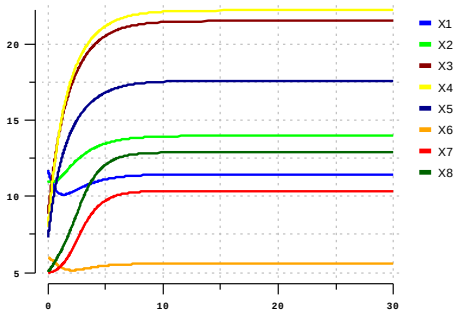

Figure C.2: Effluent overflow: Full PDE: Set I: positive concentration gradient upward

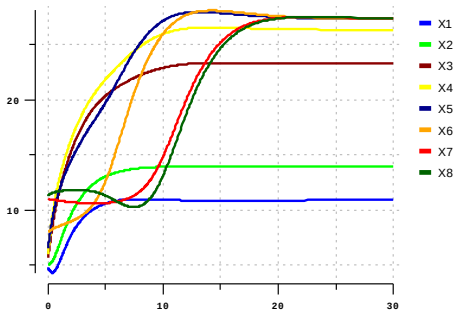

Figure C.3: Effluent overflow: Full PDE: Set I: positive concentration gradient downward

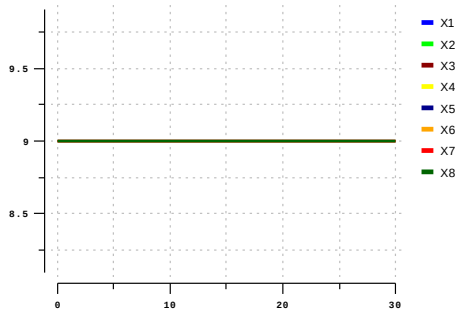

Figure C.4: Pure diffusion: Set II: uniform concentration

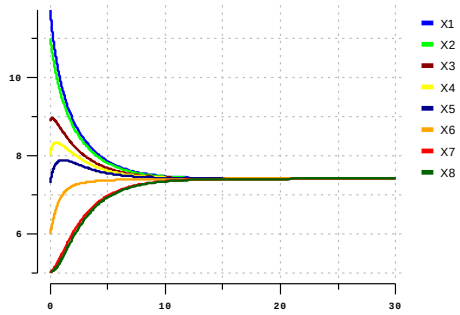

Figure C.5: Pure diffusion: Set II: positive concentration gradient upward

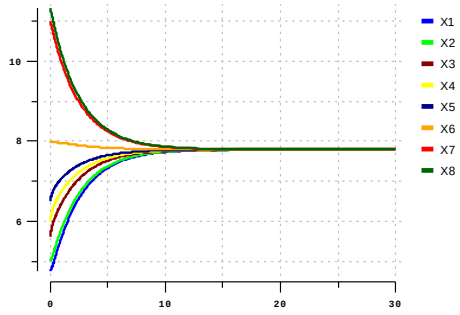

Figure C.6: Pure diffusion: Set II: positive concentration gradient downward

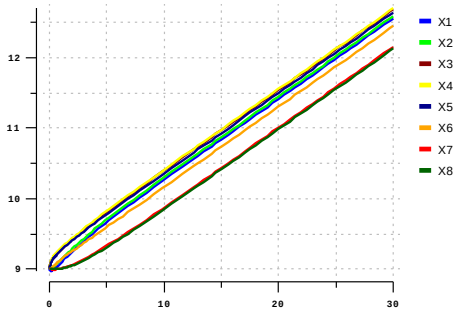

Figure C.7: Diffusion with source: Set III: uniform concentration

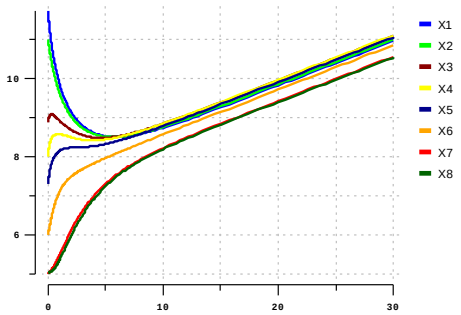

Figure C.8: Diffusion with source: Set III: positive concentration gradient upward

These three figures show what happpens in the clarifier, if there is no settling velocity or underflow, but there is diffusion along with a source influx. As expected, for all three initial concentration profiles, the concentration at all points increases. Also notice that when there is an initial concentration gradient, there is an initial diffusive regime, which smooths out the gradients, followed by the increase in concentration uniformly at all points. This is exactly what one expects by examining the corresponding PDE in this limit.

4. Set IV: Only gravitational settling:

These figures show the effect of gravitational settling in the clarifier, with no other terms affecting the time evolution of the concentration. There is no change in the concentration profile if the initial profile is uniform. However, when there is an initial gradient present, the evolution of the profile depends on the *direction* of the gradient. When the initial concentration increases upward, there is more or less a net decrease in concentration across the clarifier. If the concentration increases downward, there is a net increase in the concentration. This is because the gradient term appears in the PDE with a negative sign. It may also depend on the form of the Vesilind settling velocity.

5. Set V: Gravitational settling with diffusion:

These figures are similar to those in Set IV above. The magnitude of the diffusivity is small compared to the settling velocity, hence the diffusion does not have a major contribution, except to provide a diffusive

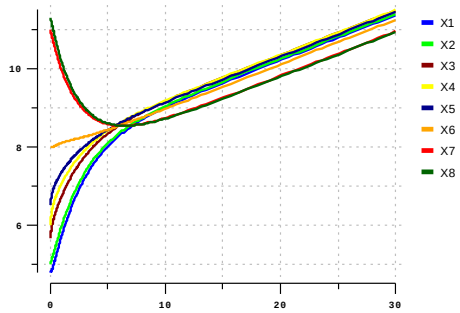

Figure C.9: Diffusion with source: Set III: positive concentration gradient downward

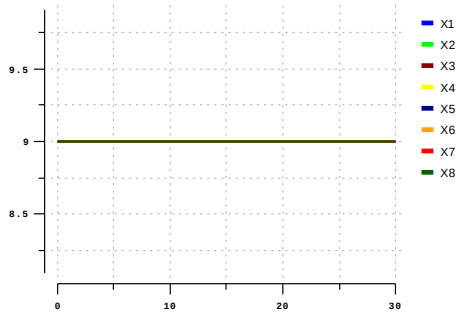

Figure C.10: Only settling: Set IV: uniform concentration

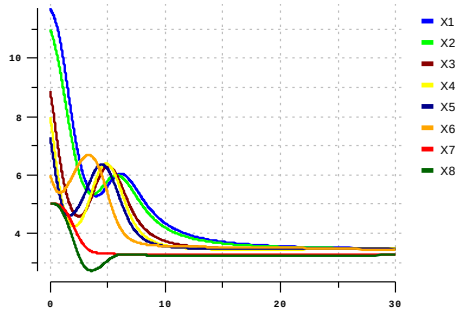

Figure C.11: Only settling: Set IV: positive concentration gradient upward

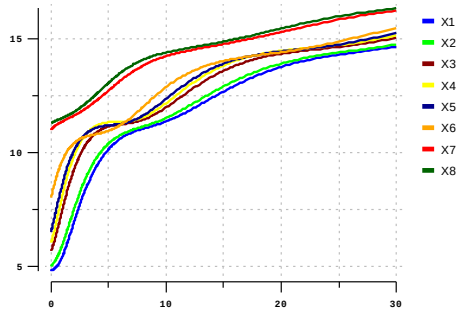

Figure C.12: Only settling: Set IV: positive concentration gradient downward

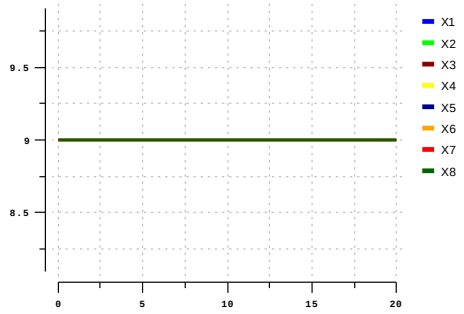

Figure C.13: Settling with diffusion: Set V: uniform concentration

envelope on the pure settling velocity curves.

C.3.2 Effluent pumped

1. Set I: Full PDE:

In this set of figures, the effect of explicitly drawing out the effluent can be seen. The concentrations along the clarifier do not settle to different asymptotic values. Instead, they all decay to the value of the source concentration asymptotically.

2. Set II: Diffusion with source:

Here again we see the effect of pumping the effluent out, with only diffusion and the source term being present. It can be seen that all concentrations decay ultimately to the source value.

C.4 Simulation results: batch case

C.4.1 Group 1: Taylor series approximation

The three figures in both Set I and Set II demonstrate that the simple linearization scheme using Taylor expansion is not such a good approximation.

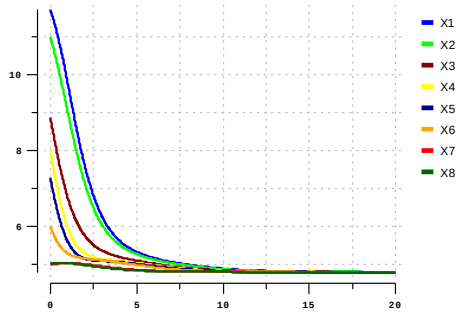

Figure C.14: Settling with diffusion: Set V: positive concentration gradient upward

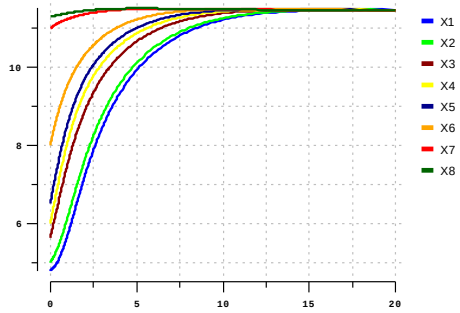

Figure C.15: Settling with diffusion: Set V: positive concentration gradient downward

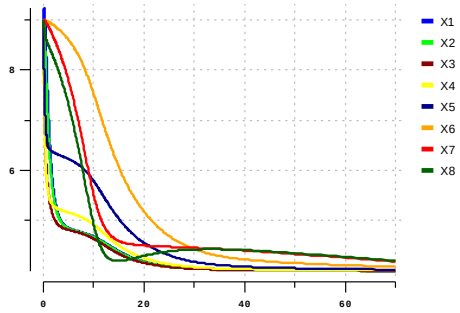

Figure C.16: Effluent pumped: Full PDE: Set I: uniform concentration

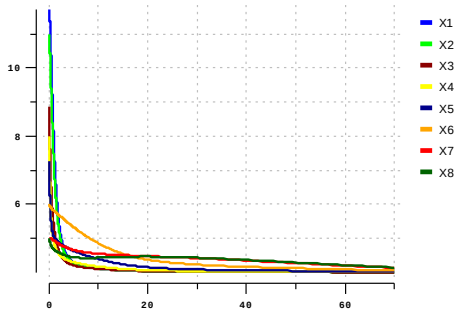

Figure C.17: Effluent pumped: Full PDE: Set I: positive concentration gradient upward

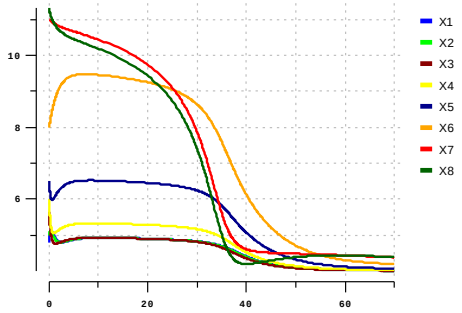

Figure C.18: Effluent pumped: Full PDE: Set I: positive concentration gradient downward

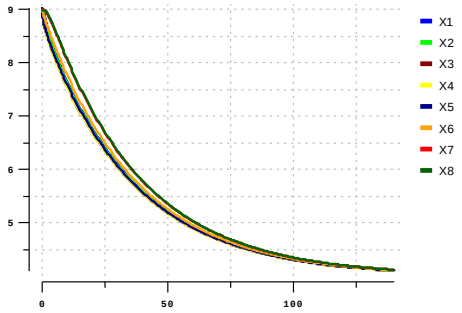

Figure C.19: Effluent pumped: Diffusion with source: Set II: uniform concentration gradient

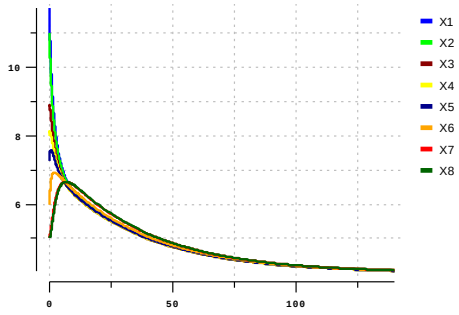

Figure C.20: Effluent pumped: Diffusion with source: Set II: positive concentration gradient upward

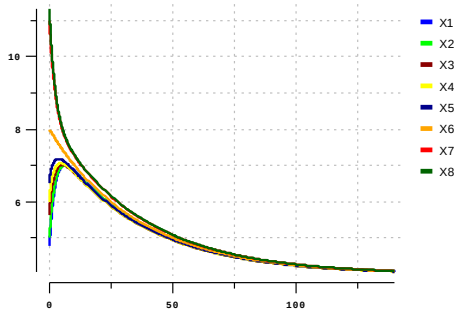

Figure C.21: Effluent pumped: Diffusion with source: Set II: positive concentration gradient downward

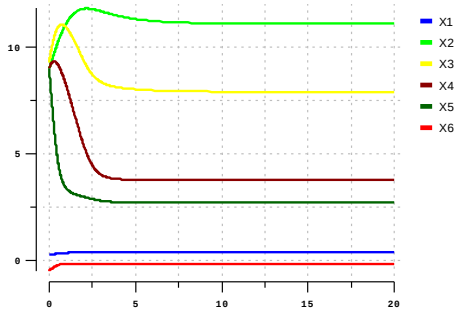

Figure C.22: Group 1: Set I: uniform concentration

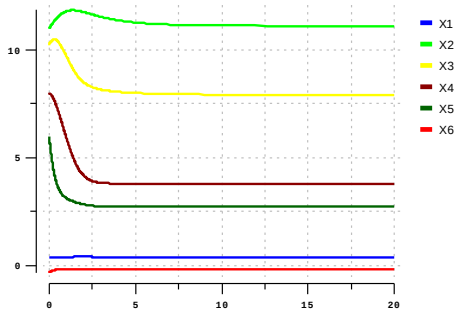

Figure C.23: Group 1: Set I: positive concentration gradient upward

1. Set I: Both settling and diffusion:

We see from the figures that the boundary concentrations X_1 and X_6 remain close to zero, with in fact X_6 staying slightly below zero, which is not physically meaningful. The intermediate concentrations settle to asymptotic values in the order $(X_2 > X_3 > X_4 > X_5)$. This is true for all three initial concentration profiles. Obviously, these results do not bear any resemblance to the simulations of Set V in the continuous case.

2. Set II: Only settling:

Again in Set II, we see that the boundary concentrations are always at zero. The other concentrations go to asymptotic values in the same order as in Set I, $(X_2 > X_3 > X_4 > X_5)$.

C.4.2 Group 2: Piecewise linear approximation

1. Set I:

We see that the piecewise linear approximation scheme indeed yields the best results. The concentrations do not go negative, and the results are, in fact, similar to those obtained in Set V for the continuous sedimentation case. This validates the fact that in the limit that there is no influx into, or outflow from, the clarifier in the continuous sedimentation case, it must reduce to the batch case (with the corresponding boundary conditions).

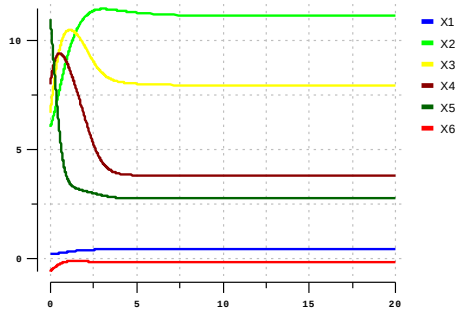

Figure C.24: Group 1: Set I: positive concentration gradient downward

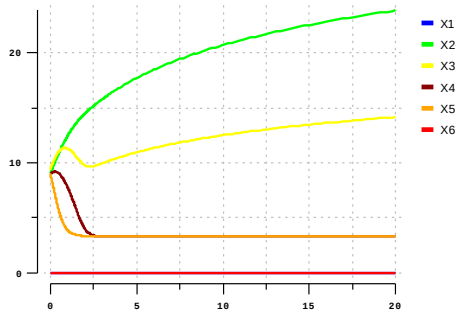

Figure C.25: Group 1: Set II: Only settling: uniform concentration

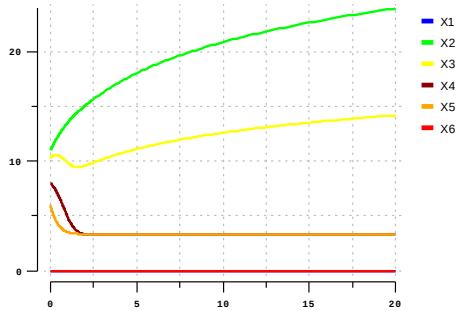

Figure C.26: Group 1: Set II: Only settling: positive concentration gradient upward

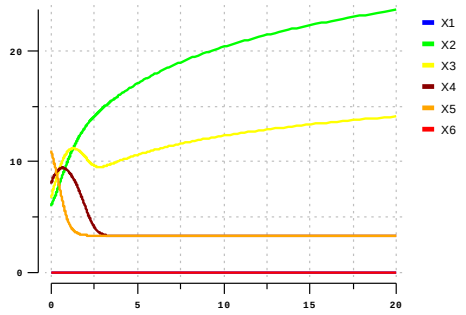

Figure C.27: Group 1: Set II: Only settling: positive concentration gradient downward

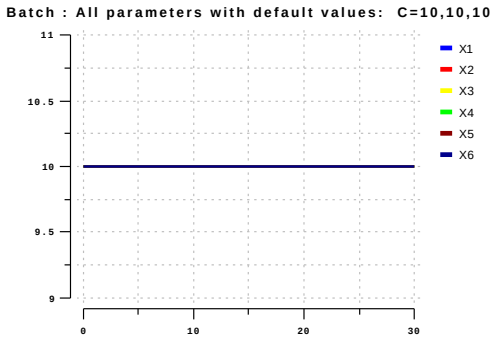

Figure C.28: Group 2: Set I: uniform concentration

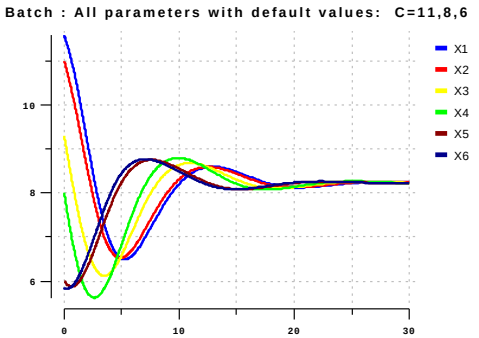

Figure C.29: Group 2: Set I: positive concentration gradient upward

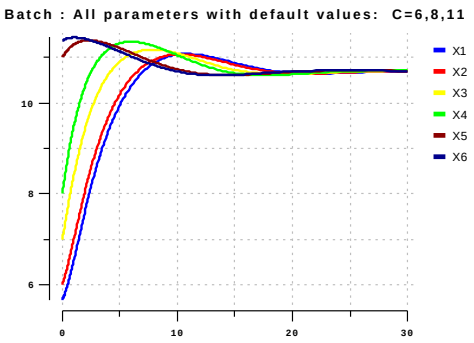

Figure C.30: Group 2: Set I: positive concentration gradient downward

C.4.3 Concluding remarks

Our results for the test case PDEs, and their various limits, lead us to the following remarks on the implementation of PDEs within WEST++ using orthogonal collocation:

- Even with the simplest possible discretization scheme (two and three elements), and a small number of collocation points (three and four), we were able to simulate the test case PDEs well enough to obtain physically meaningful results in the various limit cases. We also used the simplest possible values for (α, β), setting them to zero, which means that the collocation points were uniformly located within each element. The effect of varying (α, β) needs to be investigated further.
- We were able to handle non-linear boundary conditions in a meaningful way by performing piecewise linearization on the non-linear function involved, when simple linearization failed to yield reasonable results. We could show that the batch settling results using this piecewise linear approximation are similar to those of a particular limit case of continuous sedimentation, which is to be expected from physical considerations. However, the piecewise linearization scheme must be tested on other non-linear boundary conditions.
- In conclusion, PDEs can indeed be simulated effectively within WEST++ using orthogonal collocation with matrices.
- The PDEs transformed into ODE form are easily integrated in the WEST++ class hierarchy for re-use. Thanks to the efficient solution of ODEs in WEST++, simulation performance and accuracy is high.