A Language and Simulator for Hybrid Systems

PROEFONTWERP

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 1 september 1999 om 16.00 uur

door

Georgina Fábián

geboren te Boedapest, Hongarije

Dit proefontwerp is goedgekeurd door de promotoren:

prof.dr.ir. J.E. Rooda

en

prof.dr. M. Rem

Copromotor:

dr.ir. D.A. van Beek

Print: Universiteits Drukkerij, Eindhoven.

Stan Ackermans Instituut, Centrum voor Technologisch Ontwerpen.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Fábián, Georgina

A language and simulator for hybrid systems / by Georgina Fábián.
Eindhoven: Technische Universteit Eindhoven, 1999.
Proefschrift. - ISBN 90-386-2591-X
NUGI 852

Trefwoorden: simulatietalen / hybride systemen / regelsystemen / simulatoren
Subject headings: simulation languages / hybrid systems / continuous time systems / discrete-event dynamic systems / simulators

Summary

Many of today's industrial systems exhibit mixed discrete-event/continuous-time behaviour. Typical examples are batch processes from the process industry, digitally controlled plants, automobiles and aircraft. Models play an important role in the design process of such systems, providing means to evaluate design decisions at an early stage. Models written in a specification formalism present only the relevant system aspects. Such models contribute significantly to the understanding of the functionality of the systems. Hybrid systems are used increasingly in safety-critical applications, and therefore, their reliability is becoming more and more important. For verification purposes, formal semantic models and frameworks are used.

The χ language is a hybrid specification formalism, suitable for the description of discrete-event, continuous-time and hybrid systems. It is a concurrent language, where the discrete-event part is based on Communicating Sequential Processes (CSP [Hoare, 1985]) and the continuous-time part is based on Differential Algebraic Equations (DAEs). Models written in the χ language can be executed by the χ simulator.

The contribution of the present work is twofold. First, the hybrid χ language semantics is specified. This is an extension to the discrete-event semantics that had already been developed in a previous work. Second, the existing discrete-event simulator has been extended to execute hybrid models. To make a specification language suitable for the design and analysis of complex systems, it needs to have a precise semantics. Specifically, a hybrid language semantics has to preserve the pure continuous-time and the pure discrete-event functionality while integrating the two smoothly. In the case of the χ language, it is also important that models can be executed efficiently, using existing numerical algorithms.

Differential and algebraic variables with corresponding operators are defined in the χ language. This distinction emphasizes the physical role of the variables, and in this way supports the development of physically correct models. Steady-state initialization is supported directly by the language. Currently, index 0 and 1 systems of DAEs can be solved directly. Higher index systems can be modelled by using substitute equations.

The semantics of model composition is an important issue in hybrid languages. Model composition is well defined for pure discrete models by CSP. This thesis treats the continuous aspects of model composition. Continuous variables of sub-models are connected by the so-called channels in the χ language. After studying

existing hybrid languages, algebraic equations have been chosen to be the meaning of connections between continuous variables. This, together with using only local variables in processes, have contributed significantly to more clear and robust models. Furthermore, state-event conditions are evaluated only in consistent states leading to more predictable model behaviour.

The mathematical model of continuous state calculation is specified together with the hybrid language semantics; the latter in the form of a computational model. According to this model, discrete phases are divided into sub-phases, each ending with a consistent initial state calculation. Within each sub-phase, processes work with local variables only and interaction between sub-models is restricted to explicit communication. This simplifies reasoning about model behaviour.

The χ simulator has been extended to execute hybrid models. It integrates a nonlinear equation solver to solve the initial state problem and a DAE solver to solve the DAEs during the continuous phases. The local continuous variables of processes are dynamically mapped onto a global variable set that the solvers work with, to increase performance. The χ simulator allows sophisticated logging of continuous variables, such that their trajectory can be visualized by third party software. The user may also request a detailed trace file to be generated.

In the future, formal methods and techniques should be developed, based on the operational semantics defined in this thesis, to carry out formal analysis of hybrid χ models. A possible extension to the χ language is to include dimension information in the data types. In the present work, it is proposed to perform physical correctness checking based on this information. To extend the range of the physical systems that can be modelled in χ, it is necessary to be able to specify the so-called instantaneous equations. The performance of the χ simulator can further be enhanced by optimizing the equations, based on symbolic analysis and manipulation.

Preface

In 1996, in co-operation with P.L. Janson, I have built the first hybrid χ simulator (χ 0.1), as the design project of a two-year post-masters programme "Software Technology" [Fábián and Janson, 1996]. I was introduced to hybrid systems during this project, though, at that time, not to the full range of science that is related to them. That was yet to come. Following this project, I continued to work on the hybrid aspects of the χ language and on its simulator, which is presented in this thesis.

The first version of the χ simulator, and many later updates, have been used for over two years for educational and for research purposes. The second version (χ 0.5), with a new scheduler and with many new features was released in the autumn 1998. This thesis presents the design of this second version.

I want to express my gratitude to my supervisors, prof.dr.ir. J.E. Rooda and prof.dr. M. Rem for their guidance and encouragement during the course of this work. I would like to thank my daily supervisor dr.ir. D.A. van Beek for his stimulating guidance, advice and support.

I am grateful for present and previous members of the 'χ club' at the Systems Engineering Group who have contributed in many ways to this work: W.T.M. Alberts, N.W.A. Arends, V. Bos, L.F.P. Etman, A.T. Hofkamp, P.L. Janson, J.J.T. Kleijn, G. Naumoski and J.M. van de Mortel-Fronczak.

Contents

Chapter 1

Introduction

Hybrid systems are mixed discrete-event/continuous-time systems with both discrete and continuous components. At discrete time instants some actions take place. Between consecutive actions, the continuous components of the system change with time; these changes are governed by differential equations.

The interest in specifying dynamic systems in such a mixed way has grown enormously in the last two decades. A typical example for a hybrid system is a digital controller that controls a physical process. With the rapid spread of embedded systems such examples can be found today in almost all engineering fields. There are several systems that have been considered in the past as being purely continuous, such as multibody systems or chemical processes. However, if one is to study the way these systems interact with their environment, they are not purely continuous any more; they show discrete behaviour as well. For example, valves and pumps are switched on and off in chemical plants in a discrete fashion. The application areas of hybrid systems today range from process control, to robotics, flexible manufacturing, avionics and automated highway systems.

The hybrid approach provides modellers with great flexibility. Parts of the system or certain physical processes can be specified in detail using differential equations, whereas other parts or processes may be modelled at a high abstraction level by discrete actions. The complexity of hybrid models can grow enormous. This is because purely continuous and purely discrete systems can already be rather complex and in hybrid systems these two world-views are present in a combined way. It is difficult to envisage the behaviour of complex hybrid systems, therefore, simulation is an important means to get insight into their dynamics. However, simulation only is not sufficient to deduce system properties. Hybrid systems are increasingly used in safety-critical applications, therefore, their reliability is becoming more and more important. For the scientific analysis of these systems, a formal model is necessary, as is a formal framework for the verification of system properties. As a first step towards this goal, hybrid specification languages must have a precise language semantics.

As is usually the case with today's technologies, specification and analysis of hybrid systems require techniques and tools of several different disciplines. The description and control of physical systems lies in the area of mathematical modeling and control

theory. The control of production systems is the expertise of process and production technologists. Hybrid specification formalisms and verification of hybrid systems is rooted in computer science. Researchers have different interests in specifying hybrid systems. In control theory, the emphasis lies on synthesis of reliable and possibly optimal control for hybrid systems. In production engineering, often an existing production system must be analyzed and its performance enhanced, possibly within the boundaries of certain safety criteria. Computer scientists are developing formal frameworks for the analysis and verification of hybrid systems.

It is a rather challenging task to facilitate all these needs by a single hybrid formalism. Today, there are two mayor tendencies in hybrid languages. On the one hand, there are those languages and tools that are aimed at modelling complex, industrial-size systems. Modelling and simulation of these systems require high level language concepts, complex data types, an integrated development environment and a fast simulator. However, the complexity and flexibility of these languages is also the cause of their weakness. There is usually no formal language semantics available, therefore, a scientific analysis of the systems specified in these languages is not supported. On the other hand, there are specification formalisms that are aimed at formal analysis and verification of hybrid systems. For them, a formal language semantics is available together with a verification methodology. Because of the enormous complexity of hybrid systems, those formalisms where a verification is feasible are often limited. For example, only primitive control structures and data structures can be used, and usually only Ordinary Differential Equations (ODEs) can be specified or even only linear ODEs.

The aim of the hybrid χ language is to bridge the gap between these two tendencies. With carefully selected language constructs and data types, the expressiveness of the language is sufficient enough to describe industrial systems. One such example has been the specification of a plant where integrated circuits are manufactured [Rulkens et al., 1998]. The specification describes the operation of 500 machines producing 60 different products. On the other hand, the χ language is also designed to facilitate analysis and verification. Therefore, only a minimum number of orthogonal language constructs are used, with a well-defined semantics. The formal discrete-event semantics is described in [Bos and Kleijn, 1999], and an example of the verification of a discrete model can be found in [Kleijn et al., 1998].

The contribution of this thesis is twofold. First, the hybrid χ semantics is specified in an operational form. A discrete-event operational semantics had already been available at the beginning of this project. In this work, the semantics has been extended with hybrid elements. The extension consists of the semantics of the continuous language elements (the way the values of the continuous variables are defined by the equations and by the continuous connections), and the semantics of possible interactions between the discrete and the continuous components. These hybrid aspects together with the existing discrete-event semantics define the complete hybrid χ language semantics. When defining different language constructs, several alternatives have been considered. These alternatives are presented throughout in

this thesis together with their pros and cons. Also, many examples are given that illustrate the intended use of language elements.

As part of this project, a hybrid χ simulator has been developed, in order to evaluate by means of examples the decisions made about the language semantics. The hybrid χ simulator is based on an existing discrete-event simulator that could simulate purely discrete-event χ models only. The second contribution of this thesis is the description of the design of the extended hybrid simulator. This description gives further insight into the techniques of hybrid simulation.

1.1 Thesis outline

In Chapter 2, hybrid modelling and simulation is introduced. Hybrid language concepts and new simulation phenomena that are typical to hybrid systems are discussed. Also, an overview is given of hybrid simulation languages.

In Chapter 3, the χ language is presented. The chapter is concluded with a survey on connection semantics in hybrid languages. This explains the choice for the connection semantics in χ. A new concept has been developed for the type system to express dimensionality of physical quantities; it is not implemented yet by the χ simulator. This concept is presented in Appendix A.

The next two chapters treat the hybrid aspects of the language semantics. In Chapter 4, the mathematical model of the continuous state calculation technique is presented. Chapter 5 concentrates on the integrated discrete/continuous behaviour. This chapter specifies the scheduling algorithm implemented in the hybrid χ simulator, and important features of this algorithm are illustrated by examples.

The next three chapters present the design of the χ simulator. First, in Chapter 6, the architecture of the χ simulator is outlined. The implementation of the continuous variables and the representation of the equations is discussed in Chapter 7. Numerical calculations, state-event location and representation are discussed in Chapter 8.

In Chapter 9, a collection of χ models are presented to illustrate the broad application area of the χ language. Furthermore, it is illustrated how substitute equations can be used for index reduction and to reveal hidden constraints in Differential Algebraic Equations.

The thesis is concluded in Chapter 10, with a discussion of the contribution made and suggestions for further research.

Chapter 2

Hybrid modelling and simulation

In this chapter, hybrid modelling and simulation techniques are introduced. First, hybrid modeling is treated, which specifies the concepts that a hybrid language may be able to express. Following this, *hybrid simulation phenomena* are discussed. These are phenomena that typically occur when simulating hybrid systems. Also, current technologies that handle these situations are presented. Finally, a brief overview of the history and the state-of-the-art in hybrid simulation is given in Section 2.3.

2.1 Hybrid modelling

In hybrid systems, there are discrete and continuous components represented by discrete and continuous variables. The discrete variables may change at discrete time instants only. Their values are for instance, determined by program segments that are executed at discrete times. The values of the continuous variables change continuously in time. Their values are determined by differential equations. In some continuous languages, the equations are specified in other forms, for example as assignments.

Hybrid modelling combines discrete and continuous modelling techniques. It is desirable that in a hybrid language purely continuous and purely discrete sub-systems can be described. Therefore, these techniques are equally important. The most relevant concepts of a discrete formalism for hybrid languages are:

- parallelism
  Program segments can arbitrarily be combined sequentially or in parallel.
- modularity
  A system can be composed of sub-systems, with each sub-system having a well defined interface. The compound behaviour of the system is defined in terms of the behaviour of the sub-systems. There are hybrid languages, such as Omola [Mattson et al., 1993] and Modelica [Modelica, 1997], that use the object-oriented paradigm.

- communication
  Sub-systems can communicate with each other. The means of communication can be message passing, shared variables, synchronous data channels, etc.

- time-events
  Time-events specify the time of an event explicitly, relative to the current simulation time. They can also be considered as waiting commands, because the execution of the given sub-system is ceased for the specified amount of time.

- distribution functions
  Distribution functions facilitate modelling of stochastic behaviour of discrete-event systems.

From the continuous point of view, the most relevant concepts for a hybrid formalism are:

- symbolic equation-oriented approach
  The continuous behaviour of the system can be described symbolically, in the form of equations. There is no restriction on the form or on the order of the equations.

- Differential Algebraic Equations (DAEs)
  Differential equations can be specified with algebraic constraints. As a next step, partial differential equations can be specified.

- differential inclusions
  Lower and upper bounds can be specified on the derivatives of the variables. For example, the differential inclusion $1 \leq \dot{x} \leq 3$ restricts x to any trajectory whose slope stays within the interval [1,3].

- rate predicates
  Conjunctions of linear inequalities can be formulated on the time derivatives of the variables. For example, $\dot{x} < \dot{y}$ ensures that y grows faster than x.

- modularity
  A system can be composed of sub-systems and the compound behaviour (a global set of equations) is defined in terms of the behaviour of the sub-systems.

In hybrid modelling, there is a new element, namely, the discrete and the continuous system components can interact with each other. The discrete components influence the continuous behaviour by changing the values of the continuous variables by discrete actions. This can be accomplished in the following ways:

- A continuous variable is assigned.

- A discrete variable that is used in an equation is assigned. As a consequence, all other continuous variables that depend on its value are changed implicitly.

- One equation is replaced by another equation.

Continuous components influence the discrete behaviour in the following ways:

- The value of a continuous variable is referenced in a discrete action.
- A boolean condition that depends on the values of some continuous variables becomes true and triggers a discrete action. This is a so-called *state-event*.

Hybrid languages can express (some of) these kinds of interactions. A comprehensive study of hybrid modelling phenomena can be found in [Branicky, 1995].

2.2 Hybrid simulation

Interactions between the discrete and the continuous components result in new simulation phenomena that are typical to hybrid systems. A hybrid simulator therefore, must be able to handle these. There is, however, no general agreement as to what extent a simulator should be able to handle these phenomena to qualify as a hybrid simulator. In each of the topics shown below, there are rather advanced concepts and techniques reported in the literature. The implementation of these techniques in today's simulators though often lags behind theory. In a recent paper [Mosterman, 1999], Mosterman presents a survey of todays hybrid simulation packages and their support of these phenomena.

2.2.1 Differential Algebraic Equations

Strictly speaking DAE solving falls into the area of continuous modelling, but it has a crucial role in hybrid systems' modelling as well. Differential Algebraic Equations are a set of differential equations with additional algebraic constraints. This is in contrast to Ordinary Differential Equations (ODEs), where there are no such constraints. While the solving of ODEs remains important, in the last two decades research interest has turned to the numerical solution of DAEs. DAEs present analytical and numerical difficulties that are quite different from those encountered in ODEs [Petzold, 1982]. The systematic study of DAEs began in the 80's and has resulted in several methods that can solve DAEs directly. These research results are summarized in the monographs [Brenan et al., 1996, Griepentrog and März, 1986, Hairer et al., 1980, Hairer and Wanner, 1991].

The continuous behaviour of many physical systems can be described with DAEs. In the most general form they are specified as

$$\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, t) = \mathbf{0}, \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the vector of variables, $t \in \mathbb{R}$ is the independent variable and $\dot{\mathbf{x}}$ denotes the time derivative of $\mathbf{x}$ with respect to t. Equation (2.1) is the *implicit* form. If equation (2.1) can be re-written into the *explicit* form

$$\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}, t), \tag{2.2}$$

then (2.1) is in fact an Ordinary Differential Equation (ODE). In this case, (2.1) is called an *implicit ODE*.

However, re-writing a system of DAEs into a system of ODEs may not always be possible, or may not be desirable. Often such re-writing requires some problem-specific manipulations of the equations, and/or simplification of the problem. Also, after such a re-formulation, the expressiveness of the model is often reduced. Consider for example, the DAEs that describe a tank of substance that is being emptied. Variables V and h denote the volume and the height of the substance in the tank, respectively, Q_o is the outgoing flow, A is the cross-sectional area of the tank and k is constant.

$$\begin{aligned} \dot{V} &= -Q_o \\ Q_o &= k\sqrt{h} \\ Ah &= V. \end{aligned}$$

To obtain the corresponding ODE, variables Q_o and h need to be eliminated.

$$\dot{V} = -k\sqrt{\frac{V}{A}}.$$

This elimination itself already reduces the expressiveness of the model, since each time the values of Q_o or h are needed, they have to be calculated from V. Finally, re-formulation slows down development of complex models, because it must be repeated each time a model is altered. Therefore, in many applications it is desirable to be able to solve the DAEs directly. If the algebraic constraints in the DAEs are explicit, then we have a *semi-explicit* DAE system

$$\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t) = \mathbf{0} \tag{2.3a}$$

$$\mathbf{g}(\mathbf{x}, \mathbf{y}, t) = \mathbf{0}, \tag{2.3b}$$

where the Jacobian of $\mathbf{f}$ with respect to $\dot{\mathbf{x}}$ ($\partial\mathbf{f}/\partial\dot{\mathbf{x}}$) is non-singular. The variables are now divided into two categories. The variables in $\mathbf{x}$ are called *differential variables*, whereas variables in $\mathbf{y}$ are called *algebraic variables*.

Index

DAEs are characterized by their *index*. Here, the definition of the *differential index* is given, as defined by Gear [Gear, 1988].

Definition 1 *The index of equation (2.1) is* m*, if* m *is the smallest number such that the system of equations*

$$\begin{aligned} \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, t) &= \mathbf{0}, \\ \frac{d\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, t)}{dt} &= \mathbf{0}, \\ &\vdots \\ \frac{d^{(m)}\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, t)}{d^{(m)}t} &= \mathbf{0}, \end{aligned} \tag{2.4}$$

can be transformed into an explicit ODE ($\dot{\mathbf{x}} = \mathbf{g}(\mathbf{x}, t)$) by algebraic manipulations.

Obviously, the index of an ODE is zero. In general, the higher the index the greater the numerical difficulty one is going to encounter, when trying to solve the system numerically. Systems with indexes greater than 1 are particularly difficult to solve. These are called *high index DAEs*.

The first method to solve DAEs was proposed by Gear [Gear, 1971]. This is based on a backward difference formula (BDF method). Several codes have been written to solve DAEs that exploit this method. For example, LSODI [HindMarsh, 1980], DASSL [Petzold, 1983], DASOLV [Jarvis and Pantelides, 1992]. These codes are capable of solving DAEs of index 0 and 1. Furthermore, implicit Runge-Kutta methods, originally developed to solve stiff ODEs, have been successfully applied to DAEs [Petzold, 1986, Hairer et al., 1980]. The direct solution of high index systems, except for some rather specific forms, however, is not possible. Results of studies in this direction can be found in [Gear et al., 1985, Brenan et al., 1996]. Therefore, high index problems are mostly solved by first reducing the index to 0 or 1, then the resulting system is solved by available solvers.

Consistent initialization

Consider the DAEs in the form

$$\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t) = \mathbf{0} \tag{2.5}$$

One of the key differences between DAEs and ODEs is that in DAEs not all variables can freely be initialized. The initial values of variables $\mathbf{x}, \dot{\mathbf{x}}$ and $\mathbf{y}$, denoted by $\mathbf{x}(0), \dot{\mathbf{x}}(0), \mathbf{y}(0)$, must satisfy equation (2.5) at time 0:

$$\mathbf{f}(\dot{\mathbf{x}}(\mathbf{0}), \mathbf{x}(0), \mathbf{y}(0), 0) = \mathbf{0} \tag{2.6}$$

If they do so, then the initial values are *consistent*. A further characteristic of DAEs is that the initial values may have to satisfy additional algebraic constraints. These are *hidden constraints* that are not explicitly present in the original DAEs. This is particularly true for high index DAEs, but may also be the case in some index 1

systems as well. These conditions can be revealed by executing the m differentiations in (2.4) [Leimkuhler et al., 1991]. In this way however, all equations are differentiated, possibly unnecessarily. Pantelides proposes an algorithm [Pantelides, 1988], that identifies those subsets of the system that must be differentiated in order to reveal such hidden constraints.

Hidden constraints may be present in index 1 DAEs. This is usually the case if the system is in the form

$$\begin{aligned} \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, \mathbf{u}, t) &= \mathbf{0} && \text{(2.7a)} \\ \mathbf{g}(\mathbf{x}, \mathbf{u}, t) &= \mathbf{0} && \text{(2.7b)} \\ \mathbf{u} &= \mathbf{h}(t). && \text{(2.7c)} \end{aligned}$$

The new variable category $\mathbf{u}$ represents variables that are functions of t. These are the so-called *input variables*. Those components of $\mathbf{x}$ that occur in (2.7b) are *dependent differential variables* (to be specified later). Differentiating (2.7b) once produces equations for the derivatives of the dependent differential variables. By combining these equations with (2.7a), hidden constraints may be revealed. Take, for example, the following index 1 problem

$$\begin{aligned} \dot{x}_1 + \dot{x}_2 &= a(t) && \text{(2.8a)} \\ x_1 + 2x_2 &= b(t). && \text{(2.8b)} \end{aligned}$$

After differentiating (2.8b), and subtracting (2.8a) from the result, we get the hidden constraint on the initial value of $\dot{x}_2$:

$$\dot{x}_2 = \dot{b}(t) - a(t).$$

In general, dependent differential variables are those components of $\mathbf{x}$ in equation (2.1) that can be written in the form

$$\begin{aligned} \mathbf{g}(\mathbf{x}, \mathbf{u}, t) &= \mathbf{0} && \text{(2.9a)} \\ \mathbf{u} &= \mathbf{h}(t). && \text{(2.9b)} \end{aligned}$$

This form may be obtained after differentiation of (2.1) and algebraic manipulations. Dependent differential variables are usually a sign of high index, but not necessarily as example (2.8) illustrates.

2.2.2 Discontinuities

Discrete interactions in the continuous behaviour of the system result in discontinuous functions. A discontinuous differential equation can be described as

$$\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, t) = \begin{cases} \mathbf{f}_1(\dot{\mathbf{x}}, \mathbf{x}, t) & \mathrm{s}(\mathbf{x}, t) > 0 \\ \mathbf{f}_2(\dot{\mathbf{x}}, \mathbf{x}, t) & \mathrm{s}(\mathbf{x}, t) < 0 \end{cases} \qquad \text{(2.10)}$$

where s is the *root function* or *switching function*. When s becomes zero, a discontinuity occurs. The surface where the switching function is zero is called the *root* or *manifold of discontinuity*. In [Cellier et al., 1993], discontinuous functions are called *D-functions* and functions with discontinuous first derivatives are called *DD-functions*.

Discontinuities pose problems to most numerical solvers. Usually, they can not integrate D-functions. The reason for this in the case of BDF methods (used in the χ simulator) is further explained later in Section 8.1.3; in the general case it is treated in [Cellier, 1986]. Most numerical solvers however can integrate DD-functions, although this is with a cost of low order and step-size, therefore, simulation slows down in the vicinity of the discontinuity.

Therefore, in hybrid simulators, when integrating the equations, the simulator itself must prevent the solver from attempting to integrate through discontinuities. The discontinuity must be detected beforehand to stop the integration and re-start it afresh after the discontinuity has been processed. This approach is facilitated by DAE and ODE solvers in the following way. Giving the switching function to the solver, it detects when the integration crosses the root, and by backward interpolation, it locates the root. In the effort of locating discontinuities, the modeller has to help as well, by making them explicit in the model. Hybrid languages provide several means to do this, such as state-events or conditional equations. Consider for example, the following discontinuous function.

$$x = \begin{cases} 0 & y \leq 0 \\ 1 & y > 0 \end{cases} .$$

This can be modelled by the simple equation

$$x = x_{set}$$

where x_{set} is a discrete variable. The value of x_{set} is governed by the following piece of code, where, for reasons of simplicity, it is assumed that y is a monotonically increasing function and its initial value is less than 0.

$$x_{set} := 0;\ \text{“wait until } y > 0\ \text{”};\ x_{set} := 1$$

When a discontinuity occurs, the system may become inconsistent, that is, the continuous variables do not satisfy the equations any more. Therefore, the simulator has to calculate a new consistent state. In the above example, when x_{set} is assigned the value 1, x still has the old value 0, so that the equation does not hold. After this, a new value must be calculated for x.

In connection with discontinuous functions, further difficulties have been reported in the literature. In this thesis, these problems are not addressed, but users of hybrid simulators should be aware of them. In BDF codes, it is possible that a discontinuity is completely missed, if the dynamics of the switching function is much faster than

the dynamics of the equations. See [Brenan et al., 1996, page 136], [Wijckmans, 1996, Chapter 7]. Equation (2.10) divides the operation domain into two: the one where $s > 0$ and the other one where $s < 0$. If a variable step-size method is used, when approaching the manifold of discontinuity, the step-size of the integration is decreased and the simulation slows down. When stepping into the other region, a discontinuity is detected, so that the system switches to the other equation set. If in the two regions the vector fields are directed toward the manifold of discontinuity, the simulation may switch between the two regions at a very fast rate, producing a fast *chattering behaviour* that slows down the simulation excessively. Solutions can be found in [Filippov, 1964, Stewart, 1990, Mosterman et al., 1998].

2.2.3 State-events

State-events specify in an implicit way time points when a discrete action happens. A state-event is defined by a boolean condition, called the *state-event condition*, that is dependent on the values of continuous variables. The state-event occurs at the first time instant when the state-event specification is true. For example, when the pressure in a reaction tank reaches a critical value, an emergency operation must start. This may be specified in a hybrid language as

"wait until $p > 100$ Pa" ; *valve* := "open"

where variable p represents pressure, and the second statement means that a safety valve is opened. In a hybrid system, several state-events may be awaited at the same time, and the set of state-event specifications dynamically changes in the course of the simulation. It is also possible, that two state-events occur at the same time instant. State-events may furthermore trigger other state-events at the same time instant. A hybrid simulator must detect when a state-event occurs, locate the time point of the event accurately and determine the continuous state of the system at that point [Cellier, 1986, Park and Barton, 1996]. For the detection of state-events, the root finding mechanism of DAE and ODE solvers, described previously in Section 2.2.2 can be used.

2.2.4 Flexible sets of equations

Many hybrid systems describe dynamic systems that operate in different modes. In the different modes different sets of equations may describe the continuous behaviour of the system. This can be specified for example, by means of conditional equations, where boolean conditions indicate the operation modes.

A discontinuity may occur when the system transits from one mode to another one. This must be detected and a consistent new state has to be calculated. For this calculation, some additional equations may be necessary, that are considered only for the calculation of the new state. These equations are called *instantaneous*

equations. They usually express continuity and physical conservation principles. In the χ language, currently only continuity principles are considered. This is presented in detail in Chapter 4. For the derivation of physical conservation principles, see [Mosterman and Biswas, 1996].

A more advanced way of using flexible sets of equations is to allow variables and their governing equations to be dynamically allocated and de-allocated (see e.g., [van Beek et al., 1997]). If there is no such facility in a hybrid language, then in those modes where a variable should vanish it can be given some default value.

2.3 Hybrid simulation languages

The development of general purpose mixed continuous/discrete simulation languages and tools has come to the current state via two trajectories. On the one hand, purely continuous packages have been later extended to be able to model discrete actions. On the other hand, several formalisms previously used in computer science to describe discrete-event systems have been extended to include continuous language elements. In this section, the milestones of both trajectories are described, but without the aim of completeness. The domain oriented packages are not treated here.

Dynamic models of industrial systems have been written since the appearance of digital computers. These models, written in general purpose computer languages, such as ALGOL or FORTRAN, provided engineers with enormous flexibility. This approach however requires from engineers next to the engineering knowledge of the given discipline, much knowledge of general programming and numerical algorithms. Also, the development of these models requires a great amount of programming. Therefore, it is a rather costly approach, though up to today many models of industrial systems are still developed in this way.

2.3.1 Continuous simulation languages

In recognition of the need for dedicated modelling and simulation tools, the first general-purpose continuous languages such as ACSL [Mitchell and Gauthier, 1976], and CSMP [Speckhart and Green, 1976] appeared in the 70's. These languages are based on the CSSL standard [Augustin et al., 1967]. These first tools provided engineers with an integrated modeling environment, where they could concentrate on the model formulation rather than on coding the numerical solution of the equations. The main critique of these languages today is that they describe systems in a flow-sheet, input-output block style [Cellier and Elmqvist, 1993, Elmqvist and Mattson, 1997]. In this style, the direction of the information flow is fixed for each connection which limits the re-use of the components. The equations must be re-arranged into explicit state-space form ($\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}), \mathbf{y} = \mathbf{g}(\mathbf{x}, \mathbf{u})$), which requires still much engineering effort. Flow-sheet style simulation is however still used in today's popular simulation packages, such as SIMULINK [Simulink, 1993]

and System Build [SystemBuild, 1984]. Also, the number of equations in the early continuous packages were rather limited, compared to the size of today's models of industrial systems.

In the next generation of continuous languages, such as SpeedUp [Perkins and Sargent, 1982], DPS [Wood et al., 1984] and ASCEND [ASCEND, 1997] the continuous behaviour of the system is described symbolically, by equations. This is a large difference compared to the flow-sheet style, because now equations can be entered in arbitrary form, and the simulator handles their solution. The equation-oriented approach has been made possible by the advancement in numerical methods, computer hardware and software. Although, in ASCEND conditional equations can be used, so far these languages are suitable to model continuous systems only.

The following generation, and to date the state-of-the art in this trajectory is that group of continuous languages that have been extended with discrete-event language elements. Usually, discontinuities can be modelled, state-events can be specified and to some degree, discrete actions can be specified as well. Representatives of this group are gPROMS [Barton, 1992], Dymola [Elmqvist et al., 1993, Elmqvist et al., 1996], Omola [Mattson et al., 1993] and Modelica [Modelica, 1997]. gPROMS is aimed at process modelling. It has the most sophisticated discrete-event modelling capabilities among these languages which makes it suitable to model operating procedures, such as batch recipes. Also, numerically, it is a very robust tool; it can solve large sets of equations efficiently and it can solve partial differential equations and partially determinable systems [Jarvis, 1993]. Dymola builds extensively on hierarchical sub-model decomposition and it uses object-oriented technology. Sub-models can be connected in different ways that express the physical nature of the connections. Dymola uses symbolic formula manipulations, among others, for index reduction and optimization. Omola is very similar to Dymola; it extends the object-oriented paradigm and inheritance is defined for both models and connections. Finally, Modelica is a new language for physical modelling that aims to unify several different (mainly continuous) languages. Currently, however, the languages of this last group are not suited to model purely discrete sub-systems. Most notably there is no notion to specify communication between discrete sub-systems.

The field of computer simulation of dynamic systems (which has lead to the simulation of combined discrete/continuous simulation) has been a very active, fast moving research field since its very beginning. At regular intervals, the current states of the simulation packages are evaluated and new phenomena are identified to be supported by the next generation simulators. Such milestone papers have been published in chronological order by Fahrland [Fahrland, 1970], Perkins [Perkins, 1986], Cellier [Cellier, 1986], Marquardt [Marquardt, 1991], Pantelides et al.[Pantelides and Barton, 1993], van Beek et al. [van Beek and Rooda, 1998a] and Mosterman [Mosterman, 1999].

2.3.2 Discrete formalisms

In the last few decades, the interest in program specification has turned from writing computer programs that calculate output from input to specify and verifying *reactive systems*. Reactive systems are systems that continuously interact with their environment. Such systems can only be analyzed together with their environment in time perspective. The challenge today is to specify real-time behaviour of reactive systems, and to give a framework for the verification of timed properties. With the evolution of specification formalisms, the underlying semantic and time models became more and more sophisticated and complex.

The discrete-event approach describes system behaviour as a sequence of instantaneous actions. Between such actions the system remains unchanged. In many cases, a digital clock is introduced in discrete-event formalisms. In such systems, time can advance only with a multiple of a fixed time-unit. These are the so-called discrete-time systems. Such an approach is sufficient to describe digital systems that interact with digital environments. The underlying time model is a *discrete-* or *integral-time* model, for example, being the integer numbers $\mathbb{Z}$. Examples can be found in [Jahanian and Mok, 1986, Ostroff, 1989, Harel et al., 1990, Nicollin et al., 1990]. Although, integral time models describe physical systems less accurately, the verification techniques are relatively simple under this model and important system properties are decidable [Henzinger et al., 1992, Alur and Henzinger, 1993, Alur and Henzinger, 1994].

The next step towards more realistic modelling of time is to use a dense time model for example, the non-negative real numbers $\mathbb{R}_{\geq 0}$, in which case the time is called *real time*. With the introduction of real time it is possible to specify non-instantaneous actions that take an arbitrary amount of time. This can be modelled by delays that are not multiples of the system clock as in the previous approach. Examples for a dense time model can be found in [Koymans et al., 1987, Reed and Roscoe, 1987, Pnueli and Harel, 1988, Lewis, 1990, Moller and Tofts, 1990, Abadi and Lamport, 1992, Alur et al., 1993, Nicollin et al., 1993]. The real time approach can be considered as a system description that has a single continuous variable: the time. The difficulty of this time model is that verification of many timed properties is not feasible, i.e., they are undecidable. A detailed discussion about integral and real time models is presented in [Alur and Henzinger, 1992].

From the modelling point of view, in many cases, it is desirable to consider actions to be instantaneous. This is the case for example, when the time scale of certain actions are much smaller than that of primary interest. When developing a formal model of reactive systems this abstraction helps to make models in which only relevant delays are represented. A time model that allows more actions to take place at a single time instant is the *super dense time* model, introduced in [Maler et al., 1992][1].

[1] Although, in the original paper this time model is introduced as a dense time model, in a later article [Manna and Pnueli, 1993] the authors call it super dense. In this thesis, it is also referred to as super dense.

To be able to model systems that interact with a continuously changing environment, previous timed formalisms have been extended to hybrid formalisms. In hybrid systems, continuous variables are governed by complex rules: by ODEs or DAEs. The introduction of discrete changes into a continuous formalism leads to a problem in the mathematical model of continuous time and the continuous variables. A mathematical framework is presented in [Iwasaki et al., 1995], where, in order to preserve the continuity of variables through instantaneous discrete changes, *hyper-reals* are used for time and for continuously changing quantities. In this model, actions take an infinitesimal amount of time (an amount that a real valued clock can not measure). During such actions, the continuous variables remain continuous. Furthermore, this representation allows instantaneous actions to take a different amount of (infinitesimal) time.

One of the most often used hybrid formalisms is that of hybrid automata [Alur et al., 1995]. A hybrid automaton consists of a set of control locations, each labeled with evolution laws. The values of the variables at the locations change in time according to these laws. The locations are connected by transitions, each labelled with a set of guarded assignments. A transition is enabled when the associated guard is true. When the system takes a transition, assignments take place. The locations are also labelled with invariants that must hold when control resides at the location. There are several versions of hybrid automata. Research focuses today on identifying sub-classes of hybrid automata where the reachability problem (whether an unsafe region is reachable from an initial region) is decidable [Henzinger et al., 1995, Lafferriere et al., 1999]. One of the most promising sub-classes is that of linear hybrid automata. In these systems, the continuous components of the system can change only linearly, and all terms used must be linear. A comprehensive theory of these systems can be found in [Alur et al., 1995], where verification of examples are reported using symbolic model checkers. Another hybrid automaton is the I/O automaton [Lynch et al., 1996], where connections between sub-systems define input/output relations. SHIFT [Deshpande et al., 1997] is a programming language and simulator for describing dynamic networks of hybrid automata. In SHIFT, interconnections of the automata can dynamically be changed, and new automata can be generated and removed from the system. Also, universal quantifiers can be used to query the values of the variables. This mechanism makes SHIFT suitable to describe systems where individual components, described by means of differential equations, arrive at and leave the system.

Petri nets have been extended in various ways to express continuous dynamics. One kind of extension can be found in [David and Alla, 1990, Bail et al., 1992]. In these hybrid petri nets, there are discrete and continuous places and discrete and continuous transitions. A continuous place is marked by a real value, instead of the usual integer value. Continuous firing is carried out like a positive flow. When a transition is fired, a fraction amount of marks are taken out of the places. In another hybrid extension [Okuda and Ushio, 1990], very similarly to hybrid automata, a place corresponds to a state of a variable and the firing of a transition corresponds to crossing a landmark.

Finally, CSP [Hoare, 1985] has also been extended with time and with continuous language elements. A timed CSP (TCSP) is specified in [Reed and Roscoe, 1986] and – with some other modifications – in [Hooman, 1991]. The discrete-event part of the χ language is another timed CSP. This is described, for example, in [van de Mortel-Fronczak et al., 1995, van de Mortel-Fronczak and Rooda, 1996, van de Mortel-Fronczak and Rooda, 1997]. A hybrid extension of CSP is given in [Jifeng, 1994]. This extension introduces a continuous statement for differential equations in the form of $\mathcal{F}(\dot{s}, s) = 0$, where s is a vector of continuous variables. Also, boolean conditions are introduced for state-event specifications. A follow up of this work, a hybrid CSP (HCSP) is presented in [Chaochen et al., 1996]. In the HCSP, communication is extended for continuous variables as well, that is, the value of a continuous variable can be sent along a communication channel. In the HCSP, all interactions between processes take place by means of communication. The χ language is another hybrid extension of CSP. Unlike in the HCSP, in χ, continuous variables can be connected. In this way, the value of a continuous variable can be made known in other processes without explicit communication, which is essential for providing compositionality of continuous or hybrid sub-models. Another important difference is that in χ, communication, time-out and state-event specifications can be combined in a so-called selective waiting statement. Such a statement expresses in a very compact form, that a process waits simultaneously for different kinds of events.

Chapter 3

The χ language

In this chapter the hybrid χ language is described. The name χ is the letter 'h' in the Greek alphabet and it is the first letter in the word hybrid. The syntax is described in BNF notation, accompanied with some semantics. The description of the discrete language elements is in an introductory style. For continuous language elements, we give more elaborate semantics and insight into their intended use. The continuous state calculation is described in detail in the next chapter. The mixed discrete/continuous aspects of the language semantics is treated in Chapter 5.

The discrete language description presented here is based on [Arends, 1996] and [Naumoski and Alberts, 1998], although, some lexical elements have been changed. The description of the continuous language elements is rooted in [Arends, 1996], but it has evolved much in the last few years.

After the basic concepts have been presented, the language is treated from its elementary building blocks, units, dimensions, types, etc. to its highest constructs, processes and systems. Our choice of connection semantics in the χ language is explained by a survey of possible connection semantics in hybrid languages, presented in Section 3.14. At this moment, a limited continuous type concept is implemented in the χ simulator. A more complete concept for units, dimensions and continuous types for the future is presented in Appendix A.

3.1 Basic concepts

A χ specification consists of process and system specifications, and on the top level one instantiation is made of one of these specifications. This top instantiation is the actual χ model that can be executed. The χ model, on the lowest level, consists of a set of process instantiations, that are running concurrently. In χ, concurrency is defined as interleaving sequential execution of the process instantiations. In this thesis, depending on the context, process means process specification when the χ specification is treated, and it means a process instantiation when execution of a model is treated. Process instantiations can be connected via channels. Discrete channels are used for point-to-point synchronous data communication. Continuous

channels connect continuous variables of the process instantiations and establish an equality relation between them. Variables used in a χ specification are either discrete or continuous. Hierarchy is achieved by grouping processes into systems, in which a fixed connection layout of the contained processes and subsystems is specified. The specification allows for both a top-down and bottom-up approach or for any combination of these two. Furthermore, in χ, functions are used for data manipulation.

The process specification is purely continuous-time based or purely discrete-event based, or a combination of the two. The discrete actions are specified in a CSP-like concurrent language [Hoare, 1985]. The continuous-time part is based on Differential Algebraic Equations (DAEs). Simulation of a χ model can be described as an alternating sequence of *discrete* and *continuous phases*[1] . In a discrete phase, multiple events and discrete actions can take place, all at the same time point. In continuous phases, time elapses, and the continuous variables change according to the DAEs.

The χ language is based on a small number of orthogonal language constructs which makes it easy to use and to learn. Where possible, the continuous-time and discrete-event parts of the language are based on similar concepts. The χ language uses LaTeX symbols for specifications which makes specifications compact and, after getting acquainted with the symbols, easy to read. For computer tools, ASCII equivalents are used.

In the χ language there are two scopes, a global and a local one. At the global level, constants, units, dimensions, types, functions, processes and systems are defined. Within the definition of these elements, all variables are local.

3.2 Notation

For the description of the χ language we use the extended Backus-Naur Form (BNF) notation. Each rule consists of a non-terminal on the left side, followed by a '::=' symbol and the syntax rule on the right side. Alternatives are separated by the '|' symbol. Square brackets ([...]) are used for optional elements, while curly brackets ({...}) mean that the enclosed elements can be used zero or more times. For readability, sometimes enumerated nonterminals are used in repeated patterns. For example,

$$TE \quad ::= \quad t_1 \text{ `}\times\text{' } t_2 \text{ `}\times\text{' } t_3 \text{ `}\times\text{' } \ldots \text{ `}\times\text{' } t_n$$

is equivalent to

[1] In this thesis the term 'discrete phase' [Maler et al., 1992] is used instead of *'discrete-event'* to denote the time instant when discrete actions take place. The reason for this is that the term 'event' is interpreted differently by different research communities. In χ for example, events not only include state- and time-events but communications as well. Furthermore, in χ the discrete phase is further divided into sub-phases and, to be precise, an event may evoke a sub-phase rather than a 'discrete-event'.

$$TE \quad ::= \quad t\,\{\text{`}\times\text{'}\,t\}$$

Non-terminals are printed in capital and in italic shape, as in *GW*. Terminals are enclosed in quotation marks, as in ‘,’. Keywords are printed in Sans serif font style, as in ‘syst’.

Accompanying the language syntax, some semantics is explained. For this purpose, semantical sets, denoted by capitals in calligraphic style are introduced. These sets are defined for each χ specification. The set of identifiers is denoted by $\mathcal{I}$. Identifiers in χ are sequences of letters and numbers that start with a letter. The following subsets of $\mathcal{I}$ are defined.

- $\mathcal{I}_{keyword}$ is the set of keywords.
- $\mathcal{I}_{unit}$ is the set of unit identifiers.
- $\mathcal{I}_{dimension}$ is the set of dimension identifiers.
- $\mathcal{I}_{type}$ is the set of type identifiers.
- $\mathcal{I}_{constant}$ is the set of constant identifiers.
- $\mathcal{I}_{function}$ is the set of function identifiers.
- $\mathcal{I}_{procsyst}$ is the set of process and system identifiers.
- $\mathcal{I}_{p,loc}$ is the set of local variable and parameter identifiers of p, where $p \in \mathcal{I}_{function} \cup \mathcal{I}_{procsyst}$. When p is not specified simply $\mathcal{I}_{local}$ is used.

In these sets there are default elements. The set of the default elements is obtained by applying the *def* function. For example, $def(\mathcal{I}_{type})$ is the set of default type names. The rules that hold for the identifier sets are not specified completely yet; more work is needed here. Currently, at least the following rules hold for the identifiers.

- $(\mathcal{I}_{unit} \cup \mathcal{I}_{dimension} \cup \mathcal{I}_{type} \cup \mathcal{I}_{constant} \cup \mathcal{I}_{function} \cup \mathcal{I}_{procsyst} \cup \mathcal{I}_{p,loc}) \cap \mathcal{I}_{keyword} = \emptyset$ for all $p \in \mathcal{I}_{function} \cup \mathcal{I}_{procsyst}$
- $\mathcal{I}_{dimension} \cap \mathcal{I}_{type} \cap \mathcal{I}_{function} = \emptyset$
- $\mathcal{I}_{p,loc} \cap \mathcal{I}_{constant} \cap \mathcal{I}_{function} = \emptyset$ for all $p \in \mathcal{I}_{function} \cup \mathcal{I}_{procsyst}$

In the grammar description expression sets are used. These sets have base elements. Other elements are obtained by applying the corresponding operators on the base elements (the sets are transitiv closures). The following expression sets are defined.

- $\mathcal{E}_{unit}$ is the set of unit expressions.
- $\mathcal{E}_{dimension}$ is the set of dimension expressions.

- $\mathcal{E}_{type}$ is the set of type expressions.
- $\mathcal{E}_{p,loc}$ is the set of expressions that can be defined on the local variables and parameters of the element p, where $p \in \mathcal{I}_{function} \cup \mathcal{I}_{procsyst}$. If p is not specified simply $\mathcal{E}_{local}$ is used.
- $\mathcal{E}_{global}$ is the set of expressions at the global level.

The set of the base elements is obtained by applying the *base* function.

3.3 Units

Units are used for continuous types, to indicate the unit of measurement of the physical quantity that is represented by the type. In χ, there are base units, like kg, s (second) and compound units that are constructed of already defined units with unit operators such as m^3/s. New units, base or compound, can be introduced at the global level with the keyword 'unit'.

UDS ::= 'unit' *UD* {',' *UD*}

Base units

The fundamental SI units, listed in Table 3.1, and the one-unit '-' are the built-in base units. The latter one can be used to measure, for example, pH.

Quantity	Name	Unit
length	meter	m
mass	kilogram	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of matter	mole	mol
amount of light	candela	cd

Table 3.1: Fundamental SI units.

The default unit identifiers are the built-in units: $def(\mathcal{I}_{unit}) =$ {m, kg, s, A, K, mol, cd, -} [2]. New base units are defined with an identifier

UD ::= *i*

[2]The one-unit '-' is an exception to the rule that identifiers have to start with a letter.

where $i \in \mathcal{I}_{unit}$. New base units should be defined only, if the unit to be introduced cannot be expressed as a compound unit, as described below. Fahrenheit is, for example, such a unit. It can be defined as

unit Fahrenheit

Compound units

Compound units are derived from base units using unit operators. A compound unit definition is in the form

$$UD \quad ::= \quad i \text{ `=' } UE$$

where UE denotes a unit expression and $i \in \mathcal{I}_{unit}$. The base elements of the unit expressions are the unit identifiers: $base(\mathcal{E}_{unit}) = \mathcal{I}_{unit}$. The following operations are defined on units.

- multiplication and division as in uv and u/v
- exponentiation as in u^r
- multiplying by scalar as in ru

where $u, v \in \mathcal{E}_{unit}$ and $r \in \mathcal{N}$. The set $\mathcal{N}$ represents the numbers written in χ, and is specified in Section 3.5, on page 26. Syntactically, however, the correct unit expressions are restricted to those that can be derived as

$$\begin{array}{lll} U_0 & ::= & i \mid i^q \mid r\,i \mid r\,i^q \mid \text{`('}\,UE\,\text{`)'} \\ UE & ::= & U_0 \mid UE\;U_0 \mid UE\;\text{`/'}\;U_0 \end{array}$$

where $i \in \mathcal{I}_{unit}$, and $r, q \in \mathcal{N}$. Brackets are used for grouping sub-expressions. This restriction does not allow, for example, one to write unit expressions like $3\,1.2\,12\,\mathrm{kg}^{3^4}$, because such expressions are hard to read and to parse. Examples of compound unit definitions are

unit km = 1000 m
, hour = 3600 s
, sp = km/hour

3.4 Dimensions

Conceptually, what characterizes a continuous type is the *kind* of the physical quantity that it represents: the *dimension*. Examples are length, mass, force, etc. New dimensions, base or compound, can be introduced at the global level with the keyword 'dim'.

$$DDS \quad ::= \quad \text{`dim'} \quad DD \ \{\text{`,'} \ DD\}$$

There are no default dimensions: $def(\mathcal{I}_{dimension}) = \emptyset$.

Base dimensions

The base dimensions introduce a new dimension name associated with a default unit of measurement.

$$DD \quad ::= \quad i \ \text{`='} \ \text{`['} \ u \ \text{`]'}$$

where $i \in \mathcal{I}_{dimension}$ and $u \in \mathcal{E}_{unit}$. For example,

```
dim length  = [km]
,   newtime = [hour]
```

Compound dimensions

Using dimension operators, the relation among different kinds of physical quantities can be expressed at an abstract level. A compound dimension definition is in the form

$$DD \quad ::= \quad i \ \text{`='} \ DE$$

where DE is a dimension expression and $i \in \mathcal{I}_{dimension}$. The base elements of the dimension expressions are the dimension identifiers: $base(\mathcal{E}_{dimension}) = \mathcal{I}_{dimension}$. The operations allowed on dimensions are

- multiplication and division as in $d_1 d_2$ and d_1/d_2
- exponentiation as in d^r

where $d_1, d_2 \in \mathcal{E}_{dimension}$ and $r \in \mathcal{N}$. Similar to compound units, the syntactically correct compound dimensions are derived as

$$\begin{array}{lcl} D_0 & ::= & i \mid i^r \mid \text{`('} \ DE \ \text{`)'} \\ DE & ::= & D_0 \mid DE \ D_0 \mid DE \ \text{`/'} \ D_0 \end{array}$$

where $i \in \mathcal{I}_{dimension}$, and $r \in \mathcal{N}$. Brackets are used for grouping sub-expressions. Examples for compound dimension definitions are

```
dim volume = length^3
,   flow   = volume/newtime
```

At this moment, there is a clear overlap between the unit and the dimension concept; a certain kind of physical quantity can be introduced as a unit only, or as a dimension, or in a mixed way. Although, strictly speaking only the units of measurement are necessary to uniquely characterize a physical quantity, conceptually, dimensions are preferable because they express the relation between physical quantities at a more abstract level. The other advantage of using dimensions is that models can easily be re-scaled by changing the units of dimensions. For example, if the definition of newtime is changed from '**dim** newtime = [hour]' to '**dim** newtime = [s]', then the unit of measurement of flow changes accordingly. A new unit and dimension concept has been developed for the future, and is presented in Appendix A.

3.5 Types

In χ, basic and compound types are used. The basic types are the built-in discrete types, the polymorphic types and the basic continuous types. Compound types, on the other hand, are constructed of built-in and already defined types. In this section, the base and compound types are discussed together with some semantical information. For further semantic rules, readers are referred to [Arends, 1996] and [Naumoski and Alberts, 1998]. New types are introduced at the global level with the keyword '**type**'.

TDS ::= '**type**' *TD* {',' *TD*}

Basic types

The basic types are the built-in discrete types, polymorphic types, and the basic continuous types. The built-in discrete types are:

- bool = {true, false}.
- string is the set of character strings. Elements of it are written between double quotes as in "apple".
- nat = $\mathbb{N}$ is the set of the natural numbers.
- int = $\mathbb{Z}$ is the set of the integer numbers.
- real = $\mathbb{R}$ is the set of the real numbers.
- polymorphic types, that are introduced by the keyword **poly**. In χ, polymorphism is not supported at user level. It is used only in libraries to declare built-in polymorphic functions.
- void = $\emptyset$. Is the empty data type, used in synchronization channels only.

- file is the set of strings.

The default type identifiers are: $def(\mathcal{I}_{type}) = \{\text{bool, nat, int, real, poly, void, file}\}$. Numbers in χ, denoted by the set $\mathcal{N}$, are of type natural, integer or real: $\mathcal{N} = \text{nat} \cup \text{int} \cup \text{real}$.

Continuous types are numerical types, similar to real. A basic continuous type is created by choosing the unit of measurement in which the physical quantity is expressed. This is done by either explicitly defining the unit of measurement, or by using an already defined dimension that automatically has a unit of measurement. A basic continuous type is defined as

$$TD \quad ::= \quad i \text{ `=' `[' } u \text{ `]'} \mid i \text{ `=' } d$$

where $i \in \mathcal{I}_{type}$, $u \in \mathcal{E}_{unit}$ and $d \in \mathcal{E}_{dimension}$. An example is

type flow = [m^3/s]

Compound types

Compound types are constructed from built-in and already defined types. A compound type definition is in the form

$$TD \quad ::= \quad i \text{ `=' } TE$$

where TE is a type expression and $i \in \mathcal{I}_{type}$. The base elements in the set of type expressions are $base(\mathcal{E}_{type}) = \mathcal{I}_{type} \cup \mathcal{E}_{unittype} \cup \mathcal{I}_{dimension}$, where $\mathcal{E}_{unittype} = \{[u] \mid u \in \mathcal{E}_{unit}\}$ (The sets $\mathcal{E}_{unittype}$ and $\mathcal{I}_{dimension}$ are in $\mathcal{E}_{type}$ because basic continuous types can be defined with the aid of them.)

The following compound types can be constructed in χ:

1. type grouping

   $$TE \quad ::= \quad \text{`(' } t \text{ `)'}$$

   where $t \in \mathcal{E}_{type}$. Brackets are used in type expressions to circumvent application of normal operator priority.

2. list type, set type, array type

   $$TE \quad ::= \quad t\text{`*'} \mid t\text{`+'} \mid t^p$$

   where $t \in \mathcal{E}_{type}$, $p \in nat$ and $p > 0$. The list, set and array types are based on a single type, and elements of them are lists, sets and arrays of size n, of type t.

3. tuple type

$$TE \quad ::= \quad t_1 \text{ `}\times\text{' } t_2 \text{ `}\times\text{' } t_3 \text{ `}\times\text{' } \ldots \text{`}\times\text{' } t_n$$

where $t_1, t_2, \ldots t_n, t \in \mathcal{E}_{type}$. Tuples combine n elements ($n \geq 2$), and the j^{th} element ($j = 1 \ldots n$) is of type t_j.

4. named field type

$$TE \quad ::= \quad l \text{ `.' } t$$

where $l \in \mathcal{I} \setminus \mathcal{I}_{type}$ and $t \in \mathcal{E}_{type}$ is a field in a tuple. Named fields associate a name with a field of a tuple as in $id.nat \times count.int$.

5. key type

$$TE \quad ::= \quad \text{`\$' } t$$

where $t \in \mathcal{E}_{type}$ is a field in a tuple. Key types are used in tuples to denote the key fields.

6. function type, distribution type

$$TE \quad ::= \quad \text{`(' } t_1, t_2, t_3, \ldots, t_n \text{`)' `} \longrightarrow \text{' } t \mid \text{`} \longrightarrow \text{' } t$$

where $t_1, t_2, \ldots t_n, t \in \mathcal{E}_{type}$. Functions take n arguments ($n \geq 1$), the j^{th} argument ($j = 1 \ldots n$) being of type t_j, and return an element of type t. Distributions represent the set of statistical distributions with range type t. A number of distributions are predefined in χ, see [Arends, 1996].

7. channel type

$$TE \quad ::= \quad \text{`-' } t$$

where $t \in \mathcal{E}_{type}$. Channel types are created from a single type t. Variables of a channel type can be used to connect processes.

8. channel end-point type

$$TE \quad ::= \quad \text{`!' } t \mid \text{`?' } t \mid \text{`} \sim \text{' } t \mid \text{`} \multimap \text{' } t$$

where $t \in \mathcal{E}_{type}$. Channel end-point types are used in formal parameters of processes and systems. The actual parameters are channels, with the same base type. A parameter of type $!\,t$ denotes the outgoing end-point of a discrete channel, and of type $?\,t$ denotes the incoming end-point. Synchronization channels have no direction; both end-points are of type $\sim t$. Continuous channels have no direction either. Both end-points are of type $\multimap t$.

9. alias type

   $$TE \quad ::= \quad \text{`.'}p$$

   Alias types are used to create an instantiation of a process or system. Here, $p \in \mathcal{I}_{procsyst}$. Alias types can only be used in system specifications.

In χ, a type synonym can be introduced by giving a new name to an existing type. The form is

$$TE \quad ::= \quad l$$

where $l \in \mathcal{I}_{type}$. This construct however, does not introduce a new type because the type system is based on structural equivalence; see below.

A small divergence in the way types are described here compared to [Naumoski and Alberts, 1998], is that bundle types are not handled as separate compound types, but are defined as arrays of channels.

The type system

A type system is a set of rules, that prescribes how to associate objects with types, and whether operations can safely be performed on objects. Type systems ensure the consistency of operators and can detect certain classes of errors, thus increasing the reliability of computer programs [Cardelli, 1996, Cosmo, 1995].

The type system of χ is based on structural equivalence. Type synonyms are not distinguished from each other in any way. Compound types with the same structure and base types are equivalent. For example, given the following definitions

```
type id   = string
,    tup1 = id × int
,    tup2 = string × int
```

id is equivalent to type string, and *tup1* is equivalent to *tup2*. In the χ language, the type of all objects (variables, expressions, sub-expressions, etc.) must be known at compile-time (static type-checking). The χ type system is explained in detail in [Naumoski and Alberts, 1998], where further restrictions on type constructs are introduced.

Continuous types, however, introduce a new concept in the type system: the division between discrete and continuous types. This leads to further restrictions in the type system. Certain type operators can not be applied on discrete types, others are invalid on continuous types. For example, a list of a continuous type is not a valid type, because there is no sensible semantics for such a type. An array of n continuous variables on the other hand, is a valid and useful type. An example for a type operator

that is invalid on discrete types is the continuous channel-end-point operator ($-\circ$). The extension with continuous types requires the clear definition of all operators with respect to their discrete/continuous character. Further research is required here. Some aspects are treated in the proposal for continuous types in Appendix A.

At this moment, a restricted continuous type system is implemented in the χ simulator. Only basic continuous types, channels of basic continuous types, and functions defined on basic continuous types are implemented. The basic continuous types are different from the type real, however, via overloading most binary operators that work with numerical types can be used for mixed real/continuous arguments. Arrays and tuples of continuous types as well as arrays and tuples of continuous channels, although currently not implemented, are used in this thesis in specifications.

3.6 Constants

Constants can be defined at the global level with the keyword '**const** '. Constant definitions have the form

$$\begin{array}{lcl} \mathit{CDS} & ::= & \text{`}\mathsf{const}\text{'} \quad \mathit{CD}\ \{\text{`,'}\ \mathit{CD}\} \\ \mathit{CD} & ::= & i\ \text{`:'}\ t\ \text{`='}\ e \end{array}$$

where $t \in \mathcal{E}_{type}$ and $e \in \mathcal{E}_{global}$ of type t. The base elements in $\mathcal{E}_{global}$ are: $def(\mathcal{E}_{global}) = \mathcal{N} \cup \text{bool} \cup \text{string} \cup def(\mathcal{I}_{function})$. Identifier i is globally defined and has the type t and value e. For example, a small number ε, often used as a numeric error can be defined as

$$\mathsf{const}\ \varepsilon : \text{real} = 10^{-10}$$

It is required that constants can be statically evaluated, i.e., the dependency relation among constants may not contain algebraic loops.

3.7 Variables

All variables used in a χ model are either discrete or continuous. The value of a discrete variable is determined by assignments. Between two subsequent assignments the variable retains its value. Continuous variables, on the other hand, change continuously when time elapses. Their value is determined by the DAEs. An assignment to a continuous variable changes its value only for the current point in time. Variables are used in processes, systems, and in functions. All variables are local, and can be used locally only. The variable sets of processes and systems remain constant during execution. In other words, variables cannot be allocated or de-allocated dynamically.

Discrete variables are declared by using a colon (':'), whereas continuous variables are declared by using a double colon ('::').

$$\begin{array}{lll} \mathit{VDS} & ::= & \mathit{VD}\ \{\text{','}\ \mathit{VD}\} \\ \mathit{VD} & ::= & V\ \text{':'}\ t \mid V\ \text{'::'}\ t \\ V & ::= & u\ \{\text{','}\ v\} \end{array}$$

where $u, v \in \mathcal{I}_{local}$, and $t \in \mathcal{E}_{type}$. The default values in $\mathcal{I}_{local}$ are the local variable and parameter names, and the function names. Variables must have a unique name in the local scope. A continuous variable declaration implicitly declares the time derivative of the variable as well. Given a continuous variable v of type t with unit of measurement u, the time derivative of v, v', has unit of measurement u/s, i.e. u/second.

Continuous variables are further divided into two categories: those that occur differentiated in the equations are the *differential variables*, and those that do not are the *algebraic variables*. This division reflects the physical role of the variables. This role is studied in system theory, where further variable categories are identified (this is not the case in χ). In system theory, differential variables represent so-called *energy variables*. These are mainly conserved quantities, that is, they remain conserved in closed systems. These variables can be divided into two categories. The first category is the generalized *momentum*, such as momentum in mechanical systems, angular momentum in rotational mechanical systems, etc. The second category is the generalized *displacement*. Examples for these kinds of variables, depending on the model domain, are displacement, or angular displacement in rotational mechanical systems, volume in fluid systems, and charge in electrical systems. Differential variables represent a degree of freedom (if they are not dependent on other differential variables; see DAEs in Section 2.2.1); their initial value can freely be chosen. The initial value of algebraic variables, on the other hand, are determined by the equations. Typical algebraic variables are the intensive quantities that characterize systems at points, like temperature or pressure, and the geometric variables that describe the physical measures and configuration of the system. For introduction to equations of dynamic systems in a domain independent unified approach, readers are referred to [Koenig et al., 1967] and [Karnopp et al., 1990].

The simulation time is denoted by the special variable τ. It can be used without declaration, but it cannot be assigned. Its value is an element from the time domain **T**, which is the set of nonnegative real numbers $\mathbb{R}_{\geq 0}$. Initially, $\tau = 0$.

3.8 Expressions

In χ, all expressions have a unique type, though it may depend on the context. For example, the type of an empty list literal '[]', is determined from the context. Not all expressions can be used in all main language elements (processes, systems, functions).

The restrictions are specified when these language elements are discussed. Let nonterminal E denote expressions. $\mathcal{E}_{local}$ is the set of correct expressions in the given scope. $\mathcal{I}_{local}$ is the set of local identifiers. Variables $e, e_1, e_2, \ldots e_n \in \mathcal{E}_{local}$. The following expressions are used in χ.

1. basic expressions

   $$E \quad ::= \quad r \mid \text{true} \mid \text{false} \mid \tau \mid i \mid \text{`"'} \; l \; \text{`"'}$$

   These are numbers ($r \in \mathcal{N}$), the boolean literals true and false, the special variable for time, identifiers ($i \in \mathcal{I}_{local}$), and literal string sequences such as "hello world". Here, l is a sequence of characters and digits, including space and the special newline character '\n'. The base elements in $\mathcal{E}_{local}$ are the base expressions.

2. derivative

   $$E \quad ::= \quad x\text{`}'\text{'}$$

   where x is a continuous variable or an expression that is obtained by applying operators on a continuous variable, e.g., $x.1$ if x is defined as an array, such as $v :: [\text{m}]^2$. Currently, only first order time derivatives are used. Second or higher order derivatives can easily be modelled by using dummy variables. For example,

   $$\begin{aligned} v_1' &= v_2 \\ , \; v_2' &= v_3 \end{aligned}$$

3. list and set expressions

   $$E \quad ::= \quad \text{`['} \; l \; \text{`]'} \mid \text{`\{'} \; l \; \text{`\}'}$$

   where l is a list of expressions: $l = e_1, e_2, \ldots e_n$.

4. tuple and array expressions

   $$E \quad ::= \quad \text{`<'} \; l \; \text{`>'}$$

   where l is a list of expressions: $l = e_1, e_2, \ldots e_n$.

5. expression grouping

   $$E \quad ::= \quad \text{`('} \; e \; \text{`)'}$$

6. function application

$$E \quad ::= \quad e\text{`('}\ e_1, e_2, \ldots, e_n\text{`)'}$$

7. process instantiation

$$E \quad ::= \quad e\text{`('}\ \text{`)'} \mid e\text{`('}\ e_1, e_2, \ldots, e_n\text{`)'}$$

8. index operation (projection)

$$E \quad ::= \quad e_1\ \text{`.'}\ e_2$$

   Index operations identify an element in a tuple or an array.

9. sample expression, pick expression

$$E \quad ::= \quad \text{`sample'}\ e \mid \text{`pick'}\ e$$

   A sample expression takes a sample from a distribution. A pick expression picks an element from a non-empty set.

10. power expression

$$E \quad ::= \quad e_1^{e_2}$$

11. unary minus and plus

$$E \quad ::= \quad \text{`-'}\ e \mid \text{`+'}\ e$$

12. multiplication, division, modulo

$$E \quad ::= \quad e_1\ \text{`}*\text{'}\ e_2 \mid e_1\ \text{`/'}\ e_2 \mid e_1\ \text{`div'}\ e_2 \mid e_1\ \text{`mod'}\ e_2$$

13. addition, subtraction, concatenation

$$E \quad ::= \quad e_1\ \text{`+'}\ e_2 \mid e_1\ \text{`}-\text{'}\ e_2 \mid e_1\ \text{`}+\!\!+\text{'}\ e_2 \mid e_1\ \text{`}--\text{'}\ e_2$$

   The operators '$+\!\!+$' and '$--$' are list operators.

14. boolean tests

$$\begin{array}{lll} E & ::= & e_1\ \text{`}<\text{'}\ e_2 \mid e_1\ \text{`}\leq\text{'}\ e_2 \mid e_1\ \text{`}=\text{'}\ e_2 \mid e_1\ \text{`}\neq\text{'}\ e_2 \\ E & ::= & e_1\ \text{`}\geq\text{'}\ e_2 \mid e_1\ \text{`}>\text{'}\ e_2 \mid e_1\ \text{`in'}\ e_2 \mid e_1\ \text{`sub'}\ e_2 \end{array}$$

   The last two operators are set operators.

15. negation, conjunction, disjunction

$$E \quad ::= \quad \text{`}\neg\text{'}e \mid e_1\ \text{`}\wedge\text{'}\ e_2 \mid e_1\ \text{`}\vee\text{'}\ e_2$$

Further some obvious restrictions on the use of the operators is imposed by the type system as described in [Naumoski and Alberts, 1998]. For example, the base of a power expression must be of a numerical type (nat, int or real) or of a continuous type and the exponent must be of a numerical type. All operators can be applied on continuous variables that can be applied on the type real.

3.9 Processes, Systems, Functions

Processes

A process may have a discrete-event part, a continuous-time part or both. The discrete-event part is a sequential program that describes the process behaviour at discrete times. The continuous-time part consists of a set of DAEs, defined on the continuous variables of the process, and the links that describe the bindings of the continuous channels to the continuous variables. Processes are defined in the form

$$
\begin{array}{lll}
P & ::= & \text{`proc'} \quad i \text{ `(' } [\ PRS\ ] \text{ `)' `='} \\
 & & \text{`|[' } \quad [\ VDS\ ] \\
 & & [\ \text{`}\rightsquigarrow\text{'} \quad S\ ] \\
 & & [\ \text{`}\dashv\text{'} \quad LS\ ] \\
 & & [\ \text{`}\rightleftharpoons\text{'} \quad EDS\ ] \\
 & & [\ \text{`|'} \quad S\ ] \\
 & & \text{`]|'}
\end{array}
$$

where $i \in \mathcal{I}_{procsyst}$, is the process identifier. There is no default process or system identifier: $def(\mathcal{I}_{procsyst}) = \emptyset$. Non-terminal PRS denotes the process parameters. Process parameters can be channels and discrete variables. Syntactically, they have the same form as of variable declarations:

$$PRS \quad ::= \quad VDS$$

The optional blocks of a hybrid process, each introduced with a different leading symbol are:

1. VDS, the block of variable declarations.
2. S, the block of initializations in the form of a sequential program. This block is meant to set up a consistent initial state of the process before the execution of the process starts. At time zero, the values of the continuous variables are calculated from the equations. In order to do that, an initial value may need to be given to some variables. This block is introduced mainly for that purpose. The initialization statements are executed in every process before parallel execution of the processes starts. Therefore, this block cannot contain blocking statements, i.e, event statements.
3. LS, the block of links. This block describes the binding of continuous channels to continuous variables.
4. EDS, the block of equations, containing a set of DAEs that defines the continuous behaviour of the process.

5. S is a statement[3], also called the *discrete body*, that describes the discrete behaviour of the process.

Systems

Systems are composed of process and system instantiations. They are defined in the form

$$\begin{array}{lll} SD & ::= & \text{'syst'} \quad i \text{ '(' } [\ PRS\ ] \text{ ')' '='} \\ & & \text{'|' [' } [\ VDS\ ] \\ & & \text{'|' } I \quad \{\ \text{'}\|\text{'}\ I\ \} \\ & & \text{'] |'} \end{array}$$

where $i \in \mathcal{I}_{procsyst}$ is the system identifier. The system parameters, denoted by PRS, can be channels and discrete variables. The local variables of a system can be channels, and process and system instantiations. The process and system instantiations are composed in parallel with the parallel operator '$\|$'. The instantiations are given their arguments as a list of expressions.

$$I \quad ::= \quad i \mid i \text{ '(' } e_1 \text{ ',' } e_2 \text{',' } \ldots e_n \text{')'}$$

where $i \in \mathcal{I}_{local}$ and $e_1, e_2, \ldots e_n \in \mathcal{E}_{local}$. It is also possible to use instantiations of process and system arrays and compose them with the aid of channel arrays, but this is not described here.

In each χ specification one system or process instantiation is distinguished. This is the experiment, that is executed. Experiments are defined as

$$X \quad ::= \quad \text{'xper' '| [' } VD \text{ '|' } I \text{ '] |'}$$

where VD is a declaration of a variable of alias type, and I is its instantiation.

Functions

Functions are used for data manipulation and operation abstraction. They are defined imperatively, just like the discrete behaviour of processes. Functions are functions in the mathematical sense, i.e., they have no side-effects and they describe a one-to-one mapping. Function calls are non-interruptable, atomic actions. A function is defined as

$$\begin{array}{lll} F & ::= & \text{'func'} \quad i \text{ '(' } [\ PRS\ ] \text{ ')' ' } \longrightarrow \text{ ' } t \text{ '='} \\ & & \text{'| [' } [\ VDS\ ] \\ & & \text{'|' } S \\ & & \text{'] |'} \end{array}$$

[3]The word 'statement', although used in a singular form, may refer to more than one statements. For example, the sequential composition of two statements results in one compound statement.

where $i \in \mathcal{I}_{function}$ is the function's identifier. The parameters of the function, also called the domain of the function are defined in the form of PRS, and the range of the function is denoted by $t \in \mathcal{E}_{type}$. The evaluation of the function is described by the sequential program S. Functions are always called by value. The result of a function is defined by a return statement. Functions cannot contain any blocking statements, or time dependent statements. Functions defined on a continuous domain are similar to those defined on types real. When such a function is called, the current value of the continuous variable is taken as the actual value of the formal parameter, and the operation is carried out on that value.

3.10 Statements

Statements are used in processes to describe their discrete behaviour and their initialization, and in functions to describe data manipulation. Processes and functions contain a sequential program constructed of statements. Let non-terminal S denote statements. Set $\mathcal{S}$ denotes the correct χ statements in a given scope. $def(\mathcal{S}) = \{\text{skip}\}$, where skip is the empty statement. Again, $\mathcal{E}_{local}$ denotes the set of the local expressions. The following kinds of statements and compound statements are defined in χ.

Simple statements

1. empty statement

   S ::= 'skip'

   The execution of the empty statement always succeeds and does not change the state of the process.

2. assignment statement

   S ::= x ':=' e | x '::=' e

   where x is a variable or an expression that can hold a value and $e \in \mathcal{E}_{local}$ is an expression of a type equivalent to the type of x. Assignment statements change the value of variables. There are two kinds of assignment statements. Discrete variables are assigned with the ':=' symbol, whereas differential continuous variables are assigned with the '::=' symbol. The different symbols emphasize the difference between the two kinds of assignments. While a discrete variable retains its new value until the next assignment, an assignment to a continuous variable changes its value only for the current point in time; for further time instants, it is calculated from the equations. Only differential continuous variables can be assigned, since only they can freely be chosen.

Algebraic variables and derivatives of differential variables are calculated from the equations.

There is, however, one exception to this rule. Namely, the derivatives of differential variables may be initialized to their *steady-state* value. A process is in steady-state (also known as static or time-invariant), when none of the process-dependent variables changes over time. Mathematically, this corresponds to having the time derivatives of these variables equal to zero, see e.g. [Edgar and Himmelblau, 1989]. A steady-state initialization can be requested by assigning zero to the time derivative of a differential variable. For example, if variable v is a differential variable, the effect of the assignment $v' ::= 0$ is that the value of v is calculated from the equations such that v' remains zero. Thus, assignment to the time derivative of a differential variable is possible, and has a special meaning. A request for steady-state initialization is valid only for one initial state calculation, namely, for the next initialization of the process. Note, that if v occurs only differentiated in the equations, a steady-state cannot be calculated, because the value of v cannot be calculated from the equations.

3. guess
   In χ only differential variables (and their derivatives) can be assigned. However, there is a practical problem with this. In the χ simulator, a numerical solver calculates a consistent state of the system after discontinuities. The solver uses iterative algorithms that converge only if the system starts from a state that is not too 'bad', i.e., not too far from the required solution. For this reason, often an initial guess for the expected values of the variables is necessary, so that the solver can find the right solution. An initial guess can be assigned to variables using the special symbol '::∼':

   $$S \quad ::= \quad x \text{ '::}\sim\text{' } e$$

   where x is a continuous variable or an expression that can hold a continuous value. Expression $e \in \mathcal{E}_{local}$ has a type equivalent to the type of x. As a result of this statement, the value of x is changed to e, and e is used as an initial guess. The difference between the assignment and an initial guess is as follows. If a continuous variable v is assigned a value e, then in the initial state of the system the value of v is e. On the other hand, if the algebraic variable h is assigned a guess e, then the initial state solver starts its iterative solving process with the value e for h. In the resulting initial state, however, the value of h may be different from e.

   When a steady-state calculation of a differential variable v is requested ($v' ::= 0$), the value of v is calculated. Again, it may be necessary to give an initial guess for the value of v. In this special case, v – which is a differential variable – may be assigned an initial guess.

Note that initial guesses are only necessary for actual simulation, and they do not contribute in any way to system specifications. Initial guesses may be left out of specifications entirely, until the actual simulation.

4. return statement

$$S \quad ::= \quad \text{`}\uparrow\text{'}e$$

where $e \in \mathcal{E}_{local}$. Return statements may be used in functions only. When $\uparrow e$ is executed, the function returns with the value of expression e.

5. I/O statements

$$S \quad ::= \quad f \text{ `?' } x \mid f \text{ `!' } e \mid \text{`?' } x \mid \text{`!'}e$$

where f is a variable that denotes a file, $e \in \mathcal{E}_{local}$ and x is a variable or an expression that can hold a value. The first two forms are the file input/output statements; the last two forms are the standard input/output statements.

6. event statements
Event statements can block the execution of processes until an event happens. There are three kinds of event statements.

   - communication, synchronization

     $$S \quad ::= \quad c \text{ `!' } e \mid c \text{ `?' } x \mid c \text{ `}\sim\text{' }$$

     where c is a variable or an expression that denotes a channel, and x is a variable or expression that can hold a value. Expression $e \in \mathcal{E}_{local}$. Execution of a send action $c\,!\,e$ (or a receive action $c\,?\,x$) in one process causes the process to become blocked until the other process connected to c executes a receive action $c\,?\,x$ (a send action $c\,!\,e$). Subsequently, the value of expression e is assigned to variable x. Synchronization ($c\,\sim$) is a special kind of communication, where no data is exchanged. In this thesis, we do not treat synchronization separately; whenever communication is treated, synchronization is included.

   - time-out statement or so-called delta statement

     $$S \quad ::= \quad \text{`}\Delta\text{' } e$$

     A process executing a statement Δe, where $e \in \mathcal{E}_{local}$ and e is of type real becomes blocked for e time units. The time-out of a blocking delta statement is the time point when the process becomes active again.

   - state-event or so-called nabla statement

     $$S \quad ::= \quad \text{`}\nabla\text{' } e$$

where $e \in \mathcal{E}_{local}$ is a boolean expression. A process executing a statement ∇e becomes blocked until e is true in a consistent state. Execution of nabla statements is further explained in Section 5.3 and an example is presented in Section 5.6.

In χ, instead of saying an event occurred in the system, we say that an event can actually take place. A communication can actually take place if both of the communicating processes have reached a communication statement. A time-out Δs can actually take place if s seconds have passed since program execution reached this statement, and a state-event ∇e can actually take place if the boolean expression e is true in a consistent state of the system.

Compound statements

1. sequential composition

   $$S \quad ::= \quad S_1 \text{ ';' } S_2$$

   where $S_1, S_2 \in \mathcal{S}$. It means, that S_2 is executed after the execution of S_1 is finished.

2. guarded commands: selection statement, selective waiting statement
   Guarded commands have two forms: selection statements and selective waiting statements, each consisting of one or more alternatives. Selection statements have the form

   $$\begin{array}{lll} S & ::= & G \\ G & ::= & \text{'[' } b_1 \text{ ' } \longrightarrow \text{ ' } S_1 \text{ '[]' } b_2 \text{ ' } \longrightarrow \text{ ' } S_2 \text{ '[]' } \ldots \text{'[]' } b_n \text{ ' } \longrightarrow \text{ ' } S_n \text{']'} \end{array}$$

   where $b_i \in \mathcal{E}_{local}$ is a boolean expression and $S_i \in \mathcal{S}$. The boolean expression b_i denotes a *guard*, that is *open* if it evaluates to true, and is *closed* otherwise. Alternatives with open guard are the open alternatives, and those with a close guard are the closed ones. The alternatives of selection statements are composed of a guard, a right arrow symbol and a statement. After evaluating the guards, one statement associated with an open guard is executed. If none of the guards are open, the selection statement is invalid – unless it is repeated. If more guards are open, one alternative is selected non-deterministically.

   Selective waiting statements are used to allow a process to specify a number of events to wait for. They have the form

   $$\begin{array}{lll} G & ::= & \text{'[' } b_1 \text{ ';' } C_1 \text{ ' } \longrightarrow \text{ ' } S_1 \text{ '[]' } b_2 \text{ ';' } C_2 \text{ ' } \longrightarrow \text{ ' } S_2 \text{ '[]' } \ldots \\ & & \text{'[]' } b_n \text{ ';' } C_n \text{ ' } \longrightarrow \text{ ' } S_n \text{']'} \end{array}$$

where $b_i \in \mathcal{E}_{local}$ is a boolean expression, $S_i, C_i \in \mathcal{S}$ and C_i is an event statement. The alternatives consist of a guard, an event statement, a right arrow symbol, and a statement. Alternatives with a communication event are called *communication alternatives*, with a delta statement are called *delta alternatives*, and with a nabla statement are called *nabla alternatives*. An alternative in a selective waiting statement is *enabled* if it is open and the event can actually take place. A selective waiting statement that is not repeated must have at least one open alternative. The process executing a selective waiting statement remains blocked until at least one alternative is enabled. If more alternatives are enabled, the priority rules decide which one to choose. Consecutively, the event of the chosen alternative takes place and its statement is executed. The priority rules to choose an alternative are

(a) If there is an enabled delta alternative with negative time-out, choose this one. If there are more, then choose the one with the smallest time-out. If there are more alternatives with the smallest time-out, choose one non-deterministically.

(b) If there is an enabled communication alternative, choose this one. If there are more, then choose the one waiting for the longest. If there are more alternatives waiting for the longest, choose one non-deterministically.

(c) If there is an enabled nabla alternative, choose this one. If there are more, then choose one non-deterministically.

(d) If there is an enabled time-out alternative, choose this one. If there are more, then choose one non-deterministically.

3. repetition

$$S \quad ::= \quad \text{`*'} G$$

Repetition of guarded statements terminate if none of the alternatives are open. Guards that are always true may be omitted in repeated guarded commands as in $*[\ S_i\ ]$, which is a shorthand for $*[\ \text{true} \longrightarrow S_i\ ]$.

Note, that in both forms of the guarded commands the guards are evaluated only once, at the beginning of the execution of the statement. Therefore, one must be very careful when using continuous variables in guards. In case of selective waiting statements, a guard that is open may become closed while waiting for the event of the alternative. In both forms of the guarded command, a guard may become closed while executing the statement of the chosen alternative. Take, for example, a continuous variable v the value of which is defined by the equation $v' = -1$. Suppose, that the current value of v is 10 when the following selective waiting statement is executed.

$$[\ v > 5\,;\ \Delta 10 \longrightarrow S\ ]$$

where S denotes an arbitrary statement. Since the guard is open, the process waits for 10 seconds, then executes S. However, when S is executed, the value of v is 0, thus the guard is closed. This means, that the guard cannot be taken automatically as assertion when executing the statement of the alternative. Using continuous variables in guards of guarded commands is considered bad modelling practice and is highly discouraged.

3.11 Equations

The continuous behaviour of a process is described by a set of Differential Algebraic Equations (DAEs), defined on the local variables of the process. As introduced earlier, non-terminal *EDS* denotes the equations in a process definition. Equations can have a base form, from which guarded equations can be composed. Base and guarded equations are denoted by non-terminal *EQ*. Substitute equations, a special form of equations, are denoted by non-terminal *SSE*.

$$\begin{array}{lcl} EDS & ::= & ED\ \{\text{`,'}\ ED\} \\ ED & ::= & EQ \mid SSE \end{array}$$

Base equations

$$EQ \quad ::= \quad e_1\ \text{`='}\ e_2$$

where $e_1, e_2 \in \mathcal{E}_{local}$ and they are of numerical types.

Guarded equations

$$\begin{array}{lcl} EQ & ::= & \text{`[']}\ b_1\ \text{`}\longrightarrow\text{'}\ EQ_1\ \{\text{`,'}\ EQ_1\}\ \text{`[]'}\ b_2\ \text{`}\longrightarrow\text{'}\ EQ_2\ \{\text{`,'}\ EQ_2\}\ \text{`[]'}\ \ldots \\ & & \text{`[]'}\ b_n\ \text{`}\longrightarrow\text{'}\ EQ_n\ \{\text{`,'}\ EQ_n\}\ \text{`]'} \end{array}$$

where $b_i \in \mathcal{E}_{local}$ denotes a guard. The guarded equation form is a means to express that depending on the values of the guards, different sets of base equations describe the system. For example, a switch in an electrical circuit can be modelled as

$$\begin{array}{l} [\ b \quad \longrightarrow i = 0 \\ [\!]\ \neg b \longrightarrow v = 0 \\ ] \end{array}$$

where variables i and v denote the current and the voltage respectively, and boolean variable b denotes whether the switch is open.

At any time during the simulation at least one guard must be open. This requirement has to be met at each level of a nested guarded equation. The base equations that

can be reached via open guards in a guarded equation are the *enabled equations*. Base equations not embedded in a guarded equation are always enabled. The number of continuous variables is constant in any given system, therefore, the number of the enabled equations must also be constant. In order to aid the modeller in ensuring that the number of enabled equations is constant, there is an additional requirement for the form of the equations: every alternative of a guarded equation has to define the same number of base equations. For example,

$$\begin{array}{l} [\ b \quad \longrightarrow v_1 = v_{set},\ v_2 = v_1 \\ [\!]\ \neg b \longrightarrow v_1 = 0 \\ ] \end{array}$$

is an incorrect guarded equation, since the first alternative defines two base equations, and the second one defines only one. Furthermore, for reasons of easy implementation, continuous variables cannot change their differential/algebraic category during simulation; a variable defined differentially (algebraically) must remain differential (algebraic). For example, a motor that accelerates, decelerates or runs at constant speed cannot be modelled like

$$\begin{array}{l} [\ sw = \text{“acc”} \quad \longrightarrow v' = 1 \\ [\!]\ sw = \text{“high”} \longrightarrow v = 5 \\ ]\ sw = \text{“dec”} \quad \longrightarrow v' = -1 \end{array}$$

where sw would denote the position of the switch, and v would denote the speed. It is better to model the motor like

$$\begin{array}{l} [\ sw = \text{“acc”} \quad \longrightarrow v' = 1 \\ [\!]\ sw = \text{“high”} \longrightarrow v' = 0 \\ ]\ sw = \text{“dec”} \quad \longrightarrow v' = -1 \end{array}$$

Guarded equations can use continuous variables in their guards, but again, they have to be specified carefully. For example, a motor that accelerates until a certain speed is reached (denoted by v_{set}), then continues with a constant speed may be modelled as

$$\begin{array}{l} [\ v < v_{set} \longrightarrow v' = 1 \\ [\!]\ v \geq v_{set} \longrightarrow v' = 0 \\ ] \end{array}$$

It should be noted, however, that this is an implicit state event specification. The guard of the currently open alternative needs to be monitored. When it becomes false, another alternative is chosen. With continuous variables in the guards, it may be more difficult to ensure that in the course of the simulation there is always an open alternative. At this moment, guarded equations are not implemented as implicit state-event specifications. However, as long as the variables specified by the equation remain continuous, continuous variables can be used in guards. This is further explained in Chapter 7 where the implementation of guarded equations is treated.

Numerics

So far, we have only stated the requirements for the equations and variables that are set by the language. Some of them aim at simple implementation (e.g., variables cannot change their differential/algebraic category during simulation), and may be revised in future.

For a unique solution of the equations to exist, several further numerical requirements have to be met. The requirements regarding the number of equations and variables, is further discussed in the next chapter. Practically, the numerical solvers used in the χ simulator can cope only with certain categories of DAEs. This is discussed in Chapters 7 and 8.

Substitute equations

In χ, there is a special equation form, the substitute equation. Substitute equations specify that a continuous variable and/or its time derivative can be calculated by substitution of an expression. Although called equations, these constructs do not contribute to the set of equations that are considered during simulation and solved by numerical solvers. By using substitute equations, the number of continuous variables and the number of equations that are given to the numerical solvers can be reduced. Substitute equations come in two forms, in a base and in a guarded form.

$$\begin{array}{lcl} SSE & ::= & i \text{ `}\leftarrow\text{' } GSE \mid i\text{`}'\text{' `}\leftarrow\text{' } GSE \\ GSE & ::= & e \mid \text{ `[' } b_1 \text{ `}\longrightarrow\text{' } GSE_1 \text{ `}[]\text{' } b_2 \text{ `}\longrightarrow\text{' } GSE_2 \text{ `}[]\text{' } \ldots \\ & & \quad \text{`}[]\text{' } b_n \text{ `}\longrightarrow\text{' } GSE_n\text{`]'} \end{array}$$

where $i \in \mathcal{I}_{local}$ is a continuous variable, and $e \in \mathcal{E}_{local}$. Identifier i cannot be used in e. Furthermore, all variables in e must be well defined. This means that the time derivative of an algebraic variable may not occur in e. For example, the following equation

$$v \leftarrow e_1 + e_2$$

defines, that the actual value of variable v is equal to expression $e_1 + e_2$. This is equivalent to rewriting the model, and eliminating variable v by writing expression $e_1 + e_2$ wherever v occurs. The variable being substituted is called a *substituted variable*. An example for a guarded substitute equation is

$$\begin{array}{l} v \leftarrow [\ e \geq 0 \longrightarrow \sqrt{e} \\ \quad [\!] \ e < 0 \longrightarrow 0 \\ \quad] \end{array}$$

If only the variable itself is calculated by substitution, and its time derivative is not, then the variable is called a *base substituted variable*. If only the time derivative

of a variable is calculated by substitution, then it is called a *prime substituted variable*. If both the variable and its time derivative are substituted, then the variable is called a *base-prime substituted variable*. Furthermore, the derivation of the differential/algebraic category is the same for substituted variables and for non-substituted variables. Namely, if the time derivative of a variable occurs in the non-substitute equations, then it is a differential variable, otherwise it is an algebraic one.

There are some restrictions on the use of substitute equations and substituted variables. Differential base substituted variables are not allowed in χ, because then the equation set cannot be solved. For more explanation, see Section 7.1. The time derivative of algebraic base substituted variables is not calculated and cannot be used in the discrete-event part of processes. In general, substitute equations may not contain algebraic loops, and the substituted form of a variable may not be assigned.

In the future, variables that can be calculated by substitution should be identified by the simulator by analyzing the equations symbolically. Since this can be done automatically, the primary reason for introducing substitute equations into the χ language is in fact not the reduction of the equation set; substitute equations can be used for other purposes such as to model discontinuities in certain kinds of models in an efficient way and to reduce the index of DAEs. This is explained in sections 7.3.2 and 9.5.

3.12 Links

Modularity is achieved in χ models by parallel composition of process instantiations into systems. Such compositions can define connections between processes. For connections, discrete and continuous channels are used. A continuous channel connects pairs of continuous variables of processes, and defines an equality relation among them. The connection layout is fixed for any given system. Link definitions, denoted by non-terminal *LS*, define the binding of continuous channels to continuous variables. They are of the form

$$\begin{array}{lcl} LS & ::= & L\,\{\,\text{`,'}\,L\} \\ L & ::= & e_1\text{`}{-}\!\circ\text{'}e_2 \end{array}$$

where $e_1 \in \mathcal{E}_{local}$ evaluates to a channel end-point, and $e_2 \in \mathcal{E}_{local}$ evaluates to a local continuous variable. The types of e_1 and e_2 must be equivalent. For example, a channel end-point v is connected to local variable v_1 in process P in the specification

```
proc P(v :: −∘ flow ...) =
| [ v1 :: flow
⋮
⊣ v −∘ v1
⋮
] |
```

Suppose, that the other end of v is connected to variable v_2 in another process. Thus v_1 and v_2 are connected. How equality is ensured between the two variables is a crucial point. It becomes the core of the semantics of hybrid model composition, because in χ, interaction among processes takes place only via channels. Possible ways of ensuring equality, their advantages and disadvantages, and how they would fit into the χ language are discussed in Section 3.14. After studying these options, algebraic equations, identical to those defined in processes have been chosen to be the meaning of the connections. This means, that in the above example, the equation $v_1 = v_2$ is added to the equation set of the model, and is handled just as any other equation.

Note, that connections in χ models are *acausal*, i.e., there is no input/output direction associated to continuous channels. In the graphical representation of the system, for better understanding only, a direction may be associated to continuous channels. It can indicate a physical flow direction, or a controller/controlled relation. This direction, however, is not expressed in the specification, and has no affect on the way variables are calculated. Note, furthermore, that a variable can be connected to more than one channel. In this case, for each channel, one equation is generated.

3.13 Graphical representation

Graphical representation aids understanding the structure of the model. In the graphical representation, systems and processes are denoted by circles. Systems are denoted by circles with additional shading to indicate that they consist of other processes and systems. An example is shown in Figure 3.1. This model consists of a system C and two processes T_1 and T_2. Discrete channels are denoted by arrows (channel a). The arrows are directed from the process that sends data to the process that receives data via the channel. Synchronization channels are represented by dashed lines (channel s). For each continuous channel a line is drawn that optionally ends in an open dot (channels V_T and Q). They are regarded as special kinds of arrows with an open dot head. If a physical flow direction can be associated to the channel, then the 'arrow' points in the flow direction. If no physical flow direction can be associated to the channel, then the 'arrow' points from the process that determines the value of the quantity represented by the channel, (usually the controlled, or physical process) to the process that reads or uses the quantity (usually the controller).

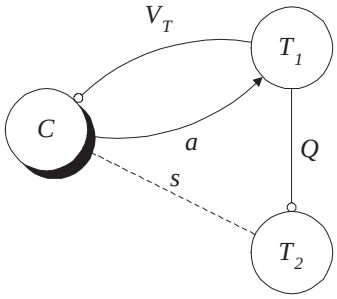

Figure 3.1: Example for graphical representation.

3.14 Semantics of model composition in hybrid languages

In this section, the different model composition semantics used in hybrid languages are surveyed. Wherever possible, we mention some simulation languages that employ the given approach. At the same time, we evaluate how these semantics would fit into the χ language. In order to do this, we first examine how the connection semantics can influence model behaviour in χ.

Model composition

Most hybrid languages support hierarchical modelling. Model hierarchy can reflect the physical hierarchy of the system, and can also be used to introduce different levels of abstraction. A hierarchical structure facilitates the process of modelling in many ways. It supports both top-down and bottom-up development. Components can be tested individually and their behaviour can be studied in different environments, which promotes the development of libraries and standardization in different application areas [Mattson, 1988].

Model composition in hybrid languages is realized by connecting (among others) continuous variables of sub-models. For this purpose, connection elements, so-called *streams* (gPROMS [Barton, 1992], ASCEND [Piela et al., 1991, ASCEND, 1997]), *terminals* (Dymola [Elmqvist et al., 1993, Elmqvist et al., 1996], Omola [Mattson et al., 1993]), or *channels* (χ) are introduced in the interface specification of the sub-models. The variables are connected by means of these elements. In most languages, a connection imposes an equality constraint on the variables. However, this is not the only meaning that can be associated with a connection. Depending on the physical quantity that the variables represent and the architecture of the physical system, continuous variables can be divided into two classes: *across variables*, that become equal, and *through variables*, that sum to zero, when connected [Koenig et al., 1967]. In this section, we concentrate on the semantics of connections that impose equality. We therefore consider across variables. Through

variables can be considered in a similar way. Please note that in languages that do not have specific language constructs for through variables, connections of through variables are modelled by means of across variables and an additional explicit equation that sets their sum to zero.

Simultaneous read/write access

Specification of discontinuities in hybrid models is an important facility, provided by hybrid languages. In the χ language, a discontinuity of a variable can be modelled explicitly by assignments, or implicitly, by altering the equation(s) that determine its value. Equations can be altered by changing variables that are used in the equations themselves or in the guards of guarded equations. Whether an implicit change to a variable (by changing the equations) results in immediate change to its value, depends on how the model is brought into a consistent state, i.e., into a state in which all variables satisfy the equations.

Processes can simultaneously access their connected variables in χ. Let us examine the situations that may arise in χ models when read or write actions are requested simultaneously, if simultaneous access is not controlled. Suppose, furthermore, that a change to a connected variable immediately changes the value of the connected variables in other processes.

If a variable is changed in one process from an old to a new value and the connected variable is read simultaneously in another process, the read value can either be the old or the new value. Depending on whether the process that changes its variable (write action), or the process that uses its variable (read action) is scheduled first, the new or the old value is stored. An additional complication may arise if the variable is used in guards of guarded commands. Now, there are two concerns.

First, the choice of the alternative may depend on actual process scheduling. Namely, if the old value is used for evaluating the guard, and the new value falsifies some open guards, the choice may depend on whether guards are evaluated with the old or with the new value.

Second, it is possible that after choosing an alternative the guard of which evaluates to true using the old value, the other, writing process is scheduled. It may change the connected variable in such a way, that the guard is falsified. Now, when continuing the execution of the statement of the alternative, the guard is not true anymore. This is illustrated in the following example. There are two processes P_1 and P_2, and their variables s_1 and s_2 are connected. The value of s_1 changes periodically with a period of 4 seconds. In the first 3 seconds of each period, s_1 is 0 and in the remaining 1 second it is 1. Process P_1 is as follows

$$\mathsf{proc}\ P_1(s :: \multimap [-]) =$$

$$
\begin{array}{l}
\mid [\ s_1 :: [-] \\
\sim\mid\ s_1 ::= 0 \\
\dashv\mid\ s \multimap s_1 \\
=\mid\ s_1' = 0 \\
\quad \mid *[\ \Delta\, 3;\ s_1 ::= 1;\ \Delta\, 1;\ s_1 ::= 0\ ] \\
]\ \mid
\end{array}
$$

Process P_2 stores the value of s_2 at every second. The values are sorted into two lists, according to whether they are bigger than 0.5 or not.

$$
\begin{array}{l}
\mathsf{proc}\ P_2(s :: \multimap [-]) = \\
\mid [\ \ s_2 :: [-],\ n : \mathrm{int},\ xl, xh : \mathrm{real}^* \\
\dashv\ \ s \multimap s_2 \\
\quad \mid *[\ \Delta\, 1;\ [\ s_2 > 0.5 \longrightarrow n := n + 1;\ xh := xh +\!\!+ [s_2] \\
\qquad\qquad [\!] \ s_2 \leq 0.5 \longrightarrow n := n + 1;\ xl := xl +\!\!+ [s_2] \\
\qquad\qquad] \\
\quad\ ] \\
]\ \mid
\end{array}
$$

$$
\begin{array}{l}
\mathsf{syst}\ S() = \\
\mid [\ p_1 : .P_1,\ p_2 : .P_2,\ s :: -[-] \\
\quad \mid p_1(s) \parallel p_2(s) \\
]\ \mid
\end{array}
$$

If after 3 seconds P_1 is scheduled first, the assignment to s_1 results in the immediate change of s_2, therefore the first alternative ($s_2 > 5$) is chosen. (If P_2 is scheduled first, then the second alternative is chosen.) Furthermore, if the concurrent assignment of s_1 ($s_1 ::= 1$) in P_1 changes the value of s_2 between the evaluation of the guards and the append action, then the second alternative ($s_2 \leq 0.5$) is chosen in P_2, yet the value 1 is appended to list xl. Obviously, this is not the desired behaviour of P_2, thus this specification is not correct without access control. Such behaviour could be prevented by considering the evaluation of the guards and execution of the longest prefix of the selected statement that does not contain event statements as a non-interruptable atomic action. This means, that after an alternative is chosen, execution of its statement cannot be interrupted, until an event statement is reached. Thus, the guard remains true while executing non-event statements. Yet, while complicating the semantics of parallel execution of processes, this approach does not remove the scheduling dependency.

If both connected variables are changed simultaneously (simultaneous write) without access control, the final values of the variables depend on the scheduling order as well.

How a language should handle the above situations depends on the purpose of the language, on the amount of control effort that may be put into control of simultaneous access of connected variables, and on the language philosophy. For example, if physical systems are modelled with some non-determinism, scheduling dependency

can be acceptable. On the other hand, such non-determinism may not be acceptable if mission critical processes are modelled. There are several possibilities to handle simultaneous accesses. Some of them are:

- The outcome of simultaneous read/write accesses of connected variables is non-deterministic. In this case, the modeller must ensure that undesired non-deterministic situations such as the one in the above example do not occur.
- The language provides mutual access control primitives, such as semaphores, synchronization primitives, etc., so that undesired non-determinism can be prevented. It is not necessary to require access control on each connected variable, so that if there is no danger of scheduling dependency, or non-determinism is intended, connected variables may be left without access control.
- Ambiguity of mutual access is resolved by other means. For example, mutual access is not allowed, see e.g., in gPROMS [Barton, 1992, page 109], or the outcome of a mutual access is deterministic.

In the χ language, to reduce unnecessary access control efforts, multiple read/write accesses to connected variables are allowed. However, to remove scheduling dependencies and to preserve the guards in guarded commands as postcondition during the execution of alternatives, a change to a local variable does not change connected variables immediately. Thus, the system can become temporarily inconsistent. The way it is brought into a consistent state is described in the next two chapters. For simplicity, no extra control primitives are introduced, because for this purpose synchronization via discrete channels can be used.

Connections

Four possible ways of connecting continuous variables are examined: shared variable, causal connection, master object, and algebraic equation. Whenever possible, we mention the languages that use the given semantics. However, it is not our aim to detail the consequences of a certain semantics in a given language. In many cases, there is not enough documentation available to do this. From the available documentation it appears that often the way certain language constructs can be used and combined may be restricted in order to decrease the chance of erroneous or ambiguous models. In gPROMS, for example, two sub-models cannot manipulate the same connection equation simultaneously.

1. *shared variable*
   Connected variables become a single entity shared by the sub-models. This implies that changes to a variable in one sub-model are immediately visible in the other sub-model. This approach is implemented in ASCEND, if streams are connected with the aid of the ARE_THE_SAME operator. Using shared

variables has the disadvantage that the outcome of simultaneous access to shared variables depends on the actual scheduling. Furthermore, as the previous example illustrates, the semantics of the conditional language constructs may be affected as well. Therefore, in languages that employ this connection semantics, either access control primitives are provided, or the access of shared variables is restricted to avoid ambiguity. It should be noted that access control primitives used in parallel programming languages (semaphores, messages) would eliminate the simplicity gained by merging the variables, and can become an unacceptable burden in models of industrial systems. An example for linear hybrid automata that use shared variables can be found in [Alur et al., 1996], where synchronization labels can be used to control access to shared variables, and invariants assigned to control locations can be used to ensure system consistency.

2. *causal connection*
   Connections define a causal interaction between the subsystems, where in each connection one side acts as input and the other side as output. A disadvantage of this approach is that the direction of the information flow is fixed for each component. This restricts re-use of a component in other environments. Another restriction is that the connections should not lead to algebraic loops. In order to satisfy this requirement, the model may need to be transformed. General purpose simulation packages like ACSL [Mitchell and Gauthier, 1976] and SIMULINK [Simulink, 1993] use this approach. These languages require the DAEs of each subsystem are written in explicit state-space form

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}) \\ \mathbf{y} &= \mathbf{g}(\mathbf{x}, \mathbf{u}) \end{aligned}$$

   where $\mathbf{u}$ is the input and $\mathbf{y}$ is the output variable. Transformation into explicit state-space form (if possible) requires extra engineering skills and the resulting equations often do not reflect the natural system structure. With the advent of DAE solvers that can solve the implicit state space form, hybrid languages do not need to employ causal relations any more [Elmqvist and Mattson, 1997, Cellier and Elmqvist, 1993].

3. *master object*
   From the model's topology, first the sets of connected variables that are equal, are identified. For each set, a master object is introduced, which holds the value of the connection. This value is distributed among the variables at the end of each continuous phase. During the discrete-event phase, the sub-models have access to their (local) variables only. Changes of these variables may lead to a temporal inconsistency in the system. The system is brought back to consistency, by updating the master object, and distributing its new value among the variables. This idea is inspired by distributed systems, where several processes share a common memory. In order to allow concurrent accesses to the

same data, duplicate copies are made of the shared block (i.e., block replication). For data coherence control, one copy is designated as a master copy, the value of which is distributed among the other copies. The difficulty in hybrid systems is to find a suitable protocol that determines the value of the master object. The mechanism has to be simple and intuitive, such that it is suitable for large and complex systems. A possible choice could be to assign the master object each time one of the connected variables is assigned. In this way, the master object always holds the last change that has been made to the connection. Its value could be distributed only at the end of the discrete-event. However, the value of the master object would depend on the process scheduling. Another difficulty with using a master object is that it can hardly be generalized for connections of through variables.

4. *algebraic equation*
   Connections define an additional algebraic equation between the connected variables. This is the case in gPROMS, Dymola and in χ. In this connection semantics, it needs to be specified when the equations are valid, and how equality is ensured after a discontinuous change to connected variables. For reasons of simplicity, it is preferable to define the semantics of connection equations identical to other equations. Note that in the implementation, connection equations may be eliminated and replaced by a shared variable at a lower level. Care should be taken that such an implementation satisfies the equation semantics, and does not for example change the semantics to that of a shared variable.

Chapter 4

Continuous state calculation

The *continuous state* of a χ model is a concrete evaluation of the continuous variables. This state is determined by the set of the valid DAEs, and the state-events monitored. At any point during the simulation the system is in a certain continuous state that is not necessarily consistent, i.e., the continuous variables may not satisfy the set of the valid equations. This is the case when variables occurring in equations are assigned. The continuous state may also depend on discrete variables, since discrete variables can be used in equations and in state-event conditions.

The language semantics describes how the valid set of the DAEs is determined, how the values of the continuous variables are calculated from the DAEs, and when state-events occur. It has to be taken into consideration that the discrete state components interact with the continuous state components: continuous variables can be assigned, and their values can be used in discrete-event actions.

In this chapter, the mathematical model of the continuous state calculations is presented. The reason for using a mathematical model is that a formal hybrid semantics is not available yet. Also, a mathematical model gives more explicit guidelines for the implementation of the simulator, and it provides users with an intuitive insight into the evolution of the continuous state.

4.1 Mathematical model

The problem solved by hybrid simulators can be formulated as

$$\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t) = \mathbf{0}, \tag{4.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is a vector of differential variables, $\mathbf{y} \in \mathbb{R}^m$ is a vector of algebraic variables, $t \in \mathbb{R}$ is the independent variable and $\mathbf{f} \in \mathbb{R}^{2n+m+1} \rightarrow \mathbb{R}^{n+m}$ is the set of DAEs as treated in [Barton and Pantelides, 1994]. The time domain is divided into an alternating sequence of discrete and continuous phases.

Let I_k denote the time interval of the k^{th} continuous phase ($k \geq 1$) and t_{k-1} and t_k be its boundaries: $I_k = [t_{k-1}, t_k]$. In this domain the equations have the form

$\mathbf{f}^{(k)} \in \mathbb{R}^{2n+m+1} \rightarrow \mathbb{R}^{n+m}$. Equation (4.1) in the k^{th} continuous phase takes the form

$$\mathbf{f}^{(k)}(\dot{\mathbf{x}}^{(k)}, \mathbf{x}^{(k)}, \mathbf{y}^{(k)}, t) = \mathbf{0} \quad t \in [t_{k-1}, t_k]. \tag{4.2}$$

It is assumed, that the number of variables remains the same in all continuous phases. Denoting the dimension of $\mathbf{x}$ and $\mathbf{y}$ in the k^{th} phase by n_k and m_k respectively, ($\mathbf{x} \in \mathbb{R}^{n_k}$ and $\mathbf{y} \in \mathbb{R}^{m_k}$), $n_k = n$ and $m_k = m$ for all $k \geq 1$. The solution to (4.2) $(\mathbf{x}^{(k)}, \mathbf{y}^{(k)})$ is the value of the continuous variables over I_k.

At time point t_k a discrete phase starts. It can be triggered by a state-event or a time-out occurring in any of the processes, i.e., a state-event condition becomes true or a time-out is finished. In the discrete phase the discrete body of the processes are executed. If variables (discrete or continuous) that occur in the equations are assigned, the equations may become inconsistent and a consistent initial state may need to be re-established. In fact, within a single discrete phase a series of initial state calculations can take place. In order to incorporate this into the above model, the discrete phase at time t_k is divided into $j_k \in \mathbb{N}$ sub-phases. Each sub-phase is enclosed between two initial state calculations as shown in Figure 4.1.

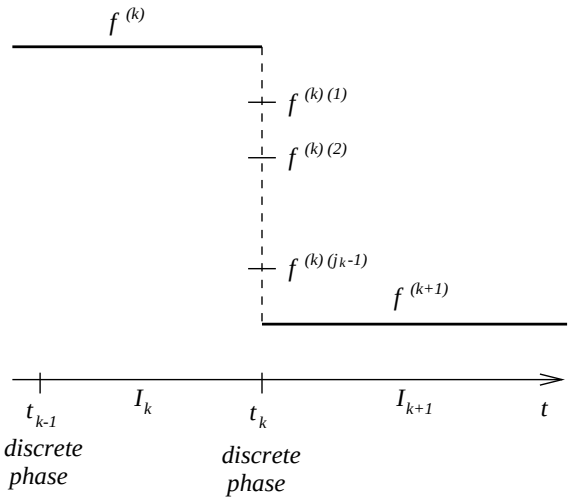

Figure 4.1: Continuous state calculations in χ.

4.1.1 Initial state problem

In the k^{th} discrete phase, j_k initial state calculations take place. At the end of the i^{th} sub-phase ($i \in [0, j_k - 1]$) the following equation is solved,

$$\mathbf{f}^{(k)(i+1)}(\dot{\mathbf{x}}^{(k)(i+1)}, \mathbf{x}^{(k)(i+1)}, \mathbf{y}^{(k)(i+1)}, t_k) = \mathbf{0}. \tag{4.3}$$

The new state $(\dot{\mathbf{x}}^{(k)(i+1)}, \mathbf{x}^{(k)(i+1)}, \mathbf{y}^{(k)(i+1)})$ has $2n + m$ components; to be able to determine the state uniquely $2n + m$ equations are necessary. According to [Barton and Pantelides, 1994], these equations are in the general form

$$\mathbf{C}(\dot{\mathbf{x}}^{(k)(i)}, \mathbf{x}^{(k)(i)}, \mathbf{y}^{(k)(i)}, \dot{\mathbf{x}}^{(k)(i+1)}, \mathbf{x}^{(k)(i+1)}, \mathbf{y}^{(k)(i+1)}, t_k) = \mathbf{0}, \tag{4.4}$$

where $\mathbf{C} \in \mathbb{R}^{4n+2m+1} \rightarrow \mathbb{R}^{2n+m}$. From the $2n+m$ equations $n+m$ are from $\mathbf{f}^{(k)(i+1)}$. The remaining n equations are mostly determined by the default or the alternative continuity assumptions.

4.1.2 Default continuity assumption

Differential variables typically represent conserved quantities like mass, energy, entropy or other quantities that are closely related to conserved ones. These quantities are conserved in closed physical systems; only external actions change them. Furthermore, these variables are, in a first approximation, modelled as continuous. Mostly, their initial value can be freely chosen, so that they represent freedom degrees. The value of a differential variable cannot be freely chosen if it is a dependent variable and/or there are hidden constraints in the equations. These cases are treated separately in sections 7.2.5 and 9.5.

The values of the differential variables can be freely chosen in χ by assignments using the '::=' symbol, as in $v ::= 2$. By default, their value is taken to be continuous across sub-phase boundaries. Let $\mathbf{x}^{(k)(i)-}$ and $\mathbf{x}^{(k)(i)+}$ denote the value of $\mathbf{x}$ at the beginning and at the end of the i^{th} sub-phase, respectively. The default continuity assumption is that for all x_l, $l \in [1, n]$)

$$x_l^{(k)(i)+} = x_l^{(k)(i+1)-}. \tag{4.5}$$

In other words, when solving (4.3) at the end of the i^{th} sub-phase $\mathbf{x}$ is constant (equal to $\mathbf{x}^{(k)(i)+}$).

The initial state calculation starts with the identification of the valid set of DAEs i.e., $\mathbf{f}^{(k)(i+1)}$. If not requested otherwise, the default continuity assumption holds for all differential variables. Therefore, the equations are solved for the algebraic variables $\mathbf{y}^{(k)(i+1)}$ and the time derivatives of the differential variables $\dot{\mathbf{x}}^{(k)(i+1)}$. The values of $\mathbf{y}$ and $\dot{\mathbf{x}}$ at the end of the i^{th} sub-phase ($\mathbf{y}^{(k)(i)+}$, $\dot{\mathbf{x}}^{(k)(i)+}$) are taken as initial guesses. At the end of the last sub-phase $\mathbf{f}^{(k)(j_k)} = \mathbf{f}^{(k+1)}$ is calculated.

Note, that in χ the continuity assumption of differential variables (4.5) does not preclude the possibility of modelling instantaneous change to them. In most models such a change denotes an interaction from the model environment. For example, some substance is added to a chemical process or charge is added to an electrical circuit by closing a switch. This can be modelled by assigning a component x_l ($l \in [1 \ldots n]$) during the i^{th} discrete sub-phase. Then, $x_l^{(k)(i)+}$ holds the assigned value. In this way, the discontinuity occurs during the sub-phase and the differential variable remains continuous across the sub-phase boundary. In Section 5.5 an example is given for this.

4.1.3 Alternative continuity assumption

In most models of industrial systems the default continuity assumption (4.5) together with the set of valid DAEs can uniquely define the state in the next sub-phase. However, there are cases when the default continuity assumption does not hold for some of the differential variables. The most common situation is when a process is initialized to its steady-state.

Steady-state initialization is possible in χ by using the alternative continuity assumption. When the alternative continuity assumption holds for a continuous variable, its derivative is taken to be continuous, rather than its value . For such a variable x_l, ($l \in [1, n]$)

$$\dot{x}_l^{(k)(i)+} = \dot{x}_l^{(k)(i+1)-} \tag{4.6}$$

holds instead of the corresponding l^{th} equation in (4.5). Therefore, when solving (4.3) $\dot{x}_l$ is taken to be constant and x_l is calculated. Steady-state initialization of a differential variable can be requested in χ by an assignment of 0 to its time derivative (for example, $v' ::= 0$). Such a request is considered only for one (i.e. for the next) initial state calculation. Recall, that normally only an initial guess can be given to the derivative of a differential variable.

4.1.4 Initial state calculation

To summarize, when the initial state is calculated, if the default continuity assumption holds for a differential variable, then its time derivative is calculated, otherwise, if the alternative continuity assumption holds then the variable itself is calculated. Algebraic variables are always calculated.

Mathematically: let $\nu \in \mathcal{P}\{1, \ldots, n\}$ denote an index set such that $i \in \nu$ iff $\dot{x}_i$ is assigned in the last sub-phase. Let $\overline{\nu}$ denote the complement of ν: $\overline{\nu} = \{1, 2, \ldots n\} \setminus \nu$. Furthermore, let $\mathbf{x}_\nu$ denote those components of $\mathbf{x}$ for which $j \in \nu$. When the initial state is calculated, (4.3) is solved for $\dot{\mathbf{x}}_{\overline{\nu}}^{(k)(i+1)}$, $\mathbf{x}_\nu^{(k)(i+1)}$, and $\mathbf{y}^{(k)(i+1)}$, while $\mathbf{x}_{\overline{\nu}}$ and $\dot{\mathbf{x}}_\nu$ are taken to be known (they are equal to $\mathbf{x}_{\overline{\nu}}^{(k)(i)+}$ and $\dot{\mathbf{x}}_\nu^{(k)(i)+}$).

4.2 Example for steady-state initialization

The simple example depicted in Figure 4.2 illustrates steady-state initialization in χ. A buffer tank is filled by a constant incoming flow Q_{set} and at the same time liquid is leaving through an outlet Q_o. The outgoing flow is determined by the height of the liquid h; $Q_o = k\sqrt{h}$, where k is a constant. The cross-sectional area of the tank is denoted by A.

The volume of the liquid in the tank is determined by $V' = Q_{set} - Q_o = Q_{set} - k\sqrt{h}$. In the steady-state, the flow rate out of the tank must equal to the flow rate into the

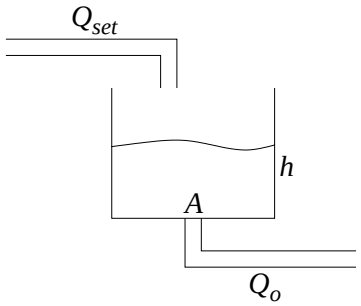

Figure 4.2: Buffer tank.

tank i.e., $V' = 0$. The steady-state initialization of the buffer tank is modelled in χ in the following way:

```
proc P(Q_set, A, k : real)=
| [ h :: [m], Q_o :: [m^3/s], V :: [m^3]
~| V' ::= 0
=| V' = Q_set - Q_o
 , Q_o = k√h
 , V = Ah
] |
```

When V' is set to zero ($V' ::= 0$), the value of V – as well as h and Q_o – are calculated from the equations. $V' = 0$ leads to $Q_{set} = k\sqrt{h}$, which yields the analytical solution

$$\begin{aligned} h &= \left(\frac{Q_{set}}{k}\right)^2 \\ Q_o &= k\sqrt{h} \\ V &= Ah. \end{aligned}$$

During simulation, the value of V, h and Q_o are approximated by a numerical solver.

4.3 Non-continuous phase shift

There are situations when a physical system shifts between operating modes in a way that for certain differential variables neither the default nor the alternative continuity assumption holds. These cases are briefly introduced here, although the current χ language specification does not support direct modeling of such mode shifts. This section is intended to introduce some of the problems that have to be solved in the future in order to incorporate modeling of non-continuous mode shifts in χ.

One such a situation is when the valid set of DAEs contain a hidden consistency relation, that makes some of the continuity equations redundant or

invalid. Namely, differentiating Equation (4.3) may reveal further constraints on $\dot{\mathbf{x}}^{(k)(i+1)}, \mathbf{x}^{(k)(i+1)}, \mathbf{y}^{(k)(i+1)}$. Pantelides presents an algorithm in [Pantelides, 1988] to locate those subsets of the equations in (4.3) that need to be differentiated in order to obtain these constraints. To implement this algorithm as well as the subsequent differentiation of the identified equations in the χ simulator, the equations need to be represented symbolically. This is not the case yet. However, the hidden constraints can be made explicit by the user, using substitute equations. In this case, the model can be simulated with the current χ simulator. This is illustrated in Section 9.5.

Another situation is when a significant change in the underlying physical system occurs just when the system shifts from one sub-phase to another. This is usually a result of high abstraction either in the time scale (*time scale abstraction*) or in the physical parameters (*parameter abstraction*) of the system [Mosterman and Biswas, 1996]. Note that in this case, the change in the system is not a result of interaction with the outside world. Rather, it is rooted in the physics of the system itself. An example of this is when two balls of mass m_1 and m_2 with speed v_1 and v_2 of opposite direction collide. Consider, for the sake of simplicity, a perfectly elastic collision, where no energy is lost. In the course of the collision the kinetic energy is saved momentarily by elastic effects, that return the energy back to the balls after the collision. The time scale of the collision is much smaller than that of primary interest, therefore, it is often modelled as an instantaneous change in the kinetics of the balls. Suppose that the two balls are set in motion by forces F_1 and F_2. The following equations describe their kinetics:

$$\begin{aligned} F_1 &= m_1 a_1 \\ a_1 &= \dot{v}_1 \\ F_2 &= m_2 a_2 \\ a_2 &= \dot{v}_2 \end{aligned}$$

Variables v_1 and v_2 are differential and normally remain continuous across sub-phase boundaries. When a collision occurs, however, the speed of the balls is determined according to the conservation laws of the momentum and the kinetic energy. Therefore, the default continuity assumption (4.5) is replaced by the following two equations:

$$m_1 v_1^- + m_2 v_2^- = m_1 v_1^+ + m_2 v_2^+ \tag{4.7a}$$

$$\frac{1}{2} m_1 {v_1^-}^2 + \frac{1}{2} m_2 {v_2^-}^2 = \frac{1}{2} m_1 {v_1^+}^2 + \frac{1}{2} m_2 {v_2^+}^2. \tag{4.7b}$$

Currently, it is not possible to change the default continuity assumption (4.5) explicitly in the χ language other than to the alternative continuity assumption (4.6). Explicit assignments to continuous variables and steady-state calculation proved to be sufficient for a large class of discontinuities. Even the above example of colliding balls can be modelled with current techniques by calculating the values of v_1 and v_2 after the collision and assigning v_1 and v_2 explicitly.

Still, in the future it might prove to be necessary to introduce a new language element that would replace some of the continuity equations in (4.5) with another set of equations that are considered only during the initial state calculation. Such an equation is called an *instantaneous equation*. Instantaneous equations would, however, require introduction of further language constructs. Namely, in (4.7) the values v_1^-, v_1^+, v_2^-, and v_2^+ would need to be referenced.

Chapter 5

Hybrid scheduling

The crucial point in hybrid language development is the smooth integration of the discrete and the continuous semantics. The purely discrete and the purely continuous semantics have to be preserved as far as possible, so that the language remains suitable to model pure systems. Furthermore, to keep the language consistent, those aspects that are present both in the discrete as well as in the continuous part of the language should fit into the same language philosophy. For example, in case of the χ language, the value of a discrete variable of one process can be made known in another process only by communication via a discrete channel. Sharing continuous variables among processes would not fit into the language philosophy for two reasons. First, there are no shared discrete variables. Second, a continuous variable could be shared by more processes, which would lead to a broadcasting mechanism, whereas discrete communication happens on a peer-to-peer bases.

The discrete and the continuous semantics are integrated by specifying the semantics of interactions and the evolution of the discrete and the continuous states relative to each other. Interactions are possible by *a)* assignments of continuous variables, *b)* using the values of continuous variables in the discrete-event body, *c)* using discrete variables in the equations, and *d)* using state-events. In this chapter the evolution of the discrete and the continuous states is specified relative to each other. The beginning and the end of the discrete phases have already been specified: a discrete phase starts when an event occurs, and lasts until all processes are blocked in a consistent continuous state, in the sense that no further discrete actions can take place before a continuous phase has elapsed. Between two adjacent discrete phases a continuous phase takes place. What is left, is to define the moment when the initial state is calculated, by taking into consideration the discrete state evaluation.

5.1 Discrete state

The *discrete state* of a χ model is defined by the discrete variables, the simulation time and the current execution points in the processes. The current execution point can be thought of as a pointer that points to a statement that is going to be executed

next. Execution points need to be taken into account in order to be able to uniquely define the state of the system in a discrete phase. Recall that the discrete behaviour specification of χ processes can contain complex control structures. Sequential, conditional and repetitive composition of base constructs are possible. Also, selective waiting statements are complex control structures. In order to be able to describe the behaviour of χ models during a discrete phase, the execution points in individual processes has become part of the discrete state.

The discrete state of a system is determined by the discrete language semantics that defines the meaning of the discrete body of the processes. In other words, we call that minimum part of the language semantics that completely describes the behaviour of a purely discrete model the discrete semantics. In this thesis the discrete semantics is not treated in detail. The discrete-event process scheduling in the χ engine is outlined in order to be able to understand the timing of hybrid models. A formal treatment of the discrete semantics by means of SOS-rules (structured operational semantics) of Process Algebra has recently been completed [Bos and Kleijn, 1999]. A previous work defines the discrete operational semantics of χ by associating possible executions with specifications. It defines the state of the model as the set of the states of all process instantiations. The state of a process instantiation is a 4-tuple, representing the statement to be executed next (execution point), a mapping between variables and values, a real number representing the process time, and a function representing the number of communications realized over the channels [van de Mortel-Fronczak, 1995]. The current discrete-event engine implementation is based on this semantics [Naumoski and Alberts, 1998]. The only difference is that instead of letting each process having its own local time, a global time is introduced. The latter change is also necessary to be able to calculate the values of the continuous variables in hybrid processes.

5.2 Discrete-event scheduling

In this section, the discrete-event scheduling of the χ engine is outlined as described in [Naumoski and Alberts, 1998]. The heart of the engine is a scheduler, which is an abstract machine. It schedules processes for execution and updates the simulation time.

A χ model consists of a set of process instantiations that are running concurrently. As a first step in the simulation, all identifiers of the model (process names, variable names, etc.) are uniquely renamed. In this section, process instantiations are referred to simply as processes. The following definitions are used:

- Y is the set of the processes of the model.
- $\mathcal{I}$ is the set of identifiers that occur in the model.
- $\mathcal{I}_{process} \subseteq \mathcal{I}$ is the set of the process identifiers.

- $\mathcal{I}_{channel} \subseteq \mathcal{I}$ is the set of the channel identifiers.
- $id \in Y \to \mathcal{I}_{process}$ is a function that defines the identifiers of the processes.
- V_P is the set of variable identifiers of process P, where $P \in Y$. The variable sets of the processes are disjoint: $(\forall\, P, Q \in V, p \in V_P, q \in V_Q : P \neq Q : p \neq q)$.
- $V = \bigcup_{P \in Y} V_P$ is the set of all variables that occur in the model.
- $\gamma \in \mathbb{R}_{\geq 0}$ is the endtime of the simulation.
- $\perp$ is the distinguished value 'undefined'. The domain of each variable is extended with $\perp$.
- Σ_P is the set of *evaluations*[1] of the variables of process P, where $P \in Y$. Each evaluation $s \in \Sigma_P$ is an interpretation of V_P. It assigns to every variable $u \in V_P$ a value $s[u]$ in its extended domain. For an expression e, we write $s[e]$ to denote the value of e in evaluation s, that is, the value obtained by evaluating e after substituting each variable v in e by $s[v]$.
- $\Sigma = \{\{s_{P_0}, s_{P_1}, \ldots, s_{P_{n-1}}\} \mid (\forall\, i \in [0, \ldots n-1] : s_{P_i} \in \Sigma_{P_i}) \wedge (\forall\, i, j \in [0 \ldots n-1] : i \neq j : P_i \neq P_j) \wedge (\bigcup_{i=0}^{i=n-1} P_i = Y)\}$ is the set of evaluations of the whole variable set. Because the variable sets of processes are disjoint, each evaluation can be uniquely decomposed into evaluations of the variable sets of the processes.
- $\mathcal{S}_V$ is the set of statements that can be composed using the variables in V.
- $stat \in Y \to \mathcal{S}_V$ is a function that defines the actual statement of a process.

The set X of all possible process states consists of tuples of the form

$$(id(P), s, S)$$

where $P \in Y$, $id(P) \in \mathcal{I}_{process}$ is the process identifier, $s \in \Sigma_P$ is an evaluation of the process variables, and S is the statement of P that still needs to be executed.

The state of the scheduler is defined as

$$M = (\tau, A, W, S_\Delta, S_{sw})$$

where

- $\tau \in \mathbb{R}$ is the current time.
- $A \subseteq X$ is the set of active process states, i.e., it contains the state of each active process.

[1] In the literature, Σ is often referred to as the set of *states*. However, the word state is already used in this thesis with a more general meaning.

- $W = \{(t, ch, p) \mid t \in \mathbb{R} \wedge ch \in \mathcal{I}_{channel} \wedge p \in X\}$ is the set of *communication initiations* that contains information about processes blocked by communication. With each process state, a real and a channel name are associated, denoting the moment when the communication is initiated and the channel that is involved in the communication. Note, that a process may be suspended by more communications at a time, if it is executing a selective waiting statement that has several open communication alternatives. In this case, the same process may occur in more elements in W.
- $S_\Delta = \{(t,p) \mid t \in \mathbb{R} \wedge p \in X\}$ is the set of *sequential delta blockings* that contains information about the processes suspended by a delta statement. In this set, each process may occur in one element only. With each process state, the time is associated when it has to be activated.
- $S_{sw} = \{(t,p) \mid t \in \mathbb{R} \wedge p \in X\}$ is the set of *selective waiting delta blockings* that contains information about the processes suspended by a selective waiting statement, that has an open delta alternative. In a selective waiting statement, always the delta alternative is chosen with the smallest time-out. If there are more alternatives with the smallest time-out, then one is chosen nondeterministically. Therefore, in this set, each process may occur in one element only: the state belonging to (one of) the alternative(s) with the smallest time-out is stored. With each process state, the time is associated when it has to be activated.

5.2.1 Initial state

Let $s_0 \in \Sigma$ denote an initial evaluation such that

$$\forall u \in V : s_0[u] = \bot.$$

The initial state of the scheduler is

$$m_0 = (0, \{(id(P), s_0, stat(P)) \mid P \in Y\}, \emptyset, \emptyset, \emptyset).$$

5.2.2 Process execution

The scheduler works as follows. As long as there are elements in the active set, it picks one element arbitrarily and executes it. When the active set becomes empty, it activates some processes and may update the simulation time. Then, the cycle starts again.

Let m denote the current state of the scheduler

$$m = (\tau, a, w, sd, sw)$$

and let

$$r = (iden, s, S) \in a$$

denote the element to be processed. Depending on the first statement in S, the scheduler makes a transition to state m'. The transitions are summarized below.

1. Non-blocking statements: skip, assignment statement, selection statement, repetitive selection statement

   $$m' = (\tau, a \setminus \{r\} \cup \{r'\}, w, sd, sw)$$

   where $r' = (iden, s', S')$ is the process state after executing the first statement in S.

2. Delta statement

   $$m' = (\tau, a \setminus \{r\}, w, sd \cup \{rr\}, sw)$$

   where $rr = (t, r)$ and t is the time-out of the delta statement.

3. Communication statement

   - If the communication does not succeed

     $$m' = (\tau, a \setminus \{r\}, w \cup \{rrr\}, sd, sw)$$

     where $rrr = (\tau, ch, r)$ and ch is the name of the channel involved in the communication.
   - If the communication succeeds

     $$m' = (\tau, a \setminus \{r\} \cup \{r', q\}, w \setminus G, sd, sw \setminus H)$$

     where $r' = (iden, s', S')$ is the process state after the communication and q is the state of the communication partner after the communication. Furthermore, G is the set of the already initiated communications via the channel involved in the communication. The communication in the communication partner may be an event in a selective waiting statement. In this case, set H has a single element: it is (one of) the selective waiting delta blocking(s) with the smallest time-out that belongs to that selective waiting statement. Otherwise, set H is empty.

4. Selective waiting statement/repetitive selective waiting statement

   - If all alternatives are closed in a repetitive selective waiting statement

     $$m' = (\tau, a \setminus \{r\} \cup \{r'\}, w, sd, sw)$$

     where $r' = (iden, s, S')$ and in S' the selective waiting statement is removed.

- If there is an open delta alternative with negative time-outs

$$m' = (\tau, a \setminus \{r\} \cup \{r'\}, w, sd, sw)$$

where $r' = (iden, s, S')$ and S' starts with the statement of the delta alternative that has the smallest time-out.

- If a communication alternative is enabled and there are no open delta alternatives with negative time-out

$$m' = (\tau, a \setminus \{r\} \cup \{r', q\}, w \setminus G, sd, sw \setminus H)$$

where $r' = (iden, s', S')$ is the state after executing the longest waiting communication, and S' starts with the statement of the chosen communication alternative. Variable q is the state of the communication partner after executing the communication. Furthermore, G is the set of the already initiated communications via the channel involved in the communication. The communication in the communication partner may be an event in a selective waiting statement. In this case, set H has a single element: it is (one of) the selective waiting delta blocking(s) with the smallest time-out that belongs to that selective waiting statement. Otherwise, set H is empty.

- If there is no enabled communication alternative and all open delta alternatives have a positive time-out

$$m' = (\tau, a \setminus \{r\}, w \cup G^*, sd, sw \cup H)$$

where G^* is the set of communication initiations belonging to the open communication alternatives. If there are open delta alternatives, then set H contains a single element: the selective waiting delta blocking belonging to (one of) the open delta alternative(s) with the smallest time-out. Otherwise, set H is empty.

5.2.3 Process activation

When the active set becomes empty, a set of processes is activated in the following way. Let the state of the scheduler be $m = (\tau, a, w, sd, sw)$. Let u denote the minimum of the time points for which the processes blocked by a delta statement are waiting. Let v denote the minimum of the time points for which processes blocked at a selective waiting statement with open delta alternative are waiting. Formally, the definitions of u and v are:

$$u = \begin{cases} \min\{\ t \mid (t,p) \in sd\} & sd \neq \emptyset \\ \gamma & sd = \emptyset \end{cases} \tag{5.1}$$

$$v = \begin{cases} \min\{\ t \mid (t,p) \in sw\} & sw \neq \emptyset \\ \gamma & sw = \emptyset \end{cases} \tag{5.2}$$

Scheduling proceeds as follows.

- If $u \leq v$

$$m' = (u, E, w, sd \setminus F, sw)$$

where F is the set of sequential delta blockings with time-out u, and E is the set of process states obtained by taking the process state in each element of F and removing the delta statement from the beginning of the statements.

- If $u > v$

$$m' = (u, E, w \setminus G, sd, sw \setminus F)$$

where F is the set of selective waiting delta blockings with time out v, and E is the set of process states obtained by taking the process state in each element of F and taking the statement belonging to the delta alternative. G is the set of the already initiated communications, that belong to the same selective waiting statements as the selective waiting delta blockings of F. The simulation time is increased if $u > \tau$ and $v > \tau$. That is, when there are no processes that can execute a statement in a given time instant.

This scheduling scheme closely follows the operational semantics in [van de Mortel-Fronczak, 1995]. Note, that in the χ language concurrent execution means interleaved execution of processes. That is, parallel execution is simulated by nondeterministically choosing from possible active processes and executing them sequentially.

5.3 Continuous state calculation in a discrete context

The problem to be solved is how to determine in the course of a discrete phase when the initial state calculation actually has to take place. A possible choice could be the introduction of a language construct(s), that specifies that a discontinuity occurs (or has occurred) and the state needs to be re-calculated. When such a statement is executed, the state may immediately be re-calculated. In gPROMS [Barton, 1992] for example, the language constructs REINITIAL, RESET and REPLACE are used for this purpose. The problem can be further divided by asking the following questions:

- Whether or not to use a special language construct to indicate a discontinuity.
- After encountering a discontinuity whether or not to immediately update the system state. If the state is not updated immediately, then the question remains at which time it is updated.

In the χ language, no extra language construct is introduced for this purpose for the following reason. It would require the modeller to specify each time that a discontinuity occurs, and that the state has to be re-calculated. In fact it is very simple to predict when the state may need to be re-calculated: it is when the variables used in the equations are changed. Therefore, using a special language element for each state calculation is not necessary. Furthermore, such an element could easily be forgotten by modellers, leading to incorrect models. If such a language construct is not used, however, state consistency has to be checked after each assignment statement which is very inefficient. Therefore, an easily recognizable system state is needed when state re-calculation can automatically take place. In this way modellers do not have to specify state re-calculation at all obvious places, still they can keep track of how the state evolves.

Furthermore, re-calculating the state immediately after a discontinuity has two mayor problems. First, χ processes are executed concurrently. While one process may trigger an initial state calculation, other processes can refer to their connected variables that are subject to change as a result of the initial state calculation. If concurrent access of changing variables is not prevented, a scheduling dependency is introduced in the outcome of the simulation as described in Section 3.14. Prevention of access to changing variables may, for the first sight seem a simple task. When a differential variable is changed in one process all connected variables in other processes change as well. This situation may indeed be easy to recognize. However, it can be substantially more difficult to predict whether an algebraic variable is subject to change as a result of the re-initialization. In order to prevent use of changing variables, the modeller may have to analyze dependency among the variables.

Second, immediate state re-calculation may lead to undesirable intermediate states. Consider, for example, a piping system where two valves are used to select an alternative routing. The two valves must be both either in position *HIGH* or *LOW*, in order to select the upper or the lower pipe. This is controlled by a control unit as depicted in Figure 5.1. An intermediate state, in which one valve is in position *HIGH* and the other one is in *LOW* may not be desirable to be established, or even cannot be established.

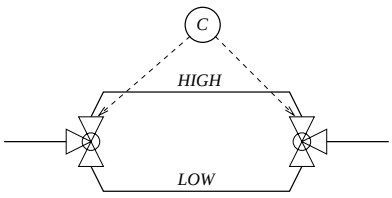

Figure 5.1: Pipe system with alternative routes.

Very often, one may want to specify only the state when both valves are in *HIGH* or in *LOW* position. When one valve is in position *LOW*, and the other one is in *HIGH* the substance will spurt from the pipe. The description of the outpouring is very complex, since it depends on the pressure in the pipe, on the kind of the substance,

the pipe cross section, etc. If modelling is not specifically aimed at analyzing faulty behaviour, the outpouring mode is not relevant in the model. It should perhaps only be detected. The describing equations are as follows

$$
\begin{array}{l}
[\ p_1 = HIGH \wedge p_2 = HIGH \longrightarrow Q_h = 1, Q_l = 0 \\
[\!]\ p_1 = LOW \wedge p_2 = LOW \quad \longrightarrow Q_h = 0, Q_l = 1 \\
]
\end{array}
$$

Variables p_1 and p_2 represent the position of the valves, and Q_h and Q_l are the flow in the upper and the lower pipes respectively. When the system switches from the upper pipe to the lower one, the code

$$p_1 := LOW\,;\, p_2 := LOW$$

is executed. After the first assignment, however, no consistent state can be established, since there are no equations associated with the state $\{\ p_1 = LOW, p_2 = HIGH\ \}$. In many hybrid languages it is possible to group several actions to specify that they all have to be executed before the initial state calculation takes place. This, as will be shown, is not necessary in the χ language. Still, grouping actions together would not solve the problem completely. Namely, those actions that should be grouped may take place in different processes. In this case yet another language mechanism is necessary to indicate that a new state should be established only when the processes are finished with their settings. (In χ models synchronization can be used for this purpose.) In Section 5.6 an example is given of this.

To prevent the above mentioned problems, in χ models the initial state is calculated when all processes are blocked, waiting for events to happen. It is an easily recognizable state: processes can become blocked only at events. Also, in this way the problem with concurrent access of changing variables is prevented. When the variables actually change, all processes are blocked. Finally, complex distributed (re-) initializations can be easily modelled. For example, some information needs to be shared among processes before they can (re-) initialize their variables. This can be modelled with the aid of discrete communication and synchronization. As long as communications do not block the processes, the state is not re-calculated. An example of this is given at the end of this chapter.

As explained in Section 3.10, state-event conditions are evaluated only in consistent states. When a state-event specification is encountered during a discrete phase, it blocks the execution of the process. The state-event condition is next evaluated when a consistent initial state is calculated. If found to be true, the process it belongs to becomes active, and a new discrete sub-phase resumes. An example for safe state-event handling follows in Section 5.6.

Between two initial state calculations, the state may become inconsistent. This, however, does not lead to modelling problems. Since all variables of the processes are local, the processes work with an independent local copy of the last consistent state.

In this way, the behaviour of the processes depends only on the local variables (and data that is sent explicitly via discrete channels). Ultimately, this leads to more clear and robust models, where reasoning about model behaviour is simplified.

5.3.1 Phase length

From the implementation point of view, the alternating behaviour of the discrete and the continuous phases can be summarized as follows. A discrete phase begins when one or more processes become active. When all processes are blocked, a consistent initial state is calculated. If subsequently a blocking state-event condition becomes true, the process it belongs to is enabled, and a new discrete sub-phase starts. This cycle continues until a consistent initial state is established in which all processes are blocked.

When a discrete phase is finished, a continuous phase starts. The length of the continuous-phase is limited by the smallest time-out. The continuous phase ends before that if *a)* the valid set of the DAEs is changed or *b)* a state-event condition becomes true in one of the processes. The first possibility occurs when continuous variables are used in the guards of the equations, and previously open guards become closed. As explained in Section 3.11, in the current implementation, continuous guards of equations are not monitored.

In practice, the length of the continuous phase that can be simulated is lower bounded by the minimum stepsize of the DAE solver, see Section 8.1.1. Therefore, if at the beginning of a new continuous phase the minimum of the pending time-outs is found to be closer to the current time point than the minimum stepsize, the next discrete phase is executed at the current time point (without changing the current time). This situation is not uncommon in big models, where events of different processes nearly coincide. Note that in this case, the simple solution of advancing the time is not feasible, because this may lead to numerical inconsistency.

5.4 The χ engine

This section treats the algorithm implemented by the hybrid χ engine. This is the computational model of the hybrid semantics. Currently, no other formal description of the hybrid semantics is available; that is subject to future research. The presented algorithm expresses, in a concrete form, the desired behaviour — especially the timing — of hybrid models.

The algorithm presented here is based on the discrete-event scheduling that has been introduced earlier in Section 5.2. As an extension, the hybrid scheduler handles state-events and processes that are blocked by a nabla statement. For this purpose, the following definitions are introduced.

- *nabla* $\in \mathcal{S}_V \rightarrow Bool$ is a boolean function indicating whether a statement is a nabla statement.
- *occurs* $\in \mathcal{S}_V \times \Sigma \rightarrow Bool$ is a boolean function indicating whether a statement is a nabla statement and its state event condition is true in an evaluation.

The state of the scheduler is extended, and is defined as

$$M = (\tau, A, W, S_\Delta, S_{sw}, S_E),$$

where $S_E \subseteq X$ is the set of *nabla blockings* that contains information about processes blocked by a nabla statement. Elements of S_E have the form

$$(id(P), s, S_0; S),$$

where $nabla(S_0)$. Note that a process may be blocked by more state-events at a time, if it is executing a selective waiting statement that has several open nabla alternatives. In this case, the same process may occur in more elements in S_E. Initially, this set is empty. The initial state of the scheduler is

$$m_0 = (0, \{(id(P), s_0, stat(P)) \mid P \in Y\}, \emptyset, \emptyset, \emptyset, \emptyset).$$

5.4.1 Hybrid process execution

Hybrid process execution is an extension of discrete process execution described in Section 5.2.2. In the hybrid version, the scheduler has to handle nabla statements, and nabla alternatives in selective waiting statements. Let m denote the current state of the scheduler;

$$m = (\tau, a, w, sd, sw, se)$$

and let

$$r = (iden, s, S) \in a$$

denote the element to be processed. The transitions made by the scheduler depend on the first statement of S.

A new transition is introduced when the first statement of S is a nabla statement:

$$m' = (\tau, a \setminus \{r\}, w, sd, sw, se \cup \{r\})$$

In this case, the process state is immediately moved into set *se*.

The transitions, described in Section 5.2.2 can change set *se*. Below, those transitions are listed, according to the first statement in S that do modify set *se*.

1. Communication statement

   - If the communication succeeds

     $$m' = (\tau, a \setminus \{r\} \cup \{r', q\}, w \setminus G, sd, sw \setminus H, se \setminus I)$$

     The communication in the communication partner may be an event in a selective waiting statement. In this case, set I is the set of nabla blockings that belong to that selective waiting statement. Otherwise, set I is the empty set.

2. Selective waiting statement/repetitive selective waiting statement

   - If a communication alternative is enabled and there are no open delta alternatives with negative time-out

     $$m' = (\tau, a \setminus \{r\} \cup \{r', q\}, w \setminus G, sd, sw \setminus H, se \setminus I)$$

     The communication in the communication partner may be an event in a selective waiting statement. In this case, set I is the set of nabla blockings that belong to that selective waiting statement. Otherwise, set I is the empty set.

   - If there is no enabled communication alternative and all open delta alternatives have a positive time-out

     $$m' = (\tau, a \setminus \{r\}, w \cup G^*, sd, sw \cup H, se \cup I)$$

     Let statement S have the form $R; Q$ and R have the form

     $$(*)[\ \begin{array}{lll} b_0 & ; e_0 & \longrightarrow R'_0 \\ \vdots & \vdots & \vdots \\ b_i & ; e_i & \longrightarrow R'_i \\ \vdots & \vdots & \vdots \\ b_{n-1} & ; e_{n-1} & \longrightarrow R'_{n-1} \end{array} \ ]$$

     where the guards are denoted by b_i, the event statements by e_i and the statements of the alternatives by R'_i, for $i \in [0 \ldots n-1]$. Set I denotes the nabla blockings belonging to the open nabla alternatives. If R is in a repetitive form, then

     $$I = \{(id, s, e_i; R'_i; R; Q) \mid i \in [0 \ldots n-1] \wedge s[b_i] \wedge nabla(e_i)\}$$

     else, if R is not in a repetitive form, then

     $$I = \{(id, s, e_i; R'_i; Q) \mid i \in [0 \ldots n-1] \wedge s[b_i] \wedge nabla(e_i)\}.$$

The process activation due to time-out that is presented in Section 5.2.3 is slightly modified as follows. Consider the case, when the selective waiting delta blockings are activated before the sequential delta blockings: $u > v$. The new state of the scheduler is

$$m' = (u, E, w \setminus G, sd, sw \setminus F, se \setminus I),$$

where sets E, G and F are the same as in Section 5.2.3. Set I contains those nabla blockings that belong to the same selective waiting statements as the elements in F.

5.4.2 Process activation when state-events occur

When a state-event condition of a blocking nabla statement becomes true in a consistent state, the process is activated. More processes can be activated at the same time. Formally, denoting the state of the scheduler by

$$m = (\tau, a, w, sd, sw, se)$$

the new state after activation is

$$m' = (\tau, a \cup G, w \setminus H, sd, sw \setminus I, se \setminus J).$$

Set J is the set of nabla blockings with a true state-event condition.

$$J = \{p \mid p = (id, s, S_0; S) \in se \wedge occurs(S_0, s)\}$$

Set G contains the states of the activated processes. If a process waits for several state-events and more than one occurs at the same time, then one is chosen non-deterministically.

$$\begin{aligned} G \quad = \quad & \{(id, s, S) | (\exists\, p = (id, s, S_0; S) \in se : occurs(S_0, s))\} \\ & \wedge (\forall\, (id_1, s_1, S_1), (id_2, s_2, S_2) \in G : id_1 \neq id_2). \end{aligned}$$

The nabla statements maybe events in a selective waiting statement. Set H contains the waiting initiations that belong to these selective waiting statement. Set I contains the selective waiting delta blocking(s) with the smallest time-out that belong to these selective waiting statements.

5.4.3 Hybrid scheduling

The algorithm of hybrid scheduling is described in the guarded command language used in computer science to specify computer algorithms. A detailed language description can be found, for example, in [Kaldewaij, 1990]. Many constructs of the χ language have been taken from the guarded command language, so the algorithm description is easy to read once familiar with χ. The syntactic differences, appearing

in the algorithm are as follows. Selection statements are enclosed between '**if** ... **fi**' pairs, and repetition of selection is enclosed between '**do** ... **od**' pairs. Otherwise, the semantics of these constructs is the same as in χ. Function and program names are written in lower case letters, in italic shape, and a dot is used for function application. Furthermore, constants and variables used by the algorithm are introduced with the keywords '**con**' and '**var**', respectively. For our purpose, a set type is introduced using the keywords '**set of**', and the special type 'procstate', denoting process states. The algorithm uses two constants.

- e is the simulation end time — it is given by the user as a simulation option
- s is the minimum stepsize of the DAE solver

The variables of the algorithm are

- τ is the current simulation time
- n is the time of the next predictable discrete-event
- m is the minimum of the time-outs of the blocking delta statement
- a is the set of active process states

The algorithm is described in Figure 5.2. The time aspects of hybrid scheduling is emphasized and the process scheduling is not detailed. For this reason, the sets of communication initiations, sequential delta blockings, selective waiting blockings and nabla blockings are omitted. The algorithm uses several operations. These are sub-programs that have access to the variables of the algorithm. We indicate only the time variables of these programs. They implement the following actions.

initialize executes the initialization block of each process instantiation, after which it fills set a with the initial states of the processes.

initialstatecalc calculates the initial state.

execute is the maximum alternating sequence of hybrid process execution (Section 5.4.1) and activation of processes blocked by a delta statement (Section 5.2.3) such that the simulation time remains the same ($u = \tau$ and $v = \tau$ in process activation). This is one discrete sub-phase. When *execute* is finished, all processes are blocked by an event statement and set a is empty.

activatenabla checks the conditions of blocking nabla statements, and if any one found to be true, it implements the process activation of state-events as described in Section 5.4.2.

mindelta returns the minimum of the time-outs as described in Section 5.2.3.

```
[[ con e, s : real
  var τ, n, m : real
            a : set of procstate
    τ, n, m := 0, 0, 0
  ; a := ∅
  ; initialize
  ; initialstatecalc ①
  ; do a ≠ ∅ ∧ τ ≤ e
       ⟶ do a ≠ ∅
              ⟶  execute ②
                 ; initialstatecalc ③
                 ; activatenabla ④
            od
       ; m := mindelta
       ; n := m min e ⑤
       ; if n < τ + s ⟶  activatedelta.n
                       ; if a ≠ ∅ ⟶ skip [] a = ∅ ⟶ τ := e fi
         [] n ⩾ τ + s ⟶ if continuous  ⟶  solve.n ⑥
                                       ; if τ = m ⟶ activatedelta.m
                                         [] τ ≠ m ⟶ skip
                                         fi
                       [] ¬continuous ⟶  τ := n
                                       ; activatedelta.n
                       fi
       fi
  od
]]
```

Figure 5.2: The scheduler algorithm.

activatedelta.t moves the process states of those processes that are blocked by a delta statement with a time-out equal to t to set a as described in Section 5.2.3.

continuous is true if the model contains continuous variables.

solve.t solves the DAEs until time t. It monitors the conditions of blocking nabla statements and updates τ. If any state-event condition evaluates to true, solving stops and it implements the process activation of state-events as described in Section 5.4.2.

It can be shown that the hybrid extension leaves the discrete-event behaviour intact, that is, purely discrete models are executed as described previously in Section 5.2. In the next section, the execution of the algorithm is demonstrated on two example models. For this purpose, the breakpoints ① ... ⑥ are introduced in the algorithm.

5.5 Example for multiple sub-phases

To illustrate how the scheduler works, consider the control system, depicted in Figure 5.3.

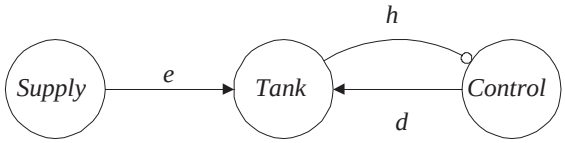

Figure 5.3: Tank process with supply and control processes.

```
proc Tank( h_ :: –o [m], d : ? int, e : ? real, k, A : real) =
| [ V :: [m^3], h :: [m], Q_o :: [m^3/s]
  , V_add : real, n : int
~| V ::= 10; n := 0
-| h_ –o h
=| V' = –n · Q_o
  , V  = A · h
  , Q_o = k · √h
  | ① *[ d ? n      ⟶ skip
       [] e ? V_add ⟶ V ::= V + V_add
       ]
] |

proc Supply( e : ! real) =
| [ *[ ① Δ10; e ! 30 ] ] |

proc Control( h_ :: –o [m], d : ! int, h_max, h_min : real) =
| [ h :: [m]
-| h_ –o h
  | *[ ① ∇h > h_max; d ! 1; ② ∇h < h_min ; d ! 0 ]
] |
```

Process *Tank* represents a tank filled with some substance. Variables V and h denote the volume and the height of the substance, respectively. The tank has an outlet with a valve. The outgoing flow is denoted by Q_o. The substance is entered by process *Supply* in discrete doses, represented by a discrete communication over channel e. Process *Control* controls the level of the substance in the tank. If it exceeds a maximum level h_{max} the outlet is opened, if it drops under a minimum level h_{min} the outlet is closed. The system is initialized with the following parameters:

```
syst S() =
| [ t : .Tank, s : .Supply, c : .Control
```

$| h :: -[\mathrm{m}],\ d : -\mathrm{int},\ e : -\mathrm{real}$
$| t(h, d, e, 1, 2.5) \parallel s(e) \parallel c(h, d, 15, 6)$
$] |$

The model has been executed with $endtime = 100$. The beginning of the trace is shown in Table 5.1. In the first column, the breakpoints in the scheduling algorithm are given. The values of the variables are printed in the order of $Tank.V$, $Tank.V'$, $Tank.h$, $Tank.Q_o$, $Control.h$, $Tank.n$.

breakpoints	variables	set *actives*	τ
①	$10, 0 \quad 4, 2, 4 \quad 0$	*Supply* ①, *Control* ①, *Tank* ①	0
⑤	$nextpoint = 10$	$\emptyset$	0
⑥	$10, 0 \quad 4, 2, 4 \quad 0$	*Supply* ①	10
②	$40, 0 \quad 4, 2, 4 \quad 0$	$\emptyset$	10
③	$40, 0 \quad 16, 4, 16 \quad 0$	$\emptyset$	10
④	$40, 0 \quad 16, 4, 16 \quad 0$	*Control* ①	10
②	$40, 0 \quad 16, 4, 16 \quad 1$	$\emptyset$	10
③	$40, -4 \quad 16, 4, 16 \quad 1$	$\emptyset$	10
④	$40, -4 \quad 16, 4, 16 \quad 1$	$\emptyset$	10
⑤	$nextpoint = 20$	$\emptyset$	10
⑥	$10, -4 \quad 4, 4, 4 \quad 1$	*Control* ②	17.75

Table 5.1: Example trace.

At time point 10 there is a discrete-event with two sub-phases. The supplier adds 30 m^3 substance (fourth row). The increase in the height of the tank is detected by the control process after the new initial state has been calculated (sixth row). In this example, a state-event condition that was originally false, evaluates to true in the new consistent state.

5.6 Example for safe state-event handling

There are multiple sub-phases in the following example as well. But, contrary to the previous example, in this system a state-event condition that was previously true becomes false in a new consistent state. If the state-event condition were evaluated in the old state, the state-event specification would not block the process which would result in an incorrect execution. This example illustrates why state-events should only be evaluated in consistent states.

Consider a part of a production line, where a product in a liquid state is produced and subsequently filled into cans. The example consists of two mixing tanks where the product is mixed, a buffer tank and a filling station. The mixing tanks and the buffer tank are supervised by a control unit. The layout of the χ model is depicted in Figure 5.4.

The following constants and types are used in the model.

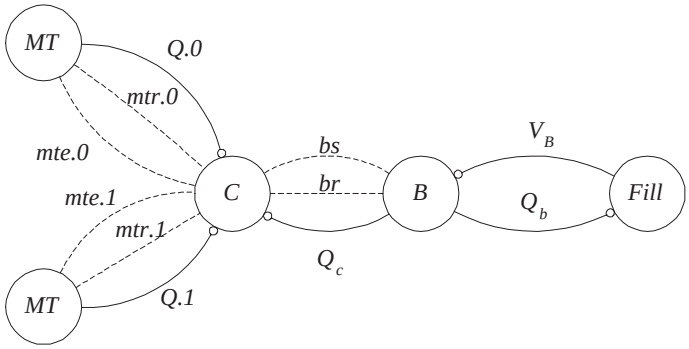

Figure 5.4: Mixing tanks with buffer and filling processes.

$\mathsf{const}\ \varepsilon : \mathrm{real} = 10^{-10}$

$\mathsf{type}\ \mathrm{vol} \quad = [\mathrm{m}^3]$
$,\quad \mathrm{flow} \quad = [\mathrm{m}^3/\mathrm{s}]$
$,\quad \mathrm{flowfactor} = \mathrm{nat}$

Filling and mixing in the mixing tanks are modelled in a discrete-event way using delta statements. The filling and mixing time could be stochastic, but in order to keep the model small, they are constant. Process MT represents the mixing tank. It synchronizes with the controller via channel *mtr* when it is finished with mixing, and via channel *mte* when it is empty, so that filling can start anew.

$\mathsf{proc}\ MT(\ Q\_ :: \multimap \mathrm{flow}$
$\qquad , mtr, mte : {}^{\sim}\,\mathrm{void}$
$\qquad , t_{fill}, t_{mix}, V_{full} : \mathrm{real}) =$
$|\ [\ V :: \mathrm{vol},\ Q :: \mathrm{flow}$
$\sim|\ V ::= 0$
$\dashv|\ Q\_ \multimap Q$
$=|\ V' = -Q$
$\quad |\ *[\ \Delta\, t_{fill};\ V ::= V_{full};\ \Delta\, t_{mix};\ mtr^{\sim};\ \nabla V \leq 0;\ mte^{\sim}\ ]$
$]\ |$

Only one mixing tank can empty its contents to the buffer tank at a time. This is controlled by the control process C. Process B represents the buffer tank. It has an incoming flow that comes from either one of the mixing tanks and an outgoing flow that goes to the filling station. Boolean variable *done* indicates whether the buffer tank is finished with a mixing tank. When process B notices that the incoming flow is switched off by the controller ($\nabla Q_i < \varepsilon$), so that the mixing tank is empty, it becomes immediately ready to empty a new tank. Practically, some more time could be needed before the tank is ready again, because of regular checking, cleaning, etc. This is not modelled here for sake of simplicity. The control process is notified that the buffer

tank is ready by a synchronization via channel br. The buffer tank may become full while emptying a mixing tank ($\nabla\, V \leq V_{max}$). In this case, it synchronizes with the controller via channel bs, in order to request the incoming flow to be temporarily switched off. When the contents in the buffer tank falls under a certain predetermined value $V_{max} - V_{hys}$, a new synchronization is needed to switch the incoming flow on again.

$$
\begin{array}{l}
\mathsf{proc}\ B(\ Q_{i\_}, Q_{o\_} :: \multimap \mathrm{flow},\ V_{\_} :: \multimap \mathrm{vol} \\
\qquad\quad , br, bs : {}^{\sim}\,\mathrm{void} \\
\qquad\quad , V_{max}, V_{hys} : \mathrm{real} \\
\qquad\quad) = \\
|\ [\ V :: \mathrm{vol},\ Q_i, Q_o :: \mathrm{flow} \\
\ \ , done : \mathrm{bool} \\
\rightsquigarrow |\ V ::= 0 \\
\dashv |\ Q_{i\_} \multimap Q_i \\
\ \ , Q_{o\_} \multimap Q_o \\
\ \ , V_{\_} \ \multimap V \\
= |\ V' = Q_i - Q_o \\
\ \ |\ *[\ br^{\sim}\,;\ done := \mathrm{false} \\
\qquad ;\ *[\ \neg done;\ \nabla\, V \leq V_{max} \longrightarrow bs^{\sim}\,;\ \nabla\, V < V_{max} - V_{hys}\,;\ bs^{\sim}\ \textcircled{1} \\
\qquad\quad [\!]\ \neg done;\ \nabla\, Q_i < \varepsilon \quad\ \longrightarrow done := \mathrm{true} \\
\qquad\quad] \\
\qquad] \\
]\ |
\end{array}
$$

Process C controls the flow between the mixing tanks and the buffer tank by setting the constants $l.0$, $l.1$ in the flow equation $Q_o = Q_{mt}.0 \cdot l.0 + Q_{mt}.1 \cdot l.1$. Boolean variable *done* indicates whether a mixing tank is finished. Process C chooses a mixing tank that is ready to empty. The number of the tank currently emptied is stored in variable j. While emptying, process C may receive a request from the buffer tank to temporarily switch the flow off ($bs^{\sim}$), or a notification from the mixing tank currently being emptied that it has finished ($mte.j^{\sim}$). In the former case, it switches the flow off ($l.j := 0$), and synchronizes with the buffer tank again ($bs^{\sim}$) when the level in the buffer tank drops so that it can switch the flow on again ($l.j := 1$).

$$
\begin{array}{l}
\mathsf{proc}\ C(\ Q_{mt\_} :: \multimap \mathrm{flow}^2,\ Q_{o\_} :: \multimap flow \\
\qquad\quad , mtr, mte : {}^{\sim}\,\mathrm{void}^2,\ br, bs : {}^{\sim}\,void \\
\qquad\quad) = \\
|\ [\ Q_{mt} :: \mathrm{flow}^2,\ Q_o :: \mathrm{flow},\ V :: \mathrm{vol} \\
\ \ , done : \mathrm{bool},\ l : (\mathrm{real})^2,\ j, k : \mathrm{nat} \\
\rightsquigarrow |\ V ::= 0\,;\ l :=< 0, 0 > \\
\dashv |\ k \leftarrow [0..2) : Q_{mt\_}.k \multimap Q_{mt}.k \\
\ \ , \qquad\qquad\qquad Q_{o\_} \multimap Q_o \\
= |\ Q_o = Q_{mt}.0 \cdot l.0 + Q_{mt}.1 \cdot l.1
\end{array}
$$

```
  | done := true
  ; *[ done  ; br~      ⟶ [ k ← [0..2) : mtr.k~ → j := k ]
                           ; l.j := 1; done := false
     [] ¬done; bs~      ⟶ l.j := 0; bs~; l.j := 1
     [] ¬done; mte.j~   ⟶ done := true; l.j := 0
     ]
] |
```

Finally, the filling station, modelled by process $Fill$, fills containers of volume V_{max}. The filling can start if the buffer contents is at least 0.1m^3 ($\nabla\, V_B > 0.1$). When a container is filled ($\nabla\, V_{CON} \geq V_{max}$), a new container is placed below the filling nozzle. This takes t_{crp} time units ($\Delta\, t_{crp}$). The number of containers filled is stored in variable n. If the buffer is empty before the container is full, the flow is temporarily switched off ($\nabla\, V_B \leq \varepsilon \longrightarrow l := 0$).

```
proc Fill(Q_- : -o flow, V_B- : -o vol
          , t_crp, V_max, Q_set : real) =
| [ V_B, V_CON :: vol
  , l : flowfactor, n : nat
~| V_CON ::= 0; l := 0; n := 0
-| Q_-  -o Q
  , V_B- -o V_B
=| V'_CON = Q
  ,      Q = l Q_set
  | *[ ∇ V_B > 0.1; l := 1
     ; [ ∇ V_CON ≥ V_max ⟶ l := 0; Δ t_crp; V_CON ::= 0; n := n + 1
       [] ∇ V_B ≤ ε      ⟶ l := 0
       ]
     ]
] |
```

The complete system is specified as

```
syst S() =
| [ mt : .MT, b : .B, c : .C, fill : .Fill
  , Q :: −flow², Q_c, Q_b :: −flow, V_b :: −vol
  , mte, mtr : −void², bs, br : −void
  | k ⟵ [0..3) : mt.k(Q.k, mtr.k, mte.k, 10, 60, 20)
  || b(Q_c, Q_B, V, br, bs, 30, 5)
  || c(Q, Q_C, mtr, mte, br, bs)
  || fill(Q_B, V_B, 2, 5, 2)
] |
```

There is a deadlock in this model when the mixing tank becomes empty and the buffer tank becomes full at the same time. In that situation, in process B both $\nabla\, V \leq V_{max}$

and $\nabla Q_i < \varepsilon$ become true simultaneously. Suppose, that the first alternative is chosen. Now both process B and process MT will try to communicate with the control process; process MT via channel $mte.j$, process B via channel bs. If the controller chooses to respond to process MT, $done$ becomes $true$, and the controller proceeds to wait for a ready signal from process B ($br\sim$). But process B is blocked, waiting for the controller to respond to the temporary switch off request at channel bs.

In order to prevent this situation, we have to take into account that when the buffer tank is full the controller may not respond for the previous reason. In that situation, however, the flow does not have to be switched off, since the mixing tank is already empty. In this case, the buffer tank can just wait until the volume drops under $V_{max} - V_{hys}$, and then proceed to signal the controller that it is ready to empty another buffer ($br\sim$). This change means that the line marked by ① in process B is replaced by

$$[\!] \neg done\,;\, \nabla V \leq V_{max} \longrightarrow \left[\begin{array}{ll} bs\sim & \longrightarrow \nabla V < V_{max} - V_{hys}\,;\, bs\sim \\ [\!] \nabla V < V_{max} - V_{hys} & \longrightarrow \text{skip} \end{array}\right]$$

5.6.1 Required behaviour

Consider the moment when one mixing tank becomes empty, and the system switches to the other mixing tank. Variable *done* of process B is false, and the process waits either for $V \leq V_{max}$ or for $Q_i < \varepsilon$. The mixing tank currently emptying is waiting at $\nabla V \leq 0$. Variable *done* of process C is false, and process C waits either for $bs\sim$ or for $mte.j\sim$. When the mixing tank becomes empty, the controller synchronizes with it ($mte.j\sim$). The controller proceeds, synchronizes with the buffer tank ($br\sim$), chooses the next mixing tank, and switches the flow on ($l.j := 1$). The buffer tank notices that the flow has been switched off ($\nabla Q_i < \varepsilon$). Consecutively, variable *done* is set to true and the execution proceeds with the synchronization with the controller ($br\sim$), after which *done* is set to *false*. Since process B has the old local value of variable Q_i, which is less than ε, the expression $Q_i < \varepsilon$ is true at this moment. However, choosing this alternative ($\nabla Q_i < \varepsilon$) for execution would be a fatal error, meaning that the buffer tank considers the mixing tank already empty. In this model it is crucial that the nabla statement $\nabla Q_i < \varepsilon$ blocks the execution of the process, and is evaluated only in a new consistent state. Then, in the new state, the effect of the fact that the controller has switched another buffer on is considered correctly: process B remains blocked.

5.7 Discussion

5.7.1 Integration in other languages

It is interesting to note that the χ perception of hybrid systems is just the complement of the hybrid automata perception in the way the discrete and the continuous states evolve relative to each other. On the one hand, when processes of a χ model execute their discrete body in a discrete sub-phase, they mainly shift from discrete states to discrete states. A discrete action may of course imply a change in the continuous state as well if, for example, a continuous variable is changed. Furthermore, the continuous state during the discrete state shifts is not necessarily consistent. On the other hand, in case of hybrid automata, the system transits between consistent continuous states. While transiting, some discrete actions can take place, so that the discrete state evolves.

5.7.2 The underlying time model

The underlying time model of the χ language is based on the super dense time model as described in [Maler et al., 1992]. The time domain $\mathbf{T}$ is the nonnegative real numbers $\mathbb{R}_{\geq 0}$. A *progressive time sequence* is an infinite sequence of time stamps

$$\Theta : t_0, t_1, \ldots$$

where $t_i \in \mathbf{T}$, for each $i = 0, 1, \ldots$, and

- $t_0 = 0$
- Time does not decrease, i.e., for every $i \geq 0, t_i \leq t_{i+1}$.
- Time eventually increases beyond any bound. That is, for every time element $t \in \mathbf{T}, t_i > t$ for some $i \geq 0$. This is equivalent to the non-zeno criterion.

Let $\Theta : t_0, t_1, ...$ be a progressive time sequence. The time-structure induced by Θ is an infinite set of pairs

$$\mathsf{T}_\Theta : \{\langle i, t\rangle \mid i = 0, 1, \ldots \quad t = t_i \vee t_i < t < t_{i+1}\}$$

The elements of T_Θ are called $\Theta - \textit{moments}$, or simply *moments*. The moments in the form of $\langle i, t_i\rangle$ are the *discrete moments* of T_Θ, whereas those in the form of $\langle i, t\rangle$, where $t_i < t < t_{i+1}$ are the *continuous moments* . In order to incorporate our sub-phase structure, a *phase sequence* is introduced. A phase sequence is an infinite sequence of the natural numbers

$$\Lambda : \lambda_0, \lambda_1, \ldots$$

where $\lambda_i \in \mathbb{N}$, for each $i = 0, 1, \ldots$, and

- $\lambda_0 = 0$
- $\lambda_{i+1} = \lambda_i \vee \lambda_{i+1} = \lambda_i + 1$

The idea is that the discrete sub-phases are indexed, and λ_i denotes the number of the sub-phase the discrete moment $\langle i, t_i \rangle$ belongs to. A *phase sequence of a progressive time sequence* Λ_Θ is such that

- if $\lambda_i = \lambda_{i+1}$ then $t_i = t_{i+1}$
- if $t_{i+1} > t_i$ then $\lambda_{i+1} > \lambda_i$

The requirement is that only discrete moments of the same time stamp can belong to the same sub-phase.

The phase sequence of a progressive time sequence generates a sequence of *pseudo moments*.

$$\mathsf{P}_{\Lambda_\Theta} : \{\langle i, t_i \rangle \mid t_i = t_{i+1} \wedge \lambda_i \neq \lambda_{i+1} \vee t_i < t_{i+1}\}$$

These are the moments when the state of the system is updated. A pseudo moment $\langle i, t_i \rangle$ means that the system state is re-calculated after the discrete moment i at time stamp t_i. Although at these moments the system may change, there is no corresponding discrete action. For example, the time structure induced by the time sequence $\Theta : 0, 2, 2, 3.5, 3.5, 3.5, 4, \ldots$ and the phase sequence $\Lambda : 0, 1, 1, 2, 2, 3, 4, \ldots$ is shown in Figure 5.5.

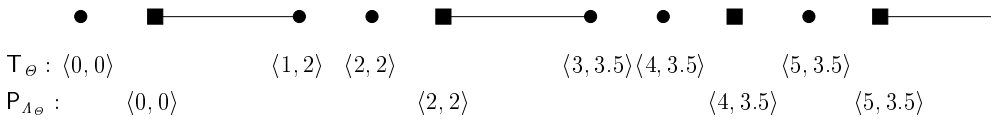

Figure 5.5: Time structure.

The discrete moments are depicted with a circle (•) and the pseudo moments with a rectangle (■). There are uncountably many continuous moments between two discrete moments with different time stamps; they are represented with a line.

A discrete phase is a maximal subsequence $\langle i, t \rangle, \langle i+1, t_{i+1} \rangle, \ldots, \langle j, t_j \rangle$, for $i \leq j$, where $t_i = t_{i+1} = \ldots = t_j$. A continuous phase is a nonempty open-interval $O_i : \{\langle i, t \rangle \mid t_i < t < t_{i+1}\}$, for $t_i < t_{i+1}$. A discrete sub-phase is a maximal subsequence $\langle i, t \rangle, \langle i+1, t_{i+1} \rangle, \ldots, \langle j, t_j \rangle$, for $i \leq j$, where $\lambda_i = \lambda_{i+1} = \ldots = \lambda_j$.

Chapter 6

The architecture of the χ simulator

In the coming three chapters, the implementation of the hybrid language elements are presented. In order to do this, first the architecture of the χ simulator is outlined in this chapter. The two main system components, the χ compiler and the χ engine are introduced. Their functionality is explained and their central objects are discussed in more detail. This architecture served as basis for the implementation of the continuous elements. The hybrid extension follows the design of the already existing components and makes use of information available. Wherever possible, continuous language elements are implemented similarly to their existing discrete counterparts.

6.1 Notation

We use the *object model* of *Object Modeling Technique* (*OMT*), as described by Rumbaugh et al. [Rumbaugh et al., 1991] to present the design of the classes. The OMT object model describes the static structure of objects in a system and their relationships. The object model is represented graphically by object diagrams. In this thesis we use *class* and *instance diagrams* . A class diagram is a graph whose nodes are classes and whose arcs are *relationships* between classes. In a class diagram, a class is depicted by a box containing the name of the class, the *members* or *attributes* of the class, and its *methods*. An instance diagram shows how a particular set of objects relate to each other. It is used to depict an example of a class diagram. Classes can be associated, which is depicted by lines connecting the classes. The name of the association can be written on the line. A class can be a *generalization* of another class (*'isa' relation*). This is depicted by a line with a triangle on it, with its base at the subclass (also called child class) side and its apex at the super class (also called parent class) side. *Aggregation* is the *'has-a' relation*, denoting that objects of certain classes are assembled into a bigger unit. This is depicted by lines ending with a rhombus at the assembly class. A relation has a *multiplicity*. This specifies how many objects of a class are related to a single object of the associated class. A single line denotes exactly one object. Multiplicity can be denoted by dots at the end of the association line. A solid dot (•) means that zero or more objects, open dot (○) means that zero

or one object is associated. Finally, the exact number of objects participating in a relation can be given by a number written on the association line.

6.2 System layout

The current χ simulator, version 0.5, is a hybrid simulator capable of executing discrete-event, continuous-time, and hybrid χ models. So far, this is the second version that integrates continuous language elements; other, previous versions were capable of executing discrete-event models only. The continuous language elements have been added, in both hybrid versions, later.

The discrete-event simulator that served as basis for the current hybrid simulator was that of described in [Naumoski and Alberts, 1998]. Its structure is shown in Figure 6.1.

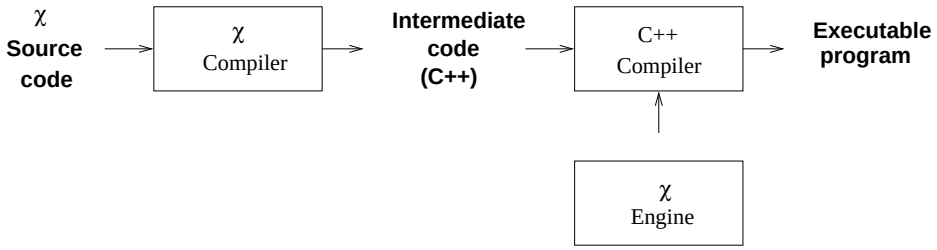

Figure 6.1: The discrete-event χ simulator.

The simulator consists of two pieces of software; the χ *compiler* and the χ *engine* (also called the χ kernel). The simulator works in two steps. First, the χ compiler reads the source code and checks it syntactically and semantically. Then, if the source code is correct, an intermediate code written in C++ is generated. This code is linked to the χ engine by the C++ compiler, which results in an executable program. Simulation means the execution of this program. The χ engine is a runtime environment that implements the concurrent behaviour of χ processes. The heart of it is a scheduler that has been described earlier in Section 5.2.

Hybrid extension to this architecture requires first the identification of the information that is required by the engine to be able to execute the model. This information needs to be extracted from the source code by the compiler. This leads to the interface specification between the compiler and the engine with regard to the continuous language elements. The interface depends on how these elements are implemented in the engine. The implementation can be divided into the following main subjects:

- continuous variable representation
- equation representation
- implementation of the continuous state calculation

- extension of the process representation and scheduling implementation with the execution of nabla statements
- implementation of state-event monitoring

We go through the first two items in detail in Chapter 7, and the last three items in Chapter 8. The hybrid extension to this architecture is depicted in Figure 6.2.

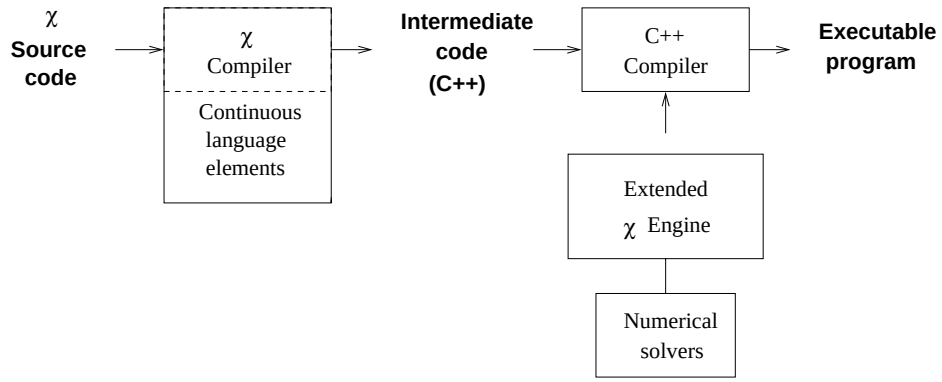

Figure 6.2: The extended χ simulator.

The χ compiler has been extended to compile continuous χ language elements, to extract the required information about the continuous behaviour of the model and to produce the output in the format required by the engine.

The χ engine has been extended to handle interactions between continuous and discrete parts of processes, to monitor discontinuities in the model, and to calculate the continuous state of the processes. Two numerical solvers have been integrated into the χ engine. The nonlinear equation solver NLEQ [Novak and Weiman, 1991] carries out initial state calculations, as described in Chapter 4. During a continuous phase, the DAEs are solved by DDASRT [Petzold, 1983] which is a DAE solver with root finding functionality. The latter one being used for monitoring state-event specifications. The χ engine can integrate other solvers in a similar way.

Both the compiler and the engine are written in C++ , using object-oriented software technology. This proved to be beneficial, allowing reuse of code, and fast and smooth extension. The development platform is UNIX (Linux), while experiments take place in both UNIX and MS-Windows environment.

6.3 The χ compiler

Compilers in general, are complex pieces of software that translate programs written in a source language into machine language. The χ compiler differs from ordinary compilers in a way, that it translates χ source code into a high level language: C++ .

Nevertheless, it follows the phase structure of ordinary compilers, to be described in the following. In order to keep full control on the internal data structures in the hands of developers, and to exploit advantages of object-oriented programming, the χ compiler is written in C++ . It is implemented as a number of separate phases that work together. These are

- lexical analysis
- syntactic analysis
- intermediate code generation
- semantic analysis
- code generation

The job of the lexical analyzer or *scanner* is to break the source code into meaningful units called *tokens*. Then, the syntactic analyzer or *parser* determines the structure of the program based on the grammar description. The intermediate code generation phase generates an internal representation of the program, that reflects the information uncovered by the parser. Note that according to the literature we use here the term 'intermediate code' meaning the internal code on which the compiler works in the next two phases. In the simulator we use 'intermediate code' denoting the C++ code that is the output of the compiler. On the intermediate code, semantic checking is carried out by the semantic analyzer, as well as some preparatory work, targeted for the next step. Finally, the output code is generated based on the results of the previous steps. The compiler makes use of the freeware counterparts of Lex and Yacc: flex and bison, for generating a scanner and a parser.

The first three phases - the *front-end* of the compiler - work closely together. The lexical analyzer goes into operation when the parser requests a token. Subsequently, if the parser requests it, a piece of intermediate code is generated. The last two steps operate on the intermediate code, which is the internal representation of the χ model. The structure of the original code is reflected in a class hierarchy, which allows these steps to be executed hierarchically. An advantage of using object-oriented technique is that the compiler is easily extendable. We have seen this advantage especially when a language element had to be added, where similar elements have already been implemented. For example, a new type or a new event statement.

In this thesis, the first two phases, scanning and parsing are not covered in detail, since these follow standard techniques that are well documented in the literature. For a comprehensive work on compiler construction, readers are referred to the 'Dragon Book' of Aho et al., [Aho et al., 1986], and to the introductory work of Parsons [Parsons, 1992]. The ultimate book on object-oriented compiler construction is [Holmes, 1995].

6.3.1 The CChiCompiler class

The central entity of the χ compiler is an instance of the class CChiCompiler, depicted in Figure 6.3. A compilation session starts with the construction of this object and

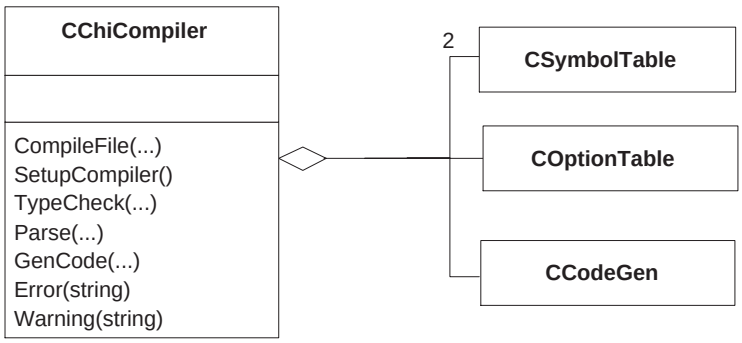

Figure 6.3: The CChiCompiler class.

continues with the call of the compilation method CompileFile. This method governs compilation through the phases described above. The functionalities provided by the class CChiCompiler are

- parsing and building the intermediate code (Parse)
- semantic analysis (TypeCheck)
- code generation (GenCode)

Implementing a central entity, the class CChiCompiler provides the following additional functionality.

- setting up the compilation process (SetupCompiler)
  - processing configuration options
  - initializing the compilation environment with built-in units, types, functions, etc.
  - opening files
  - reading libraries
- handling files
- reporting errors and warnings (Error and Warning)

The first three phases of the compilation are implemented in the Parse method that runs the parser and builds the intermediate code. The intermediate code is constructed in data members of the class CChiCompiler and in temporary data

structures. Semantic analysis is implemented by the TypeCheck method. It cascades down in the intermediate code, checking correctness at different levels. To carry out semantic analysis, several tables that are data members of the class CChiCompiler are available. These are the table of types (CSymbolTable), a table of units (CUnitTable) and a table of global symbols (CSymbolTable).

6.3.2 Intermediate code of the χ compiler

For each parse tree node (except for the keywords and special symbols), a C++ object is generated. To maintain consistency, the classes are named after the nonterminal they represent. For example, an object of type CStatement is generated, whenever the production rule of a statement is parsed. The semantic attributes used in conventional syntax-trees are replaced by data members of the objects, specific to the particular language construct. In this way, the various attribute data structures are easily accessible. Semantic correctness is specified for each object and is implemented in the TypeCheck method. This method concentrates only on object specific issues and information that needs to be passed to neighboring nodes.

Semantic analysis is carried out by more traverses on the intermediate code. The properties checked can be grouped as follows.

- type correctness
  The correct definition and use of types are checked in type definitions, variable and parameter declarations, expressions, links, equations and statements. The typing environment and the techniques used for checking correct use of types are described in [Naumoski and Alberts, 1998].
- scope, name-space correctness
  Correct, unambiguous use of identifiers is checked. Overloaded function names are resolved, and polymorphic functions are instantiated with the right type.
- layout correctness
  In systems, the correct use of channels is checked (e.g., they connect exactly two processes).

With the development of the hybrid χ compiler, the structure of the already present intermediate code has not been changed. The philosophy of the compiler has been followed in the design of the new elements, making use of already present information as much as possible. Techniques and data structures used to carry out type, name-space and layout checking have been extended and used to verify correctness of continuous language elements.

6.4 The χ engine

The χ engine is a runtime environment that implements the concurrent behaviour of χ processes. Its central entity is the scheduler, represented by an object of class CScheduler. It implements the scheduling algorithm that has been described previously: the discrete-event scheduling in Section 5.2, the hybrid scheduling in Section 5.4.

In the code generation phase, the compiler maps the χ model onto a C++ code. This code closely follows the original χ specification. Each high level language construct, that is, each type, function, process and system definition is mapped onto a corresponding C++ construct. The general functionality of processes and systems are defined in the engine by the abstract[1] class CProcess. In the generated code, for each process and system specification of the χ model, a specific process or system class is derived from the class CProcess. Just like in the original χ specification, instances of the derived process classes can be executed, and instances of the derived system classes specify the layout of the contained processes and subsystems. Discrete types and functions are mapped onto C++ types and C++ functions. Discrete channels used for synchronization are implemented as instances of the class CChannel. Classes that represent channels communicating data are generated from the template CTChannel. This template takes the data type of the channel as parameter, thus channel classes with different data types can be generated.

The generated classes are instantiated in the same way as in the χ code. In this way, the same structure is built as in the χ model. Again, a set of processes are instantiated, now these are C++ objects, that are 'connected' by channel objects. Connections are associations between process and channel objects. They are implemented by letting process objects store pointers of channel objects.

6.4.1 The CScheduler class

The central entity of the χ engine is an object of the class CScheduler, depicted in Figure 6.4 that implements the scheduling algorithm and acts as a central coordinator.

Simulation starts with the creation of this object and the instances of the mapped χ model, followed by a call of the Run method of the class CScheduler.

To implement the scheduling algorithm, the following data structures had to be implemented in the χ engine.

- the simulation time
- the set of active process states
- the set of communication initiations

[1] An abstract class is a class that cannot have direct instances, and is used to define features common to its descendants [Rumbaugh et al., 1991].

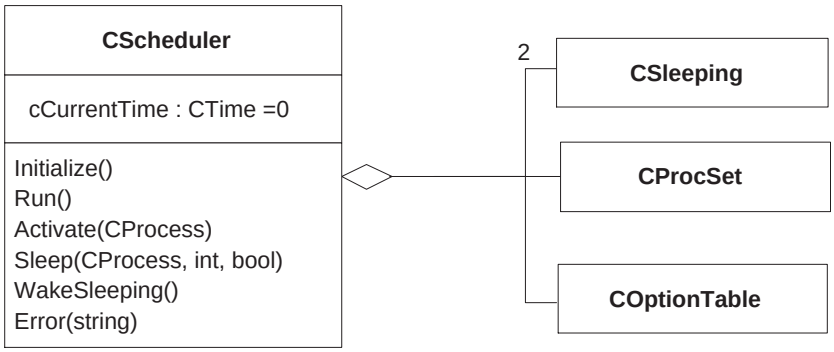

Figure 6.4: The CScheduler class.

- the set of sequential delta blockings
- the set of selective waiting delta blockings

In the hybrid engine, a new data structure representing the set of nabla blockings is added.

The simulation time is stored in the cCurrentTime attribute of CScheduler. It has type real (CTime defines a type real) and is initialized to zero. The state of the process is encoded in the process object itself. The set of active process states is updated frequently by state transitions, therefore, it needs to be implemented efficiently. For this purpose, it is implemented by a balanced leaf-tree (CProcSet), see [Naumoski and Alberts, 1998]. Channel objects store the time point when the communication is initiated, and the process that has initiated the communication. In this way, the information represented by the set of communication initiations is available via processes, so this set is not implemented as a separate data structure. Finally, the sets of the sequential and the selective waiting delta blockings are represented by data members of type CSleeping. They are implemented using balanced leaf-trees and leaf-dictionaries. For efficiency, implementation of the above data structures store only references to process objects, rather than complete process states.

Apart from implementing the scheduling algorithm, the CScheduler class provides coordination functionalities. These are

- setting up simulation (Initialize)
  - reading simulation options
  - handling files
- handling run-time errors (Error)

With the hybrid extension, a trace file can be generated that registers the simulation, recording the events and the continuous state of the model. Also, the value of the continuous variables can be logged into file at regular time intervals. These functionalities are described in Appendix B.

6.4.2 Process execution

The class CProcess depicted in Figure 6.5, implements the common functionality of processes.

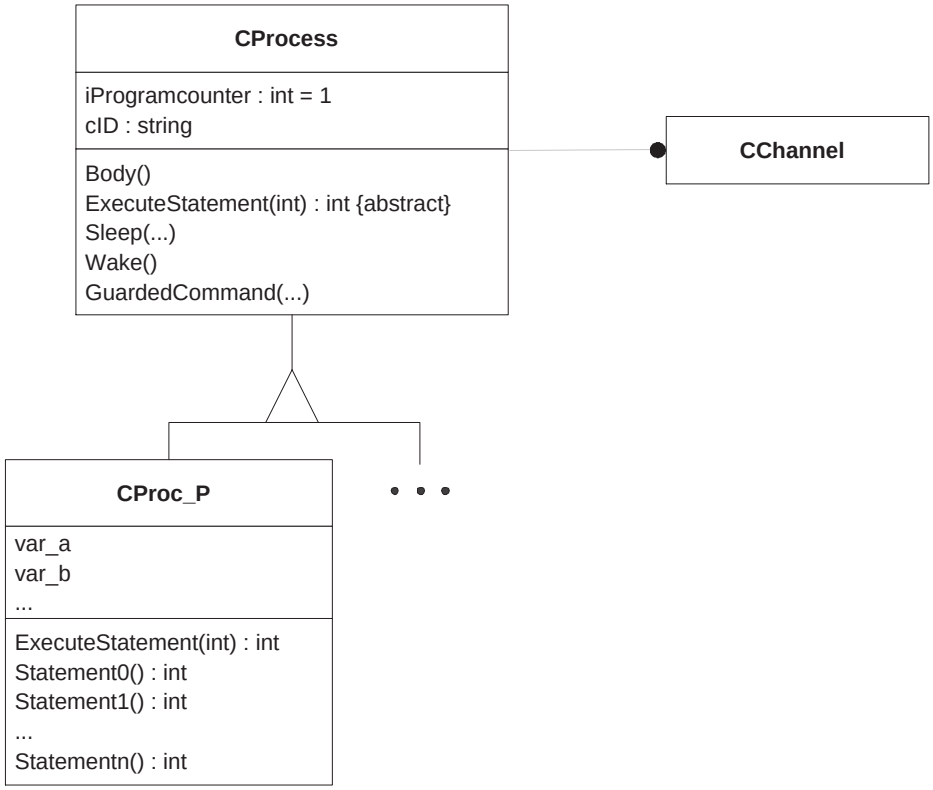

Figure 6.5: The CProcess class.

For each χ process specification, a new process class is derived that specifies the actual variables, parameters and statements of the process. In Figure 6.5, a new class CProcess_P is derived, that represents the χ process specification P. Instances of the derived classes can be executed similarly to χ process instances. The execution is implemented by mapping the discrete-event body of χ process specifications into *statement methods* that can be executed.

Statements of a χ process specification are numbered and translated into an array of statement methods of the mapped process class. The index of the statement method to be executed next is stored in the data member iProgramcounter. Having a method for each statement is inefficient. Therefore, sequences of non-interruptable statements (i.e., non-event statements) are grouped together into one method. During scheduling, the scheduler examines the set of actives, and picks a process object for execution. It

is implemented by a call of the method Body of this object. The Body method in turn calls the ExecuteStatement method. This method executes the statement methods. It starts with the execution of the one indexed with the current value of the program counter. Statement methods return the index of the next statement method in such a way, that ExecuteStatement navigates through statement methods according to the original χ process specification.

In discrete-event systems, process execution ceases if a communication is initiated and the communication partner did not reach a corresponding communication statement yet, or, if a delta statement is reached that has a positive time-out. Execution of nabla statements are implemented similarly to execution of delta statements.

Chapter 7

Variable and equation representation

The behaviour of the continuous variables has been defined in Chapters 3 through 5. Efficient simulation of this behaviour, however, requires further analysis of the different variable categories. The number of allowed variable categories has been reduced, to avoid erroneous categories and to simplify the language.

In this chapter, we discuss the representation of the continuous variables and equations in the χ simulator. For this purpose, the variable categories are further analyzed, and their behaviour is formulated per category. The correct use of the continuous variables is checked by the χ compiler. The internal variable structure of the χ engine is introduced, and operations on continuous variables (assignment, assignment of a guess) are mapped onto this structure. The presented structure is suited to calculate variables numerically by solvers, or analytically by substitution. It eliminates connection equations, still it preserves the connection semantics.

7.1 Variable categories

In the χ compiler, correct use of the continuous variables is checked during semantic analysis. Checks carried out on discrete variables are applied to continuous variables as well. For example, a variable has to be defined correctly, that is, its name has to be unique within its scope and may not clash with function names, type names, etc. Furthermore, continuous variables must have a correct continuous type.

The way continuous variables can be assigned depend on their category. The possible variable categories are presented in Table 7.1. We distinguish differential and algebraic variables (abbreviated to 'dif' and 'alg' in the first column of Table 7.1). If the time derivative of a variable occurs in the non-substitute equations (second column, 'der. in eq.'), then it is a differential variable. If it does not, then it is an algebraic variable. Note, that in the latter case, it is possible that the variable does not occur in the non-substitute equations at all. Namely, the variable itself can be calculated by

substitution, and its value may not be needed to calculate other variables. In the rest of this chapter, we refer to the non-substitute equations shortly as equations.

Another kind of division is applied to substituted variables. They are divided into base, prime or base-prime substituted variables, according to the form of the variable that is substituted . (See substituted variables in Section 3.11). In Table 7.1, the category names are abbreviated to 'b-sub', 'p-sub' and 'bp-sub' in the first column. The last two columns, 'var. subst.' and 'der. subst.' indicate whether the variable itself or its time derivative is calculated by substitution. Table 7.1 presents all possible combinations of the differential/algebraic and the b-sub/p-sub/bp-sub divisions. Don't care (x) in an entry means, that it may be '+' or '-'.

category	var. in eq.	der. in eq.	var. subst.	der. subst.
alg, (non-sub)	+	-	-	-
dif, (non-sub)	x	+	-	-
alg, b-sub	x	-	+	-
alg, p-sub	x	-	-	+
alg, bp-sub	x	-	+	+
dif, b-sub	x	+	+	-
dif, p-sub	x	+	-	+
dif, bp-sub	x	+	+	+

Table 7.1:Continuous variable categories (+ = yes, x = don't care, - = no).

The non-substituted variables are listed in the first two rows. They are either algebraic or differential. Their values are calculated by numerical solvers.

In the rest of the table, the different kinds of substituted variables are listed. Not all of these categories are allowed in χ models, however. Some categories are erroneous. Furthermore, there are some other ones that we consider as having no practical use. Allowing to use variables of these latter categories would unnecessarily complicate matters. Therefore, currently only three substitute variable categories are allowed in χ models.

7.1.1 Correct variable categories

The correct variable categories, currently allowed in χ models are listed in Table 7.2. This table serves as the definition of the categories used in the rest of this chapter. These category names are therefore printed in capitals, to make them different from the names listed in Table 7.1.

The correct substitute variable categories in Table 7.2 are ALG-B-SUB, DIF-P-SUB, and DIF-BP-SUB. The form that is substituted is calculated by evaluating the right-hand-side expression of the substitute equation. In the case of category ALG-B-SUB, the variable itself is calculated by substitution. The time derivative is not defined,

category	var. in eq.	der. in eq.	var. subst.	der. subst.
ALG	+	-	-	-
DIF	x	+	-	-
ALG-B-SUB	x	-	+	-
DIF-P-SUB	+	+	-	+
DIF-BP-SUB	x	+	+	+

Table 7.2:Definition of correct variable categories (+ = yes, x = don't care, - = no).

and cannot be used in the discrete-event statements. In the case of category DIF-BP-SUB, the variable itself and its time derivative are both calculated by substitution. A variable of the special category DIF-P-SUB is calculated as follows. Wherever its time derivative is used, in the equations or in the discrete-event part of the process, it is calculated by substitution. The value of the variable itself, however, is calculated by numerical solvers, from the equations. These kinds of variables are used for index reduction. The use of substituted variables is further discussed in Chapter 9.

7.1.2 Categories with additional restrictions

The category *differential prime substituted* in Table 7.1 can only be used in χ models if the variable itself occurs in the equations (category DIF-P-SUB in Table 7.2). Otherwise, the variable is not defined. It cannot be calculated by solvers, neither by substitution.

7.1.3 Erroneous categories

The category *differential base substituted* is left out from Table 7.1 because it is erroneous. In general, if such a variable is defined, the equations cannot be solved. To see why this is true, consider the original set of equations without using any substitute equations. It consists of $n + m$ equations, defined on n differential and m algebraic variables. One equation is then removed, and is made a substitute equation to calculate the value of a differential variable. However, the variable still needs to be calculated by a solver, since its time derivative occurs without substitution in the remaining equations. What remains to be solved by the DAE solver, is $n + m - 1$ equations and $n + m$ variables.

Only in the theoretical case that the substitute equation is obtained by integration of the time derivative, the solver would still have $n + m$ equations. For example,

$$\begin{aligned} \dot{x} &= 1 \\ , \; x &\leftarrow \tau \end{aligned}$$

where τ is the special variable for time. Such a substitute equation would serve no practical use however.

7.1.4 Left out categories

Finally, two categories have been left out from Table 7.1 in order to reduce the number of categories. This has been done, because there is no special need for variables of these categories. That is, a variable of one of the correct categories can always be used instead. The left out categories are

1. *algebraic prime substituted*
2. *algebraic base-prime substituted*

These two categories define algebraic variables. With other words, if there is a variable of any of these two categories defined in a process, no other variables in the equations of that process depend on the time derivative of this variable. Furthermore, the time derivative form of this variable is not allowed to be used in the discrete-event part. Therefore, there is no particular reason to calculate the time derivative of the variable, consequently, there is no reason to allow these categories. Even so, the value of the time derivative can still be defined by a function.

7.1.5 Example

To illustrate variable categories, consider the equations describing the motion of a cart bouncing to a spring, as depicted in Figure 7.1.

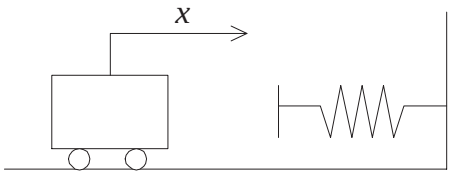

Figure 7.1: Cart-spring system.

Variables x and v denote the position and the speed of the cart, respectively. The cart touches the spring at position x_{set}. The force applied by the spring (variable F), is modelled by a substitute equation. The equations are:

$$
\begin{array}{lcl}
F & = & -m\dot{v} \\
\dot{x} & = & v \\
F & \leftarrow & [\ x < x_{set} \longrightarrow 0 \\
 & & \|\ x \geq x_{set} \longrightarrow k(x - x_{set}) \\
 & &]
\end{array}
$$

The variable categories are as follows: F is an algebraic substituted (ALG-B-SUB) variable, v is an algebraic (ALG) variable, and x is a differential (DIF) variable.

The categories of the variables are determined by their appearance in the equations. This information is collected when the equations are analyzed, and is used further to check whether the variables are used correctly in statements. This check is carried out during the semantic analysis of statements. Variables can be assigned, can be given a guess, or their value can be used in expressions in the discrete-event part of processes. The rules for assignments are as follows. Only differential variables and their time derivatives can be assigned. Substituted forms of variables cannot be assigned, neither can they be given a guess. A new category is introduced: when a steady-state calculation is requested for a differential variable, its category changes to $\mathrm{DIF_{ST}}$. This is summarized in Table 7.3.

category	ass. var.	ass. der.	guess var.	guess der.	use der.	new cat.
ALG	-	-	+	-	-	
DIF	+	$+^a$	-	+	+	$^a\mathrm{DIF_{ST}}$
$\mathrm{DIF_{ST}}$	$+^b$	+	+	-	+	^{b}DIF
ALG-B-SUB	-	-	-	-	-	
DIF-P-SUB	-	-	+	-	+	
DIF-BP-SUB	-	-	-	-	+	

Table 7.3: Use of continuous variables.

7.2 Variable representation in the χ engine

7.2.1 Design requirements

Before describing the representation of the continuous variables in the χ engine, first we discuss the design requirements.

In the discrete phases of the simulation, processes work with local continuous variables. Expressions are evaluated with the value of local variables, assignments change the value of the local variables, etc. Discrete variables are implemented in the χ engine as data members of the derived process classes, and statement methods are defined on them. Therefore, it seems straightforward to define continuous variables also as data members of the process classes, so that the implementation of statements using continuous variables is kept similar to those using discrete variables. On the other hand, when the equations are solved by numerical solvers, all equations of the model need to be gathered and solved simultaneously for all variables. This requires a mechanism to access all equations and continuous variables of the model by the numerical solvers.

To summarize, representation of the continuous variables requires on the one hand a mechanism to access local variables in statements of processes, and, on the other hand, a mechanism to access all variables and equations by the solver. This requirement is met by using two kinds of variables. Processes have their local continuous variables

as data members. These variables are used in statement methods. The local variables are mapped onto a set of global variables that are used to calculate their value, either by a solver, or by substitution. Each time the equations are solved, and a discrete phase or a discrete sub-phase may follow, the value of the global variables is copied to the corresponding local variables.

Connection equations are in the form of $variable_1 = variable_2$. They are straightforward to solve, and need not be calculated by numerical solvers. In the following we present an implementation where these equations are eliminated. However, great care needs to be taken to emulate the connection semantics, as defined in Section 3.12.

Substituted variables are in fact functions. Whenever accessed, they return the value obtained by evaluating the right-hand-side of the substitute equation. Their implementation needs to access local variables of processes.

7.2.2 Structure

The structure of the variable representation in the χ engine is depicted in Figure 7.2. Local variables are represented by the class CLocCVar. For each variable in

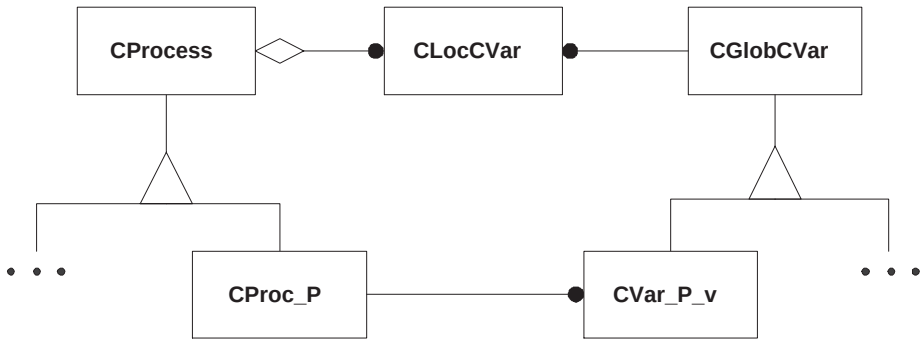

Figure 7.2: Continuous variable representation in the χ engine.

a χ process specification, an object of type CLocCVar is constructed that is a data member of the derived process class. At the same time, a set of global variables is constructed. Global variables are represented by the class CGlobCVar. For each substituted variable, a new class is derived from CGlobCVar, that specifies how the value of the variable and/or its time derivative are calculated. Each local variable is registered in exactly one global variable, which is the one that is used to calculate its value.

The goal is to build an optimized structure, where as few as possible global variables are used. This way the size of the continuous state is reduced. It is achieved by creating only one global variable to calculate connected local variables. Consider, for example, the cart-spring model depicted in Figure 7.1. Suppose, that the cart and

the spring are modelled by separate processes, C and S, respectively. Channels x and F connect the local variables x and F of the two processes, as depicted in the upper part of Figure 7.3. The χ specification of this system is as follows.

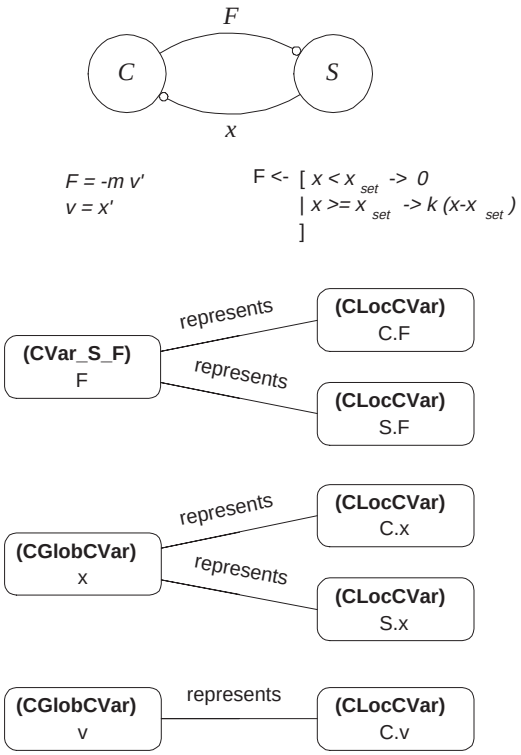

Figure 7.3: Cart-spring system with two processes.

unit $\mathrm{N} = \mathrm{kg} \cdot \mathrm{m/s^2}$

proc $C(F\_ :: \multimap [\mathrm{N}],\ x\_ :: \multimap [\mathrm{m}]) =$
$|\ [\ F :: [\mathrm{N}],\ x :: [\mathrm{m}],\ v :: [\mathrm{m/s}]$
$\rightsquigarrow\ x ::= 2.0;\ v ::= 0$
$\dashv\ F\_ \multimap F$
$,\ x\_ \multimap x$
$=\!\!\dashv\ F = -mv'$
$,\ v\ = x'$
$]\ |$

proc $S(F\_ :: \multimap [\mathrm{N}],\ x\_ :: \multimap [\mathrm{m}],\ k : \mathrm{real}) =$
$|\ [\ F :: [\mathrm{N}],\ x :: [\mathrm{m}],\ x_{set} : \mathrm{real}$
$\rightsquigarrow\ x_{set} := 5.0$

```
⊣ F_ ⊸ F
 , x_ ⊸ x
=⊣ F ← [ x < x_set ⟶ 0
       ‖ x ≥ x_set ⟶ k(x − x_set)
       ]
] |

syst CS() =
| [ c : .C, s : .S
  | F :: −[N], x :: −[m]
  | c(F, x) ‖ s(F, x, 0.2)
] |
```

The continuous variables of this system are $C.F, C.x, C.v, S.F, S.x$. The equations defined by the two processes are depicted in the middle part of the figure. The lower part of the figure shows the instance diagram of the generated local and global variables. For the substituted variable $S.F$, a new global variable class CVar_S_F is derived from CGlobCVar. The optimized structure contains three global variables: one for F, one for x, and one for v. The equations that are actually solved are

$$\begin{aligned} F &= -m\dot{v} \\ \dot{x} &= v \end{aligned}$$

where F is an expression, that has value 0 or $k(x - x_{set})$.

7.2.3 Categorization of global variables

In the χ engine, the continuous variables, both local and global ones, belong to one of the six categories ALG, DIF, DIF_{ST}, ALG-B-SUB, DIF-P-SUB, DIF-BP-SUB. The category expresses how a variable appears in the equations, and in the case of differential variables, whether steady-state initialization has been requested. The category of the local variables is defined in Section 7.1. In the case of local variables, the category is used to determine which operations are allowed on them.

In the case of global variables, the category expresses how the global variable appears in the global equations (the set of equations that is defined on the global variables and is actually solved). This information is used when the initial state is calculated. In the case of non-substituted variables, depending on the category, either the variable or its time derivative is calculated. For global variables, the category can be derived from the category of the local variable(s) that they represent.

If a local variable is not connected, then its global variable represents only one local variable. The category of the global variable, therefore, is the same as that of the local variable.

If local variables are connected, a global variable represents several local variables. The rules to determine the category of the global variable in case of two connected variables are given in Table 7.4. It is assumed, that substituted variables are not connected to each other, so that a global variable represents at most one substituted variable. If this is not the case, an error is issued. The rules in Table 7.4 are used to determine the category of the global variable if more than two variables are connected, in the following way. Take two variables, and determine the category of the global variable. Then add the rest of the local variables one by one. Take the category of the global variable, use it as local variable category, and together with the local variable to be added, determine the new category of the global variable from Table 7.4.

local_1	local_2	global	
ALG	ALG	ALG	
ALG	DIF	DIF	
ALG	DIF_{ST}	DIF_{ST}	
ALG	X-SUB	X-SUB	$X \in$ { ALG-B, DIF-B, DIF-P}
DIF	DIF	DIF	
DIF_{ST}	DIF_{ST}	DIF_{ST}	
DIF/DIF_{ST}	ALG-B-SUB	DIF-B-SUB	not allowed
DIF/DIF_{ST}	DIF-X-SUB	DIF-X-SUB	$X \in$ { P, BP }

Table 7.4: Global variable categories.

If differential variables are connected, a warning is issued at the beginning of the simulation. We will return back to this later. In the case of connected differential variables, only those situations are listed, where the categories of the local differential variables are either all DIF or all DIF_{ST}. The reason for this is that when differential variables are connected and one of the variable's time derivative is assigned (a steady-state calculation is requested), the category of all the connected differential variables and the category of the global variable change to DIF_{ST}. Likewise, when one of the local differential variables is assigned, the category of all connected differential variables and the category of the global variable change to DIF.

The category of the variables of the cart-spring example in Section 7.1.5, depicted in Figure 7.3 is given in Table 7.5.

local_1	local_2	global
$C.F$: ALG	$S.F$: ALG-SUB	F:ALG-SUB
$C.x$: DIF	$S.x$: ALG	x: DIF
$C.v$: DIF		v: DIF

Table 7.5: Variable categories of cart-spring system.

Global variable categories are used during the initial state calculation. The variables of category ALG, DIF_{ST}, DIF-P-SUB, and the time derivatives of the variables of category DIF are calculated by the nonlinear equation solver. A steady-state

initialization request is valid for only one (namely the next) initial state calculation. Therefore, at the beginning of each discrete sub-phase, those variables that had category $\mathrm{DIF_{ST}}$ are changed back to category DIF.

7.2.4 Assignment operations

The assignment operations on continuous variables (assignment and assignment of a guess) are mapped onto the implemented variable structure. The operations always change the value of the local variable or its time derivative. Furthermore, an assignment operation on a local variable or its derivative may change the value of its associated global variable or the time derivative of its global variable, respectively. Finally, some operations change both the category of the variable itself, the categories of the connected differential variables and the category of the associated global variable.

When a discrete sub-phase ends, the initial state calculation starts with the current value and category of the global variables. For local variables that are not connected to a substituted variable, so that the associated global variable is not substituted, the operations are summarized in Table 7.6. The value of the representing global variable is mostly changed, unless an algebraic variable that is represented by a differential global variable is given a guess. In such a case, the algebraic variable is connected to a differential variable. Therefore, the differential variable determines the value of the global variable.

In the case that the derivative of a differential variable is assigned a guess, and its category is $\mathrm{DIF_{ST}}$ an error is issued, because its category indicates that a steady-state initialization has been requested previously. It is possible, that the request was made by another connected differential variable. A similar problem occurs, if a differential variable of category DIF is assigned a guess.

local	operation	global	action	
			change glob.	change cat. to
ALG	guess var	ALG	+	-
		DIF	-	-
		$\mathrm{DIF_{ST}}$	+	-
DIF	assign var.	DIF	+	-
	assign der.	DIF	+	$\mathrm{DIF_{ST}}$
	guess var.	DIF	error	
	guess der.	DIF	+	-
$\mathrm{DIF_{ST}}$	assign var.	$\mathrm{DIF_{ST}}$	+	DIF
	assign der.	$\mathrm{DIF_{ST}}$	+	-
	guess var.	$\mathrm{DIF_{ST}}$	+	-
	guess der.	$\mathrm{DIF_{ST}}$	error	

Table 7.6: Operations with non-substituted global variables.

A more complex situation arises, when substituted variables are connected, thus, the global variable is substituted. The current implementation of the operations are summarized in Table 7.7. The guidelines for the implementation were to keep the behaviour of substituted variables similar to the behaviour of non-substituted variables as far as possible. This, however is not always possible. A substituted differential variable (DIF-BP-SUB and DIF-P-SUB) cannot be initialized to its steady-state. Therefore, an attempt to calculate the steady-state value of a differential variable that is connected to a substituted differential variable results in an error. This is also the reason for not having any rules in Table 7.7 with the local variable having category $\mathrm{DIF}_{\mathrm{ST}}$. Furthermore, re-initialization of a differential variable that is represented by a global variable of DIF-BP-SUB also fails. This is because in this case, the value of the global variable itself is calculated by substitution. Unless the value that is assigned to the variable is equal to the value that is calculated by substitution (which is rather superfluous) this operation results in an error. Again, the value of the local variable or its time derivative is updated by all operations. The implementation is not definitive yet, and requires further research on connection semantics of substituted variables.

local	operation	global	change glob.
ALG	guess var.	ALG-B-SUB	-
		DIF-P-SUB	+
		DIF-BP-SUB	-
DIF	assign var.	DIF-P-SUB	+
		DIF-BP-SUB	(error)
	assign der.	DIF-P-SUB/DIF-BP-SUB	error
	guess var.	DIF-P-SUB/DIF-BP-SUB	error
	guess der.	DIF-P-SUB/DIF-BP-SUB	-

Table 7.7: Operations with substituted global variables.

7.2.5 Connected differential variables

When differential variables are connected, a warning is given at the beginning of the simulation. This is to draw the attention of users to the fact that the variables are dependent on each other and their value can no longer be independently chosen. In this way, the system's degree of freedom is reduced. From a numerical point of view, it often means, that the index of the DAEs is increased (see Section 2.2.1), leading to an equation set that is substantially more difficult to solve. Representing connected differential variables by only one global variable, this problem is solved. In fact, the index of the equations is reduced in this way.

Plenty of examples for connected differential variables can be found in rigid body systems. Suppose, that each component of such a system is modelled by a χ process. Typically, each process has variables x and y to denote the x and y coordinates of

the represented body (in $\mathbb{R}^2$). As the velocity of the bodies is in most cases are of interest, these variables are differential. The coordinates of the bodies can be chosen such that physical connection means that the two variables are equal.

When differential variables are connected, and more than one are assigned in one discrete sub-phase, they have to be assigned the same value. This is required because the assignment $y ::= e$, at time point t, where y is a differential variable, means according to the semantics of assignments that integration starts in the next continuous phase with $y(t) = e$. This assertion cannot be maintained if connected differential variables are re-initialized to different values.

7.2.6 The CLocCVar class

The class CLocCVar implements the local continuous variables of processes. On the one hand, these objects are used in the statement methods that implement the discrete-event behaviour of processes. In this context, the objects have to exhibit local behaviour and shield the underlying global variable set. On the other hand, objects of CLocCVar are used during the continuous phases, when they have to allow access to the underlying global variable structure. In these phases they must become 'transparent' to allow access to the global variables. The class is depicted in Figure 7.4.

Attributes dVar and dPrime store the local value of the variable and its time derivative. Attribute uType encodes the category. Each object has a reference of the global variable that represents it. This pointer is passed by the Link method, after construction. The methods AssignVar, AssignPrime, GuessVar and GuessPrime implement assignment and assignment of a guess as defined in Tables 7.6 and 7.7. These four methods are used in the statement methods of processes. The value of the variables and their time derivatives are returned by the methods Var and Prime. A boolean parameter indicates, whether the local value, or the value of the global variable should be returned. At the end of a continuous phase, and each time after a consistent initial state is calculated, the value of the global variables are distributed to the local ones. For this purpose, the Set method is used, to set the variable and its time derivative.

7.2.7 The CGlobCVar class

The CGlobCVar class, depicted in Figure 7.4, implements global variables. For each substituted variable, a new class is derived from CGlobCVar. Global variables may represent several local variables. The local variables are added by the method AddLocal, and the optimized structure is achieved by invoking the Merge operation. References to the local variables are sorted into two lists; one containing the algebraic variables, the other one containing the differential ones. The category of the global variables is represented by the cKind attribute. Non-substituted variables belong to

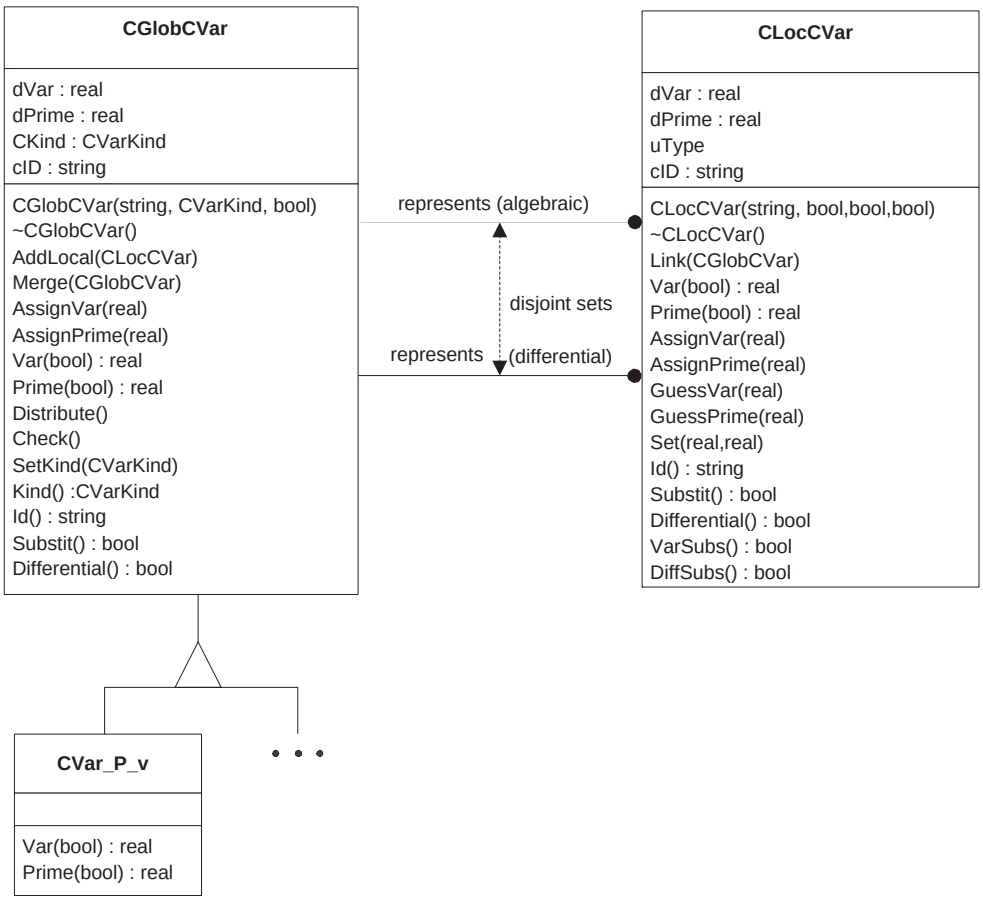

Figure 7.4: The CLocCVar and the CGlobCVar classes.

the categories ALG and DIF. The methods AssignVar and AssignPrime are invoked by the local variables, when they execute an assignment statement, or a guess is given to a local variable. The methods Var and Prime return the value of the variable and its time derivative, respectively. At the end of a continuous phase, and each time after a consistent initial state is calculated, the value of the global variables are distributed to the local variables by the method Distribute. If more connected differential variables are initialized simultaneously, they have to be initialized to the same value. This is checked at the end of each discrete sub-phase by the Check method.

In case of substituted variables, (categories ALG-B-SUB, DIF-BP-SUB, DIF-P-SUB), a new class is derived from CGlobCVar. Its name is constructed from the process name and the variable name. These classes re-implement the methods Var and Prime. The implementation is such that the order of the substitute equations in the χ model does not matter. Also, the order of the Var and Prime methods in the generated C++ code does not matter; substitute equations are automatically evaluated in the right order.

This is realized as follows. Consider, for example, the substitute equation

$$F \leftarrow \begin{bmatrix} x < x_{set} \longrightarrow 0 \\ \| \; x \geq x_{set} \longrightarrow k(x - x_{set}) \\ \end{bmatrix}$$

The class CVar_P_F is defined to represent variable F. The Var method is defined in C++ as follows:

```
double  CVar_P_F::Var(bool bGl)
{
  if ((pProc->var_x).Var(bGl)< pProc->var_x_set)
   dVar =0.0;
  else
   dVar=(pProc->var_k * ((pProc->var_x).Var(bGl) - pProc->var_x_set));
  return dVar;
}
```

The local variables of process P are available via the pointer pProc. Each substituted variable is automatically recalculated whenever its Var method is called.

7.3 Equations

7.3.1 Equation representation in the χ compiler

The class hierarchy of the equations as represented in the χ compiler is shown in Figure 7.5. The base class CEquation implements common features of equations. Attribute iEqNumber is the total number of base equations defined by the equation, and iCounter is the index of the first equation method to be generated for the given equation (to be explained later). The type of the equation (base, guarded, or substituted) is stored in the cEqKind attribute. Base equations are represented by the class CBaseEq that has two members of type CExpression, representing the left and the right expressions. The class CGuardedEq represents guarded equations. It has several alternatives, each of type CEqAlternative. This class has a member of type CExpression representing the guard, and a list of equations, represented by class CEqList. Substitute equations are represented by the class CSubstituteEq. The boolean attribute bDeriv denotes whether a variable or the time derivative of a variable is substituted. The main method of the classes in the equation hierarchy is the TypeChk method that implements semantic analysis, and counts the base equations represented by the given construct. Code generation is implemented separately by class CCppEqGenerator that operates on this structure.

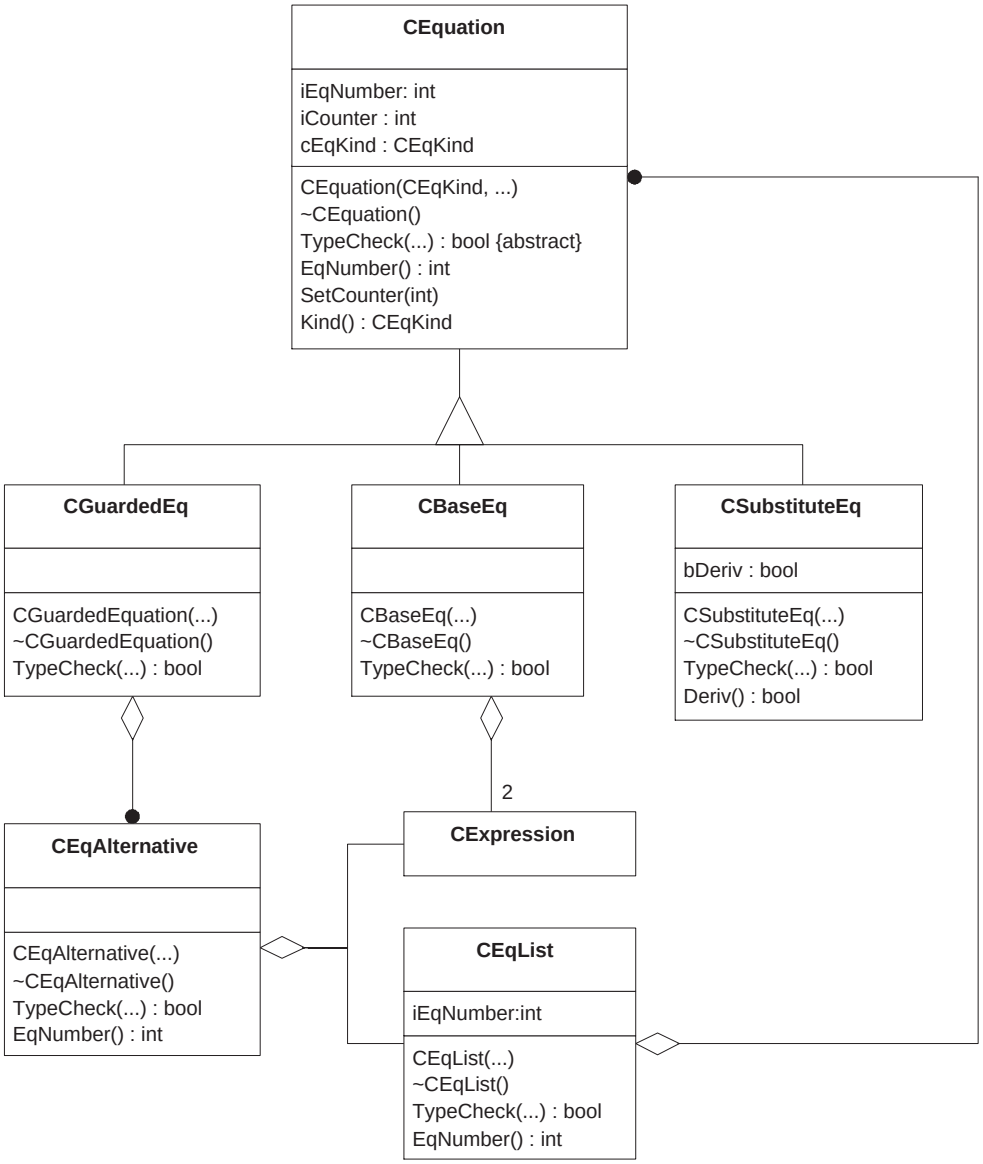

Figure 7.5: Equation representation in the χ compiler.

7.3.2 Equation methods

As introduced in Section 4.1, the equations of a hybrid system can be formulated as

$$\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t) = \mathbf{0}, \tag{7.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the vector of differential variables, $\mathbf{y} \in \mathbb{R}^m$ is the vector of algebraic variables, $t \in \mathbb{R}$ is the independent variable and $\mathbf{f} \in \mathbb{R}^{2n+m+1} \rightarrow \mathbb{R}^{n+m}$ is the set of DAEs. Numerical solvers require $n+m$ functions that define f_i, $i \in [1 \ldots n+m]$. At each step, the solvers return the current approximated value of the variables. After that, the equations f_i are evaluated with these values, and the residual is returned to the solver. Although the DAE solver and the nonlinear solver solve these equations for different variables, the two solvers work with the same implementation of f_i. This is possible, because the unknowns of the equations can be chosen in a flexible way.

In the code generation phase, the equations of the process specifications are mapped onto functions f_i, $i \in [1 \ldots n+m]$. This is done similar to the way statements are implemented in processes. Equations of a χ process specification are numbered and mapped onto *equation methods* of the derived process classes. Each equation method defines one function f_i. To be able to traverse all $n+m$ functions, an array of tuples is constructed in the class CScheduler. Each element in the array identifies a process, and an equation method index.

Base equations are mapped easily to C++ functions: one side of the equation needs to be subtracted from the other side. For example, the following base equation

$$Ah' = Q,$$

in process P is mapped onto the equation method

```
double CProc_P::eq0(void)
{
return ( var_A * var_h.Prime(true) - var_Q.Var(true) );
}
```

In guarded equations, the C++ 'if...then...else' construct implements alternatives. For example, the following guarded equation defines two equation methods.

$$\begin{array}{l} [\ b \ \longrightarrow Ah' = Q_i - Q_o, Q_x = 0 \\ [\!] \ \neg b \longrightarrow Ah' = 0, Q_x = Q_i - Q_o \\ ] \end{array}$$

The second one is mapped onto the following equation method:

```
double CProc_P::eq1(void)
{
if (b)
  return (var_Q_x.Var(true)-0.0);
else if (!b)
  return (var_Q_x.Var(true)-(var_Q_i.Var(true)-var_Q_o.Var(true)));
else
  pSched->Exception("No guard is true in a guarded equation");
}
```

In this implementation, the piece-wise definition of the function is hidden from the DAE solver. As long as the defined equation f remains continuous, the DAE solver can integrate it. Take, for example, a valve controlling the flow of a fluid (variable Q), that can be adjusted linearly between fully open and fully closed. The control input of the valve is represented by variable u. The following guarded equation models the valve

$$\begin{array}{l} [\ u \geq 1 \quad\ \longrightarrow Q = Q_{max} \\ \|\ 0 \leq u \leq 1 \longrightarrow Q = uQ_{max} \\ \|\ u \leq 0 \quad\ \longrightarrow Q = 0 \\ ] \end{array}$$

In this case, Q can be calculated without the need to stop the integration when the value of u crosses 0 or 1, so that a different equation becomes valid.

7.4 Future work

In the future, equations should be represented in such a way that they can be analyzed – and possibly manipulated – symbolically. Symbolic analysis can have several goals. The most important ones are:

- To identify high index problems and reduce the index. There are several approaches to do this. We deal with techniques of index reduction in Section 9.5.
- To reduce the size of the problem that needs to be solved by numerical solvers. The variables need to be sorted out according to their computational causality, and the equations may need to be manipulated. The goal of this operation – achieved in the ideal situation – is to bring the equation matrix into a lower triangular form. If this is possible, all variables can be computed by substitution. However, there can be circular dependency between variables that remains to be solved numerically. It means, that the matrix of the equations is of block lower triangular form. In this case, numerical solution and substitution must be combined to calculate the variables. For this, sparse matrix techniques can be used, see e.g., [Duff et al., 1986].
- To identify the type of the equations and choose a suitable numerical algorithm. The following characterization can be of interest: ODE/DAE, constant coefficient system, time-varying system, linear/non-linear system, etc.
- To derive additional information about the equations (e.g., Jacobian matrix) analytically. In order to solve the equations, much of additional information may be required about the system. These are more accurate if derived analytically, and not approximated, thus the chance for successful solution is increased. For

example, our experience is that the nonlinear equation solver NLEQ has much better convergence performance if the analytical Jacobian matrix is used.

In the future, the connection semantics needs to be extended to special cases. These are connected substituted variables and connected differential variables. The current implementation is in both cases a first approach. It needs to be investigated how the behaviour of connected substituted variables can be unified within the continuous semantics with the behaviour of non-connected substituted variables. The same needs to be investigated for connected differential variables and differential variables that are not connected to other differential variables.

Chapter 8

Numerical calculations, state-events

In the χ simulator, numerical calculations and state-event location are carried out by solvers. During a continuous phase, the DAEs are solved by DDASRT [Petzold, 1983]. It is a special version of DASSL, the widely used DAE solver, that is equipped with root finding functionality. Root finding is used to locate state-events. The initial state problem is solved by NLEQ [Novak and Weiman, 1991], a nonlinear equation solver that implements a damped Newton method.

In this chapter, the two solvers are introduced, with emphasis on their impact on hybrid simulation. Furthermore, the implementation of state-event specifications (so-called nabla statements) is explained.

8.1 DASSL

DASSL [Petzold, 1983] is a DAE solver designed to solve index zero and index one systems of DAEs. The χ simulator integrates DDASRT, which is the double-precision counterpart of DASSL, equipped with root finding capabilities. This chapter is organized as follows. First, the algorithm that DASSL implements is described briefly. Then, those aspects of DASSL that are relevant to its integration and use in a hybrid simulator are discussed. These are computation cost, stepsize selection, convergence and accuracy, discontinuity handling and root finding. Many practical tips about the use of DASSL can be found in [Brenan et al., 1996]. These apply in general to numerical problems, therefore we do not repeat them here. Other sources of information about the difficulties that may arise using DASSL, are e.g., [Sincovec et al., 1981, Petzold, 1982, Jarvis, 1993, Wijckmans, 1996].

The equations solved by DASSL are in the form

$$\mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, t) = \mathbf{0}, \tag{8.1}$$

where $\dot{\mathbf{x}}, \mathbf{x}$ are N dimensional vectors and $\mathbf{f}$ is a vector valued function. Note, that no difference is made between differential and algebraic variables. DASSL implements a backward differentiation formula (BDF) method. The general idea of this method,

as described by Gear [Gear, 1971], is to replace the time derivative of $\mathbf{x}$ in (8.1) by a backward difference formula, and solve the resulting equation by Newton's method or one of its variants. For example, using the first order backward difference formula, which is the implicit Euler method, the equation is

$$\mathbf{f}\left(\frac{\mathbf{x}_n - \mathbf{x}_{n-1}}{h_n}, \mathbf{x}_n, t_n\right) = 0. \tag{8.2}$$

This equation is solved for $\mathbf{x}_n$ at the current mesh point t_n by Newton's method

$$\mathbf{x}_n^{m+1} = \mathbf{x}_n^m - G^{-1}\mathbf{f}\left(\frac{\mathbf{x}_n^m - \mathbf{x}_{n-1}}{h_n}, \mathbf{x}_n^m, t_n\right), \tag{8.3}$$

where m is the iteration index, and the iteration matrix G is defined as

$$G = \frac{1}{h_n}\frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\,. \tag{8.4}$$

In DASSL, this approach is generalized. The k^{th} backward difference formula is used, where k may range from 1 to 5. The stepsize h_n and the order k varies, depending on the behaviour of the solution. The k-step BDF method on the n^{th} integration step can be represented as

$$\sum_{j=0}^{k} \alpha_j \mathbf{x}_{n+j-k} = h_n \beta_k \dot{\mathbf{x}}_n, \tag{8.5}$$

where the past values of $\mathbf{x}$, ($\mathbf{x}_{n+j-k}$, $j = 0, \ldots, k-1$) are known. This equation may be re-arranged to give:

$$\dot{\mathbf{x}}_n = \frac{1}{h_n \beta_k}\left(\alpha_k \mathbf{x}_n + \sum_{j=0}^{k} \alpha_j \mathbf{x}_{n+j-k}\right), \tag{8.6}$$

which can be rewritten as

$$\dot{\mathbf{x}}_n = \frac{1}{h_n \beta_k}(\alpha_k \mathbf{x}_n + \gamma_{k,n}), \tag{8.7}$$

where $\gamma_{k,n}$ is the collection of terms in the previous step. Substituting this into the DAE (8.1) yields

$$\mathbf{f}\left(\frac{1}{h_n \beta_k}(\alpha_k \mathbf{x}_n + \gamma_{k,n}), \mathbf{x}_n, t_n\right) = 0. \tag{8.8}$$

This equation is then solved for $\mathbf{x}_n$. As the stepsize varies, computed past values of $\mathbf{x}$ may be unequally spaced, whereas in the above formula, equally spaced past values are used. This problem is solved using the fixed leading-coefficient strategy. First, an initial guess is obtained for $\mathbf{x}$ and $\dot{\mathbf{x}}$ at t_n ($\mathbf{x}_n^0, \dot{\mathbf{x}}_n^0$), by evaluating a predictor

polynomial $\omega_n^P(t)$ and the derivative of the predictor polynomial at time point t_n. The predictor polynomial is the polynomial which interpolates $\mathbf{x}_n$ at the last $k+1$ values on the unequally spaced mesh

$$\omega_n^P(t_{n-i}) = \mathbf{x}_{n-i} \qquad i = 1, \ldots, k+1. \tag{8.9}$$

Then, a corrector polynomial is constructed, and the solution which is finally accepted by DASSL, is the solution to the corrector polynomial. The corrector polynomial and its derivative satisfy the DAE (8.1) at t_n, and it interpolates the predictor polynomial at k equally spaced points behind t_n. The corrector equation (determined by the definition of the corrector polynomial) is solved using a modified Newton iteration. In order to do this, the iteration matrix of equation (8.8) is necessary, which is

$$G = \alpha \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} + \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \tag{8.10}$$

where

$$\alpha = \frac{\alpha_k}{h_n \beta_k} \tag{8.11}$$

is a constant that changes whenever the stepsize or order changes. The cost of the integration is dominated by the calculation of the iteration matrix. This is especially true when solving large problems. Therefore, if the iteration matrix is not very much different from the last computed iteration matrix, then the old iteration matrix is used. The iteration matrix can either be computed by finite differences, or supplied directly by the user. In the χ simulator, the first option is used. It is the author's opinion [Brenan et al., 1996, page 132], that the weakest part of the DASSL code is the finite difference Jacobian calculation. Therefore, in the future, if the equations are analyzed and manipulated symbolically, the χ simulator should calculate and provide the analytical Jacobian to DASSL.

DASSL solves a problem from time T to time TOUT, where T and TOUT are specified by the user. Often, the user desires the numerical output at a set of output points. In this case, DASSL runs in a loop, that increments TOUT until the end of the interval. Note, that in this case DASSL is not re-started, it only continues the integration on each call. To obtain output at points that are not at the timesteps chosen by DASSL, the solution between two steps is interpolated. The χ simulator calls DASSL at the end of a discrete phase to integrate the DAEs until the next discrete phase. The values of the continuous variables can be examined at equidistant points during the simulation. The distance of the output points, (the stepsize of the simulation) can be controlled by the *stepsize* simulation option, see Appendix B. Therefore, TOUT is the next discrete event determined by delta statements, i.e., the minimum of the time-outs, or, if output is requested before the next discrete-event, the next output point.

8.1.1 Stepsize and order selection

On the first step, DASSL takes the first order BDF formula, and a small step. The initial stepsize is calculated from the formula

$$h_0 = \text{sign}(\text{TOUT} - T) \cdot \min\left(10^{-3}|\text{TOUT} - T|, \frac{1}{2} \parallel \dot{\mathbf{x}} \parallel^{-1}\right). \tag{8.12}$$

In many χ models, small time-outs are used, with lengths of only a few seconds. In these cases, the first term in the $\min(10^{-3}|\text{TOUT} - T|)$ is at most 10^{-2}, and therefore, the initial stepsize is at most 10^{-2}. After the first step, the order and the stepsize are gradually increased. DASSL uses the order and stepsize selection strategy described in [Shampine and Gordon, 1975]. This strategy tends to favor sequences of constant stepsize and order. At each step, the order and the stepsize of the next step is determined. When the stepsize is increased, it is increased by factor of 2, when it is decreased it is decreased by a factor between 0.9 and 0.25. As the initial stepsize may be very small, and is increased by at most a factor of 2 at each step, it takes DASSL several small steps to reach a stable big step integration.

The smallest step that DASSL can take at time step t_n is defined as

$$h_{min} = 4\, u \max(|t_n|, |\text{TOUT}|) \tag{8.13}$$

where u is the unit roundoff error. On PCs, we use double precision floating point numbers to represent reals, thus $u = 2^{-52}$. This leads to

$$h_{min} = 2^{-50}\max(|t_n|, |\text{TOUT}|) \sim 8.88 \cdot 10^{-16}\max(|t_n|, |\text{TOUT}|). \tag{8.14}$$

As it has already been stated in Section 5.3.1, the χ simulator takes the minimum stepsize into consideration. If the next discrete phase is closer to the current time than the minimum stepsize, then the simulation time is not advanced, and the next discrete phase is scheduled at the current time. The way the minimum stepsize is selected (formula 8.14), affects long simulations, i.e., simulations, where the simulation time becomes large. A bigger minimum stepsize is defined when a large interval is integrated in one session, compared to the case when it is divided into smaller time intervals. Namely, in the second case TOUT may be much smaller, at least, in the first few steps at the beginning of the interval. Furthermore, the minimum stepsize grows in the course of the simulation.

8.1.2 Convergence, accuracy

BDF methods have good stability and accuracy properties. Convergence results underlying the methods used in DASSL are described in [Gear et al., 1981, Brenan, 1983, Gear et al., 1985, Lötstedt and Petzold, 1986a, Brenan and Petzold, 1988]. In particular, for fully implicit index one problems, the k-step BDF method of fixed stepsize h for $k < 7$ converges to $O(h^k)$ if all initial

values are correct to $O(h^k)$ and if the Newton iteration on each step is solved to accuracy $O(h^{k+1})$. This convergence result has been extended to variable stepsize BDF methods, provided that they are implemented in such a way that the method is stable for standard ODEs [Gear et al., 1985].

In DASSL, at each step, the local error is estimated. This estimate is used to decide whether to accept the current step or to redo the step with a smaller stepsize. The solution to the corrector iteration after m steps ($\mathbf{x}_n^m$) is accepted if it satisfies the error test

$$C \parallel \mathbf{x_n^m} - \mathbf{x_n^0} \parallel \leq 1 \tag{8.15}$$

where C depends on the order and stepsize. The local error can be controlled by users, by controlling the norm in (8.15). DASSL uses a weighted root mean square norm given by

$$\parallel \mathbf{v} \parallel = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(v_i/\mathrm{WT}_i)^2} \tag{8.16}$$

where

$$\mathrm{WT}_i = \mathrm{RTOL}_i|x_i| + \mathrm{ATOL}_i. \tag{8.17}$$

The relative and the absolute tolerances, RTOL and ATOL, must be provided by the user. In equation (8.17), the value of $\mathbf{x}$ is used at the beginning of the current step ($\mathbf{x}^0$). In the χ simulator, two simulation options are available to set the relative and the absolute tolerances. These values are used for all components in $\mathbf{x}$. However, it is desirable to set the tolerances for variables individually, especially, in cases when the variables are scaled much differently from each other. Future versions of the χ simulator should facilitate this.

8.1.3 Consistent initial values, discontinuity

Initial values of $\mathbf{x}$ and $\dot{\mathbf{x}}$ must be consistent, otherwise, DASSL may fail on the first step. To illustrate this, consider the equation $x = g(t)$, where $x(0) = a \neq g(0)$. On the first step, DASSL takes the first order BDF formula, that gives $x_1 = g(t_1)$. In the error estimate (8.15), $C = 0.5$, therefore, the following test is made on each step:

$$0.5 \parallel x_1 - x^0 \parallel \leq 1. \tag{8.18}$$

Since $x^0 = a$, the term $\parallel x_1 - x^0 \parallel$ converges to a constant as the stepsize approaches 0 ($t_1 \to t_0$). Depending on the norm, the test may never be satisfied.

There is an option in DASSL to compute the initial value of $\dot{\mathbf{x}}$, if the initial value of $\mathbf{x}$ is known, and an initial guess for $\dot{\mathbf{x}}$ is supplied. In this case, DASSL takes a small implicit Euler step, and uses a damped Newton iteration to solve the system. In early

versions of the hybrid χ simulator, this option was used to calculate the initial value of $\dot{\mathbf{x}}$. However, this option does not work together with the root finding functionality. This is explained in the next section.

A problem similar to that of inconsistent initial values may arise if a discontinuity occurs in any of the components of $\mathbf{x}$. Consider the equation $\dot{x} = g(x,t)$, with a solution $x = h(t, x_0)$. Suppose, that a discontinuity occurs at time t_d: x is abruptly changed to d, where $d \neq \lim_{t \uparrow t_d} h(t, x_0)$. In any step that goes through the discontinuity, the error test (8.15) may fail. The reason for this is that the initial guess obtained for x behind t_d has a value for x as if no discontinuity occurred. This is because the initial guess is obtained by evaluating the predictor polynomial (8.9), which predicts no discontinuity. However, the corrector polynomial satisfies the DAE, therefore, the solution to the corrector iterator (x^m) takes the discontinuity into account. Therefore, $x^m - x^0$ tends to a constant as the stepsize approaches zero. As a result, the error test may never be satisfied. DASSL, therefore, needs to be restarted each time a discontinuity occurs. It can, however, usually solve discontinuities in the time derivative. In this case, the stepsize is lowered until the error test succeeds. Still, many steps are rejected, and it takes many small steps to go through the discontinuity; this may make this approach inefficient.

8.1.4 Root finding

DASSLRT is the root finding version of DASSL. A set of root functions $\mathbf{g}(\mathbf{x}, t)$, $\mathbf{g} \in \mathbb{R}^{N+1} \rightarrow \mathbb{R}^{GN}$ (also called switching functions) can be given, that is monitored while solving equation (8.1). The root finding mechanism works as follows. Whenever a step to t_n is successfully completed, the root function $\mathbf{g}(\mathbf{x}_n, t_n)$, is evaluated. If the sign of $\mathbf{g}(\mathbf{x}_n, t_n)$ is different from the sign of $\mathbf{g}(\mathbf{x}_{n-1}, t_{n-1})$, then a root must have occurred in the interval $[t_{n-1}, t_n]$. The root is then located, using the 'Illinois' algorithm [Dahlquist and Björk, 1974], which is a modified secant method. For root finding, the interpolant of $\mathbf{f}$ is used to calculate the values of $\mathbf{x}$ between t_{n-1} and t_n. In the χ simulator, root finding is used to monitor state-event specifications. The mapping of state-event specifications onto functions g_i, $i = 1, \ldots NG$, is described in more detail in Section 8.3.

The problem with using root finding together with the calculation of the initial value of $\dot{\mathbf{x}}$ is as follows. Suppose, that in a χ model two state-events need to be monitored simultaneously in a continuous phase. Suppose, that the roots occur very close to each other, one at time point t_1, the other one at time point $t_1 + \varepsilon$, where $\varepsilon \ll 1$. This is often the case if analytically the roots occur at the same time, but due to numerical errors in the integration one occurs before the other one. DASSLRT detects the first root at time t_1. After this, a discrete phase takes place, and then DASSLRT is re-started. Suppose, that the second root still needs to be located. If the initial value of $\dot{\mathbf{x}}$ needs to be calculated, then DASSLRT takes a small implicit Euler step of length δ, $\delta \ll 1$, to determine $\dot{\mathbf{x}}$. The actual integration and root finding starts after this, at time $t_1 + \delta$. Therefore, if $\delta > \varepsilon$, then the second root is over stepped, and is not

detected by DASSLRT. In general, any root that occurs between t_1 and $t_1 + \delta$ goes undetected. This leads to fatal errors in the simulation of χ models. In the current χ simulator, the initial state is calculated by a separate solver, and thus this problem is solved.

8.1.5 Strategy of using DASSL in the χ simulator

Re-starting DASSL is computation intensive. First, large computation arrays need to be initialized. Second, DASSL always starts with a small stepsize that increases with a maximum factor of 2. Because the order increases in the start phase as well, the iteration matrix needs to be recomputed frequently in this phase. Therefore, our strategy of using DASSL in the χ simulator is to re-start it only if it is necessary. It is only re-started if a discontinuity occurs in the variables, or if different set of state-events need to be monitored.

8.2 NLEQ

The initial state problem as described in Chapter 4 is solved by NLEQ [Novak and Weiman, 1991], a non-linear equation solver. NLEQ implements the affine invariant Newton techniques of Deuflhard [Deuflhard, 1974]. The standard method is implemented in the code NLEQ1, whereas the code NLEQ2 contains a rank reduction device additionally. In the χ simulator, both versions are available. We have chosen this solver to calculate initial states for two reasons. First, in NLEQ the Newton techniques are combined with a special damping strategy, in order to create global convergence. This is very important in hybrid models, because, after a discontinuity, often there are no good initial guesses available for all variables. In these cases, simple Newton methods would not converge. The other reason for using the NLEQ family is that these codes are able to solve highly non-linear systems. This is especially true for NLEQ2, where the additional rank reduction technique extends the convergence domain of the code. In [Novak and Weiman, 1991], several test cases illustrate that NLEQ is especially suited to solve large and numerically sensitive problems.

Yet, there is a need to supply initial guesses to variables, although, they do not need to be highly accurate. This is the reason why in the χ language an initial guess can be given to variables.

8.3 Nabla statements

Nabla statements are used in the χ language to specify state events. Currently, nabla statements may only have the form

$$\nabla e_1 < e_2 \mid \nabla e_1 \leq e_2 \mid \nabla e_1 > e_2 \mid \nabla e_1 \geq e_2$$

where e_1 and e_2 are numerical expressions. At least one of the expressions e_1 and e_2 must contain a continuous variable, and they may not contain a time derivative of a variable. The latter restriction is due to the fact that root functions of DASSL may not depend on time derivatives of variables. Note that equality can not be specified in nabla statements. This is because due to numerical inaccuracy, equality is almost never met when such a root is located. The above requirements are checked by the χ compiler during semantic analysis.

8.3.1 The CNablaStat class

Nabla statements are translated by the χ compiler into root functions for DASSL. The root function generated for any of the simple expressions that can be used in nabla statements is $g(t) = e_1(t) - e_2(t)$.

In the χ engine, the general functionality of nabla statements is implemented by the class CNablaStat, depicted in Figure 8.1. For each nabla statement, a new class is derived from CNablaStat, and an object of the new class is constructed. In each process, an array of CNablaStat objects is created. The index of the nabla object identifies the represented nabla statement inside the process.

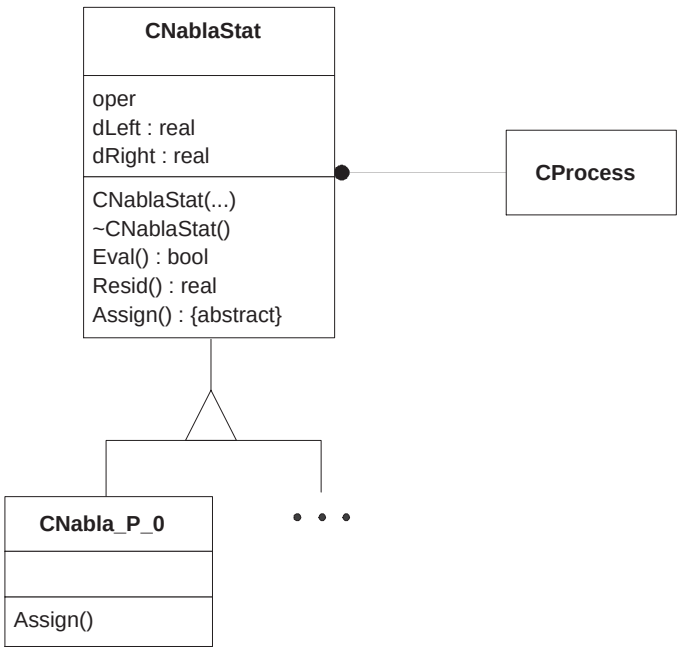

Figure 8.1: The CNablaStat class.

In the class CNablaStat, attributes dLeft and dRight store the values of the left and the right expressions e_1 and e_2. Attribute oper can be one of the four possible operators '$<$', '$\leq$', '$>$' or '$\geq$'. The abstract method Assign specifies the way dLeft and dRight

are calculated. This method is re-defined in each derived class. The Eval method evaluates the state-event specification. This method is used to determine, whether a nabla statement blocks the execution of the process. Finally, method Resid calculates the root function.

Consider, for example, the nabla statement

$$\nabla h > 100$$

specified somewhere in the discrete-event body of process P. Nabla statements are numbered uniquely. If this is the first nabla statement occurring in process P, then class CNabla_P_0 is defined, and its Assign method is defined as

```
void CNabla_P_0::Assign()
{
  Left = (pProc->var_h).Var(true);
  Right = 100.0;
};
```

The local variables of process P are available via the pointer pProc.

8.3.2 Representation of nabla blockings

In the hybrid scheduler, the set of nabla blockings, described in Section 5.4, has been implemented. In this set, information is stored about processes that are blocked by a nabla statement. Each nabla blocking is a process state, and, in particular, it encodes the nabla statement where the process is currently blocked, and the statement that follows the blocking nabla statement. In the χ engine, process references implement process states used in the scheduling specification. This is the case for nabla blockings as well. A nabla blocking is represented by objects of class CNbElem shown in Figure 8.2. The attributes of this class are a process reference (it is depicted as an association in Figure 8.2), a nabla index (Ind), and the program counter of the statement that follows the nabla statement (Next).

The set of nabla blockings is not only used for scheduling, but it also collects the set of root functions that need to be monitored at a given time. Therefore, this data structure is also used to traverse the root functions during continuous phases.

A process may wait for more nabla statements at the same time, if those statements are events of a selective waiting statement. If an alternative becomes enabled and is chosen for execution, all nabla blockings that belong to that process must be removed from the set of nabla blockings. Therefore, the representation of the set of nabla blockings must support removal of all nabla blockings that belong to one process.

The data structure that implements the set of nabla blockings is depicted in Figure 8.2. Nabla blockings, represented by CNbElem, that represent different states of the

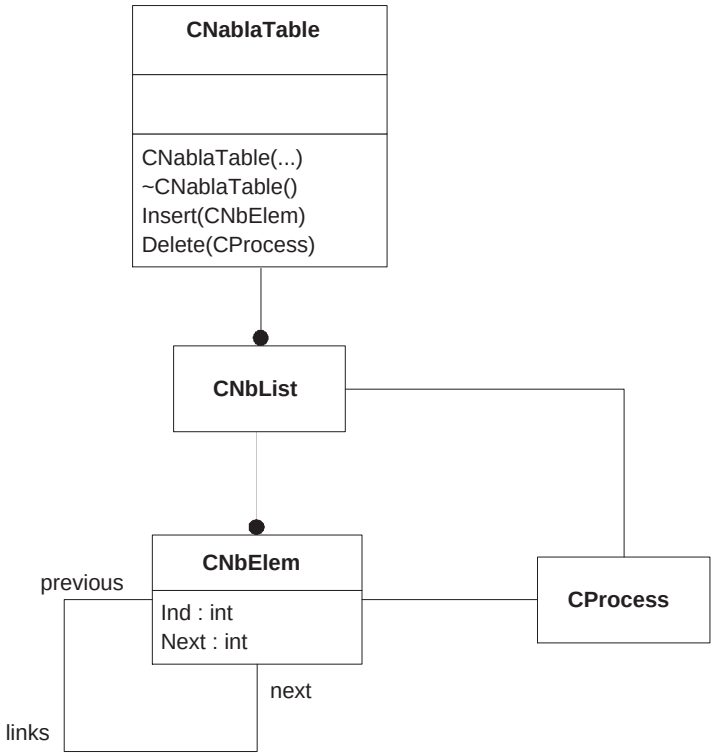

Figure 8.2: The representation of nabla blockings.

same process, are collected into lists (CNbList). These lists are stored in a table (CNablaTable), where the process (reference) is the key. The Insert method inserts individual nabla blockings. The Delete method deletes all nabla blockings that belong to one process. Within the table a linked list is maintained to allow fast traverse for root function evaluation during the continuous phases.

8.4 Future work

In the future, the χ simulator shall integrate more solvers to solve the equations in the continuous phases. By having several solvers available, users of the simulator itself may choose the solver most suitable to the problem.

New solvers may be selected to enlarge the problem domain. In [Liniger, 1979], a method is described that is more stable than BDF methods when applied to very nonlinear problems. DASOLV [Jarvis and Pantelides, 1992] is a DAE solver that can solve systems with fast transients efficiently, that is, with fewer steps and with less rejected steps than DASSL. (Fast transients are fast reactions, such as flow through bursting disks, pressure surges following a slug, etc.) There are algorithms

that can solve some high index problems, for example the modified Radau5 code [Hairer et al., 1980]. In general, one-step methods, such as Runge-Kutta methods have the advantage over BDF methods that they can restart at a high order. This can be useful in the case of high index systems and in the case of systems with frequent discontinuities.

New solvers may also be selected to increase performance. In many models, the equations are a system of ODEs, or even a system of linear ODEs. In this case, an ODE solver works much faster than DASSL.

It has been pointed out in Section 8.1.1 that the minimum stepsize of DASSL increases in the course of the simulation. This affects long simulations adversely. Towards the end of the simulation, the solver is less capable of integrating fast changing systems than at the beginning. In future DAE solvers, the minimum stepsize should be selected such that this effect is removed, or users should be given a possibility to remove it.

Chapter 9

Applications

In this chapter, a collection of χ models are presented that have been selected to illustrate the potential application fields of combined discrete-event/continuous-time simulation. The purpose of these examples is also to demonstrate the flexibility and the descriptive nature of the χ language. The simulation results have been obtained using the χ simulator.

In Section 9.1, a dry friction example is presented, to illustrate discontinuity handling in the χ language. In Section 9.2, different control algorithms are specified in a tank level control example. These examples have been taken from [van Beek and Rooda, 1998b] and [van Beek and Rooda, 1997], respectively. The next example is a filling station, which shows a χ model consisting of several processes. This example combines a discrete-event buffer process for trucks with a hybrid buffer process for liquids. It also models stochastic arrival patterns. This example is used in the under-graduate course 'Production and Process Systems' (see `http://se.wtb.tue.nl`). The last example is a consumer/distributor model, which is an example of a combined discrete/continuous model with chaotic behaviour. This system is from [Hoffmann and Engell, 1998]. The χ model is inspired by the work of J.P.M. Schmitz.

Finally, in the Section 9.5, the use of substitute equations is illustrated. When building the current χ simulator, the aim was to be able to solve systems of index 0 and index 1 DAEs with no hidden constraints. It turns out that it is possible to simulate such systems with the current χ simulator by using substitute equations. The examples demonstrate index reduction and consistent initialization of DAEs that have hidden constraints.

9.1 Dry friction

The following example describes the motion of a body on a flat surface with friction. It is a purely mechanical system with discontinuities.

A sinusoid driving force F_{d} is applied to a body on a flat surface with frictional force F_{f}. When the body is moving with positive velocity v, the frictional force is given by

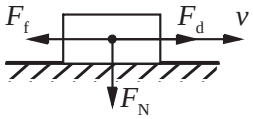

Figure 9.1: Dry friction.

$F_f = \mu F_N$, where $F_N = mg$. When the velocity of the body is 0, the frictional force neutralises the applied driving force. If the driving force becomes bigger than $\mu_0 F_N$, the body suddenly starts moving according to $F_d - F_f = mv'$, where $F_f = \mu F_N$ ($\mu < \mu_0$). In process *Friction*$_1$, defined below, discrete variable s represents the state of the process; it can have the values "neg" , "stop", and "pos". These values correspond with negative, zero, and positive velocities of the body, respectively. As a result of the different equations for the three states, variable F_f is discontinuous. Variable F_N is declared as a discrete variable (F_N : real), because its value remains constant.

```
const ε : real = 10^-5
,     g : real = 9.86

proc Friction1(ε, μ, μ0, m : real) =
| [ Ff, Fd :: [N], x :: [m], v :: [m/s]
  , s : string
  , FN : real
~| x ::= −2; v ::= 0; FN := mg; s := "stop"
=| Fd = sin 0.25πτ
  , v' = (Fd − Ff)/m
  , x' = v
  , [ s = "neg"  ⟶ Ff = −μFN
    ‖ s = "stop" ⟶ Ff = Fd
    ‖ s = "pos"  ⟶ Ff = μFN
    ]
  | *[ s = "stop" ; ∇ Fd > μ0FN  ⟶ s := "pos"
                                 ; v ::= ε
     ‖ s = "stop" ; ∇ Fd < −μ0FN ⟶ s := "neg"
                                 ; v ::= −ε
     ‖ s ≠ "stop" ; ∇ v = 0      ⟶ s := "stop"
     ]
] |
```

The discrete-event part of process *Friction*$_1$ consists of a selective waiting statement that is repeated forever. If boolean expression $s =$ "stop" is true and state-event $\nabla F_d > \mu_0 F_N$ succeeds, then the state switches to "pos" ($s :=$ "pos"), causing a different equation to be active in the continuous-time part. The velocity v is then assigned a very small value ε in order to prevent the state-event $\nabla v = 0$ from occurring immediately. Now that the state equals "pos", boolean expression $s \neq$ "stop" is true.

This means that state-event $\nabla v = 0$ is awaited. When v becomes equal to 0, the state changes to "stop" again.

Process *Friction*$_2$, that follows below, is a more detailed model. It models that a small amount of energy is required to switch from state "stop" to state "pos" or "neg". For this purpose, a fourth state "try" is introduced. If $|F_\mathrm{d}|$ becomes bigger than $\mu_0 F_\mathrm{N}$, the state switches to "try". In this state, the frictional force equals $\mu_0 F_\mathrm{N}$. If the driving force causes the velocity of the body to become bigger than ε, the state switches to "pos". If, however, the driving force falls back below $\mu_0 F_\mathrm{N}$, the state switches back to "stop". Figure 9.2 shows the results of a 10 second simulation run for the process *Friction*$_1$ and *Friction*$_2$. The difference between the results of the two processes is too small to be shown graphically.

```
const ε : real = 10^-5
,     g : real = 9.86

proc Friction2(ε, μ, μ0, m : real) =
| [ Ff, Fd :: [N], x :: [m], v :: [m/s]
  , s : string
  , FN : real
~| x ::= −2; v ::= 0; FN := mg; s := "stop"
  | Fd = sin 0.25πτ
  , v' = (Fd − Ff)/m
  , x' = v
  , [ s = "neg"  ⟶ Ff = −μFN
    ‖ s = "stop" ⟶ Ff = Fd
    ‖ s = "try"  ⟶ Ff = sign(Fd) · μ0FN
    ‖ s = "pos"  ⟶ Ff = μFN
    ]
  | *[ s = "stop" ; ∇|Fd| > μ0FN
       ⟶ s := "try"
         ; [ ∇v > ε          ⟶ s := "pos"
           ‖ ∇v < ε          ⟶ s := "neg"
           ‖ ∇|Fd| ≤ μ0FN ⟶ s := "stop" ; v ::= 0
           ]
     ‖ s ≠ "stop" ; ∇v = 0 ⟶ s := "stop"
     ]
] |
```

9.2 Tank level control system

The following system illustrates by means of several kinds of controllers, how a discrete-event controller of a nonlinear continuous system can be specified in χ. The

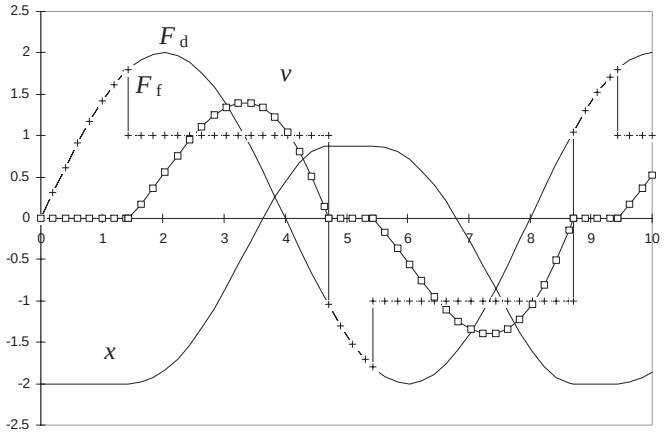

Figure 9.2:Friction simulation ($\mu = 0.1$, $\mu_0 = 0.18$, $m = 1$, $\varepsilon = 10^{-5}$, $g = 10$). The x axis denotes time in [s], F_d and F_f are measured in [N], x is measured in [m] and μ is scalar.

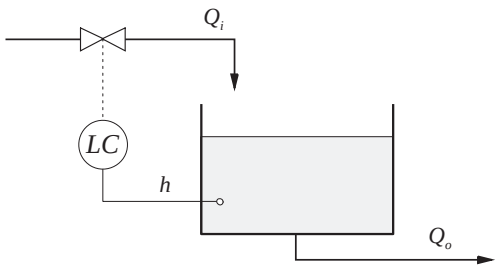

Figure 9.3: The tank level control system.

controlled nonlinear system is a tank shown in Figure 9.3. The control system tries to keep the level in the tank close to a set point h_{set}. There is a continuous flow out of the vessel (Q_o). The value of this flow depends on the height of the liquid (h) in the vessel. The controller controls the level of the tank by adjusting the position of the valve. In the discrete-event controller of the next section the valve can only be switched on and off. In the discrete-time controllers of subsequent sections the position of the valve can be adjusted linearly between 1 (fully open) and 0 (closed). The tank equations are derived by means of the formula

$$\sum p_{\text{source}} = \sum p_{\text{loss}}$$

which leads to

$$\rho g h = K_{\text{L}} Q^2,$$

where K_{L} is the loss factor. This can be rewritten as

$$Q = \sqrt{\frac{\rho g h}{K_{\text{L}}}} = k\sqrt{h}.$$

9.2.1 On-off control

The first controller is a discrete-event controller that keeps the liquid between two levels, using a hysteresis band h_{hys} of $0.1m^3$. If the liquid in the tank drops below $h_{\mathrm{set}} - h_{\mathrm{hys}}$, the valve is opened. If the liquid in the tank rises above h_{set}, the valve is closed.

The continuous variables of the model, h and Q_{o}, denote the height of the liquid and the outgoing flow, respectively. Their values are determined by the equations

$$\begin{aligned} Ah' &= \alpha Q_{\max} - Q_{\mathrm{o}} \\ Q_{\mathrm{o}} &= k\sqrt{h}. \end{aligned}$$

The valve is modelled by the variable α. When the valve is open, α equals 1, when it is closed, α equals 0. The height of the tank h is initialized to h_0 ($h ::= h_0$). The controller consists of a repetition. Initially, $h = h_0 = 0.1$ so that $h < h_{\mathrm{set}} - h_{\mathrm{hys}}$ is true. Therefore, the first nabla statement succeeds and the next statement is executed immediately. The valve is switched on which is modelled by the assignment $\alpha := 1$. When the level rises above h_{set} ($\nabla\, h > h_{\mathrm{set}}$), the incoming flow is switched off.

$$\begin{array}{l} \mathsf{proc}\ TCD(A, h_0, h_{\mathrm{set}}, h_{\mathrm{hys}}, k, Q_{\max} : \mathrm{real}) = \\ |\ [\ h :: [\mathrm{m}],\ Q_{\mathrm{o}} :: [\mathrm{m}^3/\mathrm{s}] \\ \quad , \alpha : \mathrm{int} \\ \sim|\ h ::= h_0;\ \alpha := 0 \\ =|\ Ah' = \alpha Q_{\max} - Q_{\mathrm{o}} \\ \quad ,\ \ Q_{\mathrm{o}} = k\sqrt{h} \\ \quad |\ *[\ \nabla\, h < h_{\mathrm{set}} - h_{\mathrm{hys}};\ \alpha := 1;\ \nabla\, h > h_{\mathrm{set}};\ \alpha := 0\ ] \\ ]\ | \end{array}$$

Figure 9.4 shows the results of a simulation run of the following experiment.

$$\mathsf{xper}\ = |\ [\ tcp : .TCP\ |\ tcp(10, 0.1, 1, 0.5, 20, 1, 0.1)\ ]\ |$$

9.2.2 PI control

In this example, the on/off valve used in the previous section is replaced by a valve that can be adjusted linearly between fully open and closed. In this section, a PI (proportionally integrating) control is specified. The control input of the valve is represented by variable u. The valve is modelled by a guarded equation.

$$\begin{array}{ll} [\ u \le 0 & \longrightarrow Q_{\mathrm{i}} = 0 \\ [\!]\ 0 \le u \le 1 & \longrightarrow Q_{\mathrm{i}} = uQ_{\max} \\ [\!]\ u \ge 1 & \longrightarrow Q_{\mathrm{i}} = Q_{\max} \\ ] & \end{array}$$

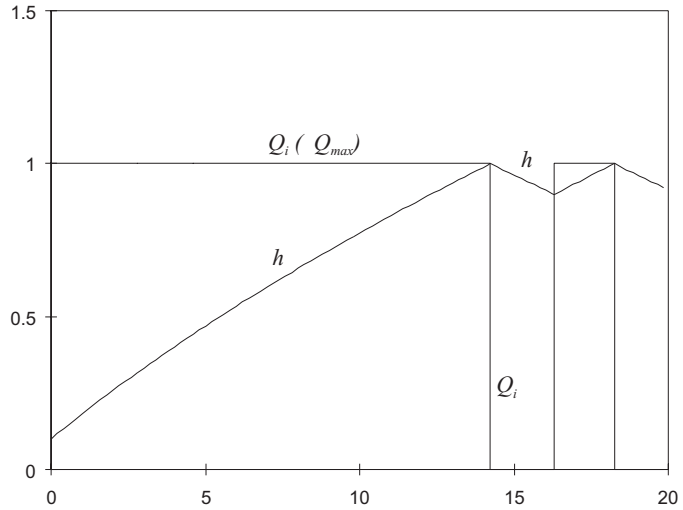

Figure 9.4:Discrete-event on/off control. The x axis denotes time in [s], h is measured in [m] and Q_i is measured in [m^3/s].

It is assumed that the valve operates in its linear mode for $0 \leq u \leq 1$ ($0 \leq u \leq 1 \longrightarrow Q_{\mathrm{i}} = uQ_{\max}$). If the valve is fully opened ($u \geq 1$), the input flow equals $Q_{\max}$. If the output u becomes negative ($u < 0$), the flow becomes zero.

Process *TCPI* specifies a discrete-time controller, because it takes a sample of the level of the liquid every t_s second. At every sample time, the error e, and the integral of the error I_e are calculated. The control variable is then determined as $u := K_{\mathrm{p}}e + K_{\mathrm{i}}I_e$, where .

$$
\begin{array}{l}
\mathsf{proc}\ TCPI(A, h_0, h_{\mathrm{set}}, k, K_{\mathrm{i}}, K_{\mathrm{p}}, Q_{\max}, t_{\mathrm{s}} : \mathrm{real}) = \\
|\ [\ h :: [\mathrm{m}],\ Q_{\mathrm{i}}, Q_{\mathrm{o}} :: [\mathrm{m}^3/\mathrm{s}] \\
\quad , e, I_e, u : \mathrm{real} \\
\sim|\ h ::= h_0; I_e := 0 \\
=|\ Ah' = Q_{\mathrm{i}} - Q_{\mathrm{o}} \\
\quad ,\ \ Q_o = k\sqrt{h} \\
\quad , [\ u \leq 0 \quad\ \longrightarrow Q_{\mathrm{i}} = 0 \\
\quad\ \|\ 0 \leq u \leq 1 \longrightarrow Q_{\mathrm{i}} = uQ_{\max} \\
\quad\ \|\ u \geq 1 \quad\ \longrightarrow Q_{\mathrm{i}} = Q_{\max} \\
\quad\ ] \\
\quad |\ *[\ e := h_{\mathrm{set}} - h \\
\quad\ \ ;\ I_e := I_e + et_{\mathrm{s}} \\
\quad\ \ ;\ u := K_{\mathrm{p}}e + K_{\mathrm{i}}I_e \\
\quad\ \ ;\ \Delta t_{\mathrm{s}} \\
\quad\ \ ] \\
]\ |
\end{array}
$$

The results of the simulation run of the following experiment are shown in Figure 9.5.

$$\mathsf{xper} = |\ [\ tcpi : .TCPI \mid tcpi(10, 0.1, 1, 0.5, 2, 20, 1, 0.1)\ ]\ |$$

It can be seen that the settling time is rather long and there is a considerable overshoot. This integrator windup is caused by the fact that the integrator keeps integrating when the output u saturates ($u > 1$).

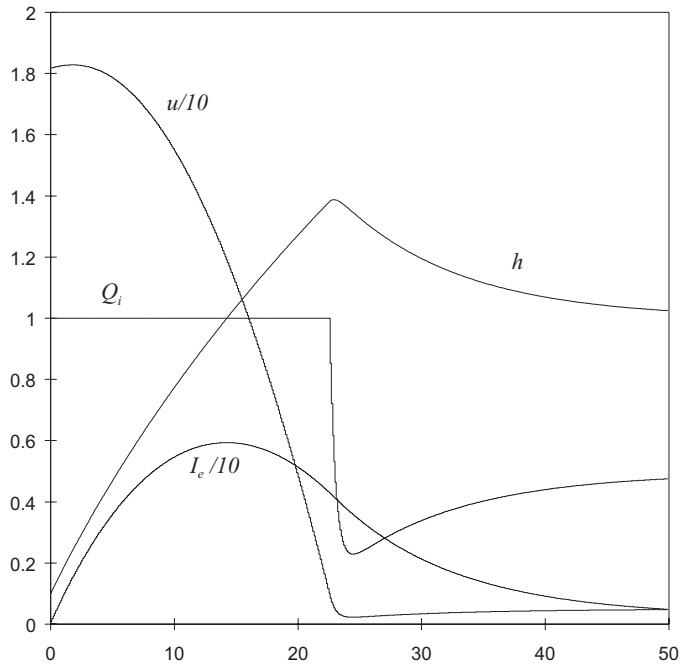

Figure 9.5:PI control ($K_\mathrm{p} = 20$, $K_\mathrm{i} = 2$). The x axis denotes time in [s], Q_i is measured in [m^3/s], h is measured in [m] and u and I_e are scalars.

9.2.3 PI control with anti-windup

Process *TCPIA* shown below uses an anti-windup strategy for the discrete-time controller. The amount of saturation of the output ($(u - 1)$ for $u > 1$, and u for $u < 0$) is subtracted from the integral term I_e in order to reduce the saturation effect. This may initially lead to negative values of I_e, as is shown in Figure 9.6. The values shown in the figure are the values of the variables just before and just after the $\Delta\, t_\mathrm{s}$ statement.

$$\mathsf{proc}\ TCPIA(A, h_0, h_\mathrm{set}, k, K_\mathrm{i}, K_\mathrm{p}, Q_\mathrm{max}, t_\mathrm{s} : \mathrm{real}) =$$
$$|\ [\ h :: [\mathrm{m}],\ Q_\mathrm{i}, Q_\mathrm{o} :: [\mathrm{m}^3/\mathrm{s}]$$

```
  , e, I_e, u : real
~| h ::= h_0; I_e := 0
=| Ah' = Q_i − Q_o
  ,  Q_o = k√h
  , [ u ≤ 0        ⟶ Q_i = 0
    ‖ 0 ≤ u ≤ 1⟶ Q_i = uQ_max
    ‖ u ≥ 1        ⟶ Q_i = Q_max
    ]
  | *[ e := h_set − h
     ; I_e := I_e + K_i e t_s
     ; u := K_p e + I_e
     ; [ u < 0        ⟶ I_e := I_e − u
       ‖ 0 ≤ u ≤ 1⟶ skip
       ‖ u > 1        ⟶ I_e := I_e − (u − 1)
       ]
     ; Δt_s
     ]
] |
```

```
xper = | [ tcpia : .TCPIA | tcpia(10, 0.1, 1, 0.5, 10, 20, 1, 0.1) ] |
```

9.3 Filling station

The following system is the filling station of a plant that produces some kind of liquid. Its χ model is a mixed discrete-event/continuous-time system that consists of several processes.

The liquid is transported by trucks each carrying a barrel of capacity V_{bar}. The trucks arrive at random intervals to the filling station. Only one truck at a time can be filled. Other trucks must wait for their turn at a waiting station.

In the plant, the liquid is stored in a buffer tank, from which the barrels on the trucks are filled. The buffer tank can hold a maximum amount of liquid V_{high}. When its contents reaches V_{high}, the incoming flow is switched off; when its contents drops below the lower threshold V_{low}, the incoming flow is switched on again. The incoming flow rate and the barrel filling flow rate are constants. Filling of a barrel starts only if there is enough liquid for one barrel in the buffer tank. Time is measured in hours in this model.

The χ model depicted in Figure 9.7 consists of four processes. A generator process G models the arrival of trucks. The waiting statition is modelled by process B. Actual filling is modelled by process *Fill*, and finally, process BT models the buffer tank.

Arrival of a truck is signalled by process G to process B via channel trk_0. In process G a negative exponential distribution of $1/t$ is used to model the average inter arrival time t of the trucks.

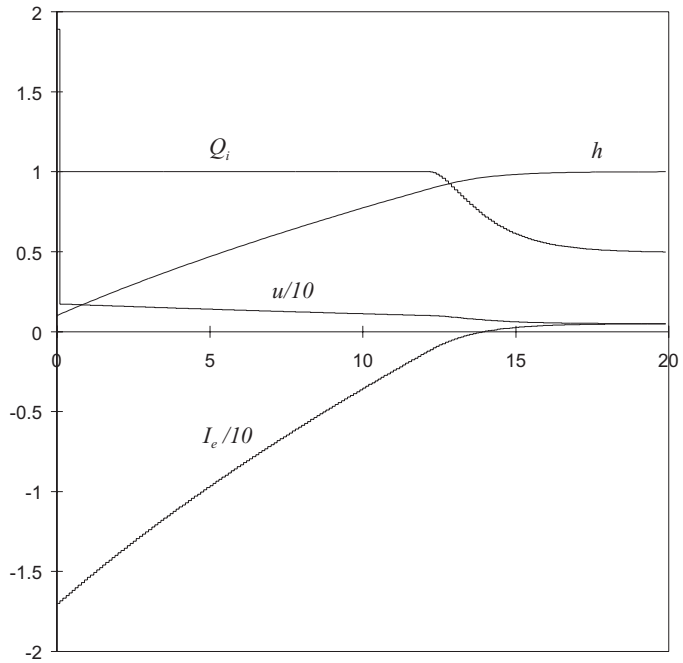

Figure 9.6:PI control with anti-windup ($K_\mathrm{p} = 20$, $K_\mathrm{i} = 10$). The x axis denotes time in [s], Q_i is measured in [m^3/s], h is measured in [m] and u and I_e are scalars.

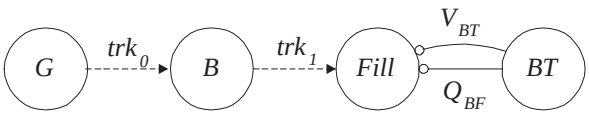

Figure 9.7: Filling station.

```
type vol  = [m^3]
,    flow = [m^3/s]

proc G(trk_0 : ~ void, t : real) =
| [ d : ⟶ real
  | d := nex(1/t)
  ; *[ Δσd; trk_0 ~ ]
] |
```

In process B, the queue of trucks is modelled by variable xs, which is a list of real numbers. The trucks are identified by their arrival time. The process executes a loop of a selective waiting statement with two alternatives. The first alternative is that a truck arrives ($\sim trk_0$), the second alternative is that a truck may move further for

filling ($\sim trk_1$). Each time a truck leaves or arrives, the current time and the number of trucks at the station are saved in the output file f_1. When one truck leaves, its waiting time is calculated ($t_{wait} := \tau - hd(xs)$)[1] together with the average waiting time ($t_{sum} := t_{sum} + t_{wait}$; $t_{avg} := t_{sum}/n$). These numbers are saved in the output file f_2, for further analysis.

$$
\begin{array}{l}
\mathsf{proc}\ B(trk_0, trk_1 : {}^{\sim}\,\mathrm{void},\ f_1, f_2 : !\,file) = \\
|\ [\ xs : \mathrm{real} \\
\quad ,\ n : \mathrm{nat},\ t_{wait}, t_{sum}, t_{avg} : \mathrm{real} \\
\rightsquigarrow |\ xs := [\,]\,;\ n := 0\,;\ t_{sum} := 0\,;\ t_{avg} := 0 \\
\quad ;\ f_1\,!\,\text{“time”},\ \mathrm{tab}(),\ \text{“nbuffer”},\ \mathrm{nl}() \\
\quad ;\ f_2\,!\,\text{“twait”},\ \mathrm{tab}(),\ \text{“tavg”},\ \mathrm{nl}() \\
\quad |\ f_1\,!\,\tau,\ \mathrm{tab}(),\ \mathrm{len}(xs),\ \mathrm{nl}() \\
\quad |\ *[\qquad\qquad trk_0\,{}^{\sim} \longrightarrow xs := xs +\!\!+ [\tau] \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ f_1\,!\,\tau,\ \mathrm{tab}(),\ \mathrm{len}(xs),\ \mathrm{nl}() \\
\qquad [\!] \ \mathrm{len}(xs) > 0;\ trk_1\,{}^{\sim} \longrightarrow n := n + 1\,;\ t_{wait} := \tau - \mathrm{hd}(xs) \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ t_{sum} := t_{sum} + t_{wait}\,;\ t_{avg} := t_{sum}/n \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ xs := \mathrm{tl}(xs) \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ f_1\,!\,\tau,\ \mathrm{tab}(),\ \mathrm{len}(xs),\ \mathrm{nl}() \\
\qquad\qquad\qquad\qquad\qquad\quad ;\ f_2\,!\,t_{wait},\ \mathrm{tab}(),\ t_{avg},\ \mathrm{nl}() \\
\quad\ ]\ \\
]\ |
\end{array}
$$

Process BT is a buffer tank with a simple on/off controller, similar to the one described previously in Section 9.2.1. Since the outgoing flow rate ($3.6\,\mathrm{m}^3/\mathrm{h}$ determined by process *Fill*) is bigger than the incoming flow rate ($0.5\,\mathrm{m}^3/\mathrm{h}$), the contents of the buffer tank does not necessarily remain above the lower threshold.

$$
\begin{array}{l}
\mathsf{proc}\ BT(Q_{BF\_} ::\ \multimap \mathrm{flow},\ V_{BT\_} ::\ \multimap \mathrm{vol}) = \\
|\ [\ Q_{BF} :: \mathrm{flow},\ V :: vol \\
\quad ,\ a, V_{high}, V_{low} : real \\
\rightsquigarrow |\ V ::= 0\,;\ a := 0\,;\ V_{high} := 2\,;\ V_{low} := 1.8; \\
\dashv |\ Q_{BF\_} \multimap Q_{BF} \\
\quad ,\ V_{BT\_} \multimap V \\
= |\ V' = a \cdot 0.5 - Q_{BF} \\
\quad |\ *[\ \nabla V < V_{low}\,;\ a := 1\,;\ \nabla V \geq V_{high}\,;\ a := 0\ ] \\
]\ |
\end{array}
$$

Process *Fill* represents the actual filling. Variables V and V_{BT} represent the contents of the currently filled barrel and the contents of the buffer tank, respectively. The capacity of the barrels is denoted by V_{bar}. When a truck arrives from the waiting station ($\sim trk_1$), first it is connected to the buffer tank. This takes 1.5 minutes. Filling

[1] In χ, τ denotes the current time.

can start if there is enough liquid for one barrel in the buffer tank ($\nabla\, V_{BT} > V_{bar}$). When the barrel is full ($V \geq V_{bar}$), the filling flow is switched off ($a := 0$) and variable V is reinitialized to 0 ($V ::= 0$), so that a new truck may move in.

$$
\begin{array}{l}
\mathsf{proc}\ \mathit{Fill}(Q_{BF\_} :: \multimap \mathrm{flow},\ V_{BT\_} :: \multimap \mathrm{vol},\ \mathit{trk}_1 : {}^{\sim}\,\mathrm{void}) = \\
|\ [\ Q_{BF} :: \mathrm{flow},\ V, V_{BT} :: \mathit{vol} \\
\quad , a, V_{bar} : \mathit{real} \\
\sim|\ V ::= 0;\ a := 0;\ V_{bar} := 0.2 \\
\dashv|\ Q_{BF\_} \multimap Q_{BF} \\
\quad , V_{BT\_} \multimap V \\
=|\ V' = Q_{BF} \\
\quad , Q_{BF} = a \cdot 3.6 \\
\quad |\ *[\ \mathit{trk}_1\,{}^{\sim} \\
\qquad ;\ \Delta 1.5/60 \\
\qquad ;\ \nabla\, V_{BT} > V_{bar};\ a := 1;\ \nabla\, V \geq V_{bar};\ a := 0 \\
\qquad ;\ V ::= 0 \\
\qquad]\ | \\
]\ |
\end{array}
$$

The parameter of the system is the average inter arrival time.

$$
\begin{array}{l}
\mathsf{syst}\ S(t : \mathrm{real}) = \\
|\ [\ g : .G,\ b : .B,\ bt : .BT,\ \mathit{fill} : .\mathit{Fill} \\
\quad , Q_{BF} :: -\mathrm{flow},\ V_{BT} :: -\mathrm{vol},\ \mathit{trk}_0, \mathit{trk}_1 : -\mathrm{void} \\
\quad |\ g(\mathit{trk}_0, t) \parallel b(\mathit{trk}_0, \mathit{trk}_1, \text{“buffer.log1”}, \text{“buffer.log2”}) \\
\quad \parallel bt(Q_{BF}, V_{BT}) \parallel \mathit{fill}(Q_{BF}, V_{BT}, \mathit{trk}_1) \\
]\ |
\end{array}
$$

The results of a simulation with parameter $t = 0.3$ can be seen in Figure 9.8. The capacity of the barrels is $0.2\,\mathrm{m}^3$, and the buffer tank is filled with flow rate $0.5\,\mathrm{m}^3/\mathrm{h}$, so one barrel of liquid is produced in 0.4 hour. Therefore, the trucks arrive faster than the plant can serve them, so that they pile up in the waiting station. The upper panel of Figure 9.8 depicts the increasing number of trucks at the waiting station with increasing waiting time. In the lower panel of Figure 9.8, the contents of the buffer tank can be seen. It shows a regular pattern because the barrels are filled continuously without interruption.

In the second simulation, an average inter arrival time of $t = 0.5$ has been used. Figure 9.9 shows the simulation results. The number of trucks at the waiting station, plotted in the upper panel seldom exceeds one. The waiting time, depicted in the middel panel is rather small, with a few random peaks. The contents of the buffer tank is depicted in the lower panel of Figure 9.9. The buffer tank is often completely full (at level 2 m^3).

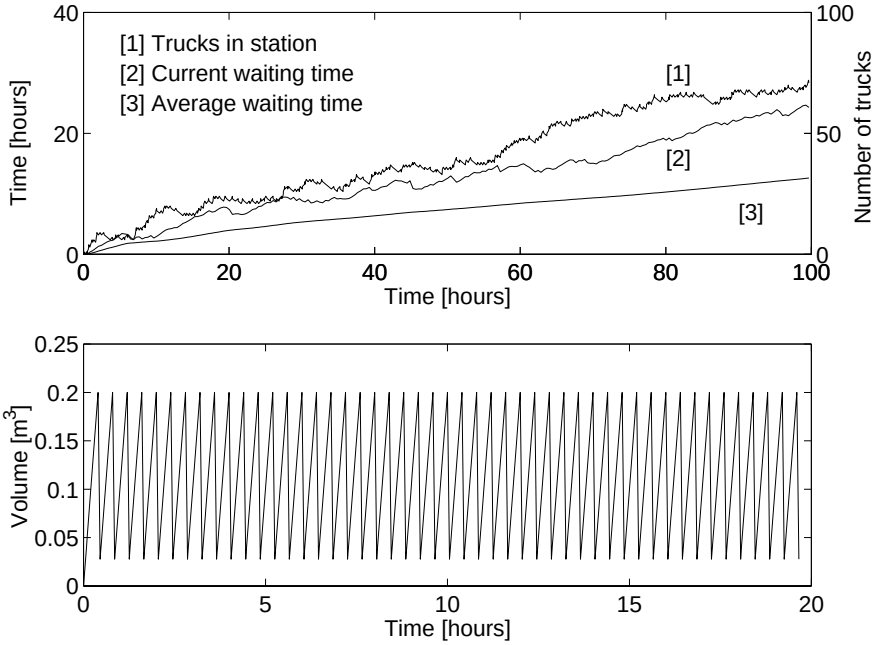

Figure 9.8:Filling station simulation ($t = 0.3$). Top: number of trucks in waiting station and waiting time. Bottom: contents of buffer tank.

9.4 Consumer/distributor system

The following consumer/distributor system (c/d system) has been described in [Hoffmann and Engell, 1998]. Such a system consists of a number of consumers (modelled as tanks) and a distributor which delivers the commodity (modelled as some kind of a liquid) according to the requests of the consumers. The interesting feature of this system is that it is chaotic.

In our realization, the system consists of three tanks and a server depicted in Figure 9.10. The server refills one tank at a time, at a constant rate. Each tank has an open outlet therefore, if the tank is not being refilled then its content decreases with time. The control objective is to keep the substance in tank i between the lower threshold L_{i0} and the upper threshold L_{i1}. The control strategy examined in [Hoffmann and Engell, 1998] is as follows. Initially, the server is in position j. It remains there and fills tank j until either the content reaches its upper threshold L_{j1}, or another tank, say k, reaches its lower threshold L_{k0}. If the latter occurs, the server swithes instantaneously to k and remains there, filling the tank until another tank reaches its lower threshold, and so on. If one tank is filled to its upper threshold while no other one has reached its lower threshold, the server stops filling and moves to an idle position until another tank reaches its lower threshold. The content x_i of tank i

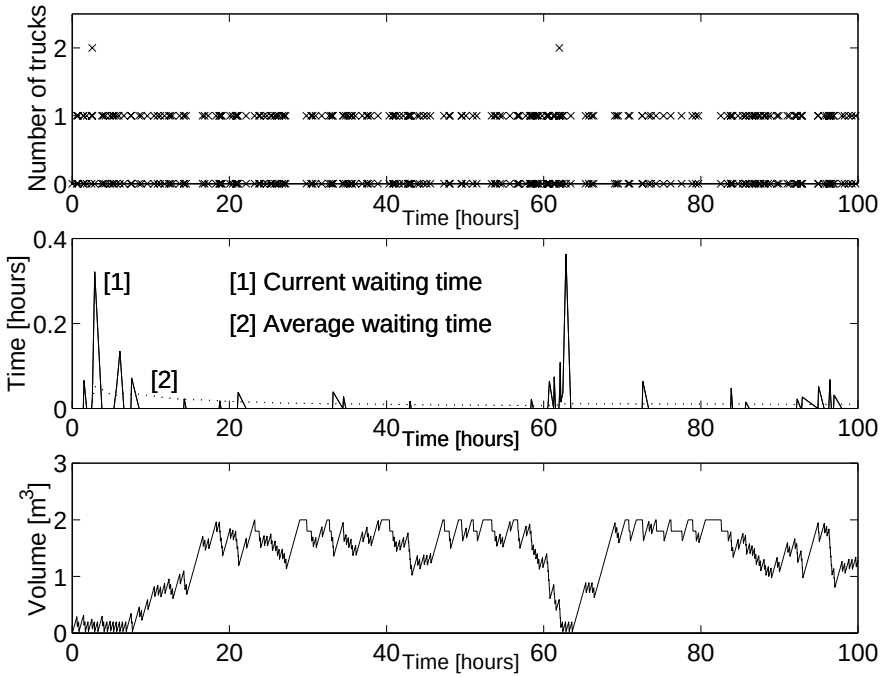

Figure 9.9:Filling station simulation ($t = 0.5$). Top: number of trucks in waiting station. Middle: waiting time. Bottom: contents of buffer tank.

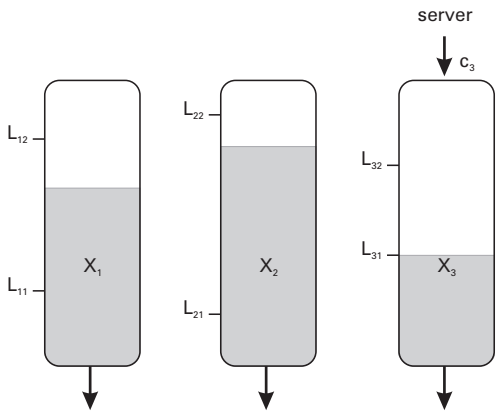

Figure 9.10: Consumer/disctributor system.

is described by the equation

$$\dot{x}_i(t) = c_i\delta_{ij} - \rho_i x_i^{\alpha}(t). \tag{9.1}$$

The first term represents the constant filling of tank i with rate c_i. The switching of the server is represented by δ_{ij}, where $i \in \{0,1,2\}$ is the tank number and $j \in \{0,1,2,3\}$

is the position of the server. When $j = 3$, the server is in the idle position.

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

The second term in (9.1) describes the outflow. Variables ρ_i and α are constants. In the simulation below, $\alpha = 0.5$ is used.

The χ model of this system consists of one process only. The tanks are represented by three equations that describe the way their contents change (equation 9.1). In this model, arrays of continuous variables are used. Array V represents the contents of the tanks and arrays Q_i and Q_o represent the incoming and the outgoing flows, respectively. The current position of the server is modelled by an array n, where $n.i = 1$ if the server is in position i, and $n.i = 0$ for all $i \in 0, 1, 2$ when the server is in the idle position. The operations on the arrays are described using enumerators (variable k). The whole system was modelled as shown below:

$$\begin{array}{l}
\mathsf{proc}\ S(\alpha : \mathrm{real},\ \rho, c, V_0 : \mathrm{real}^3,\ L : \mathrm{real}^{2^3} \\
|\ [\ Q_i, Q_o :: \mathrm{flow}^3,\ V :: \mathrm{vol}^3 \\
\quad ,\ n : \mathrm{nat}^3,\ s : \mathrm{string} \\
\sim|\ V ::= V_0\,;\ n := \langle 0,0,0\rangle\,;\ s := \text{“idle”} \\
=|\ k \longleftarrow [0..3) :\ \ Q_i.k = n.k \cdot c.k \\
\quad ,\ k \longleftarrow [0..3) :\ \ Q_o.k = \rho.k \cdot (V.k)^\alpha \\
\quad ,\ k \longleftarrow [0..3) : (V.k)' = Q_i.k - Q_o.k \\
\quad |\ *[\ s \neq \text{“fill}\mathrm{V}_0\text{”}\,;\ \nabla V.0 < L.0.0 \longrightarrow n := \langle 1,0,0\rangle\,;\ s := \text{“fill}\mathrm{V}_0\text{”} \\
\qquad [\!] \ s \neq \text{“fill}\mathrm{V}_1\text{”}\,;\ \nabla V.1 < L.1.0 \longrightarrow n := \langle 0,1,0\rangle\,;\ s := \text{“fill}\mathrm{V}_1\text{”} \\
\qquad [\!] \ s \neq \text{“fill}\mathrm{V}_2\text{”}\,;\ \nabla V.2 < L.2.0 \longrightarrow n := \langle 0,0,1\rangle\,;\ s := \text{“fill}\mathrm{V}_2\text{”} \\
\qquad [\!] \quad s \neq \text{“idle”}\,;\ \nabla V.0 > L.0.1 \longrightarrow n.0 := 0\,;\ s := \text{“idle”} \\
\qquad [\!] \quad s \neq \text{“idle”}\,;\ \nabla V.1 > L.1.1 \longrightarrow n.1 := 0\,;\ s := \text{“idle”} \\
\qquad [\!] \quad s \neq \text{“idle”}\,;\ \nabla V.2 > L.2.1 \longrightarrow n.2 := 0\,;\ s := \text{“idle”} \\
\qquad] \\
] \ |
\end{array}$$

The system has been simulated with the parameters used in [Hoffmann and Engell, 1998], shown in Table 9.1.

i	ρ_i	c_i	L_{i0}	L_{i1}	$V_i(0)$
Tank0	0.2	0.45	0.3	0.7	0.7
Tank1	0.1	0.4	0.2	0.5	0.4
Tank2	0.5	0.8	0.1	0.9	0.1

Table 9.1: Parameters of c/d system.

$$\begin{array}{l}
\mathsf{xper}\ = |\ [\ s : .S\ |\ s(0.5, \langle 0.2, 0.1, 0.5\rangle, \langle 0.45, 0.4, 0.8\rangle, \langle 0.7, 0.4, 0.1\rangle \\
\qquad\qquad\qquad\qquad , \langle\langle 0.3, 0.7\rangle, \langle 0.2, 0.5\rangle, \langle 0.1, 09\rangle\rangle) \\
\qquad] \ |
\end{array}$$

The simulation results can be seen in figures 9.11 and 9.12. In Figure 9.11, the contents of the three tanks are plotted against time. Figure 9.12 shows the total amount of liquid present in the three tanks. In a second simulation, the initial contents of tank number one has been changed from 0.4 to 0.4001. In Figure 9.12, the dotted line depicts the total liquid in this case. As can be seen, after some time the behaviour of the system diverges significantly from that of the first simulation. The system exhibits no regular behaviour so that no longterm predictions can be made about the contents of the tanks. Also, significant divergence resulted from 0.025% modification in the initial state. Numerical studies [Hoffmann and Engell, 1998] point out that this system is chaotic for almost all parameter sets.

In this model, the situation when the contents in two tanks drop below the lower threshold at the same time is not addressed. Also, in live situations fairness must be ensured, that is, the server must eventually handle requests of all tanks.

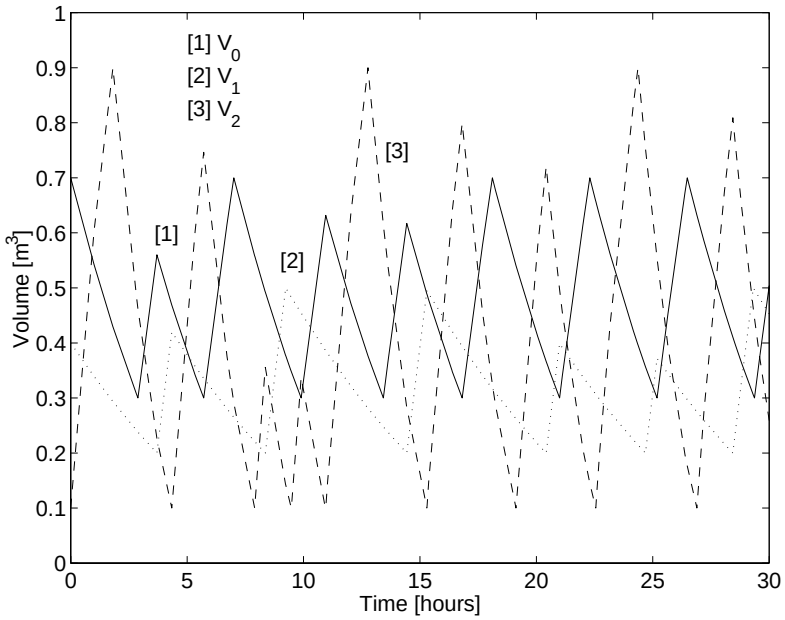

Figure 9.11: Contents of tanks.

9.5 Substitute equations

When building the current χ simulator, the aim was to be able to solve systems of DAEs with index 0 and 1. The DAE solver used in the χ simulator can solve index 1 DAEs, so this goal has been attained. High index problems can also be modelled in χ by means of substitute equations. In this section, index reduction is illustrated by using substitute equations. Also, equations with hidden constraints (that have not

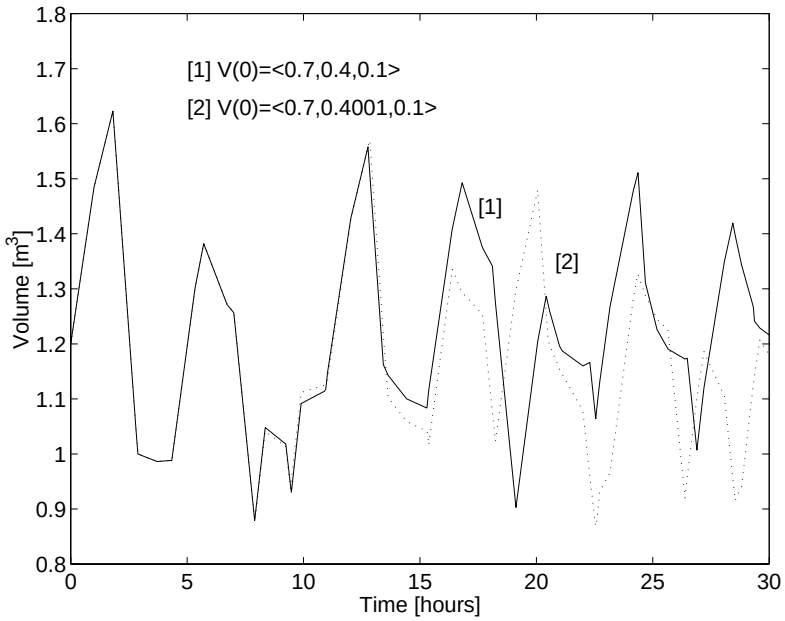

Figure 9.12: Total amount of liquid in the three tanks.

been treated in the mathematical model of the state calculations described in Chapter 4) can be modelled with the current χ simulator using substitute equations.

The index of a system of DAEs as defined in Section 2.2.1 is the number of times that all or part of the system must be differentiated, in order to reduce the system to a set of ordinary differential equations. High index systems, that is, systems with index greater than 2, arise in many application areas. However, only a few specific kinds of high index systems can be solved directly. Therefore, the usual technique is that through differentiation and algebraic manipulations, the index is lowered to 0 or 1, and the resulting system is solved with available ODE or DAE solvers. In the literature several algorithms can be found for index reduction. From these, the algorithm of Gear and Petzold [Gear and Petzold, 1984], the constraint stabilisation technique of Gear [Gear, 1988], and the Bachmann algorithm [Bachmann et al., 1990] all use symbolic differentiation and substitution. These algorithms can be applied in χ models using substitute equations. Furthermore, Mattson [Mattson and Söderlind, 1992] describes an index reduction technique, where derivatives are replaced by dummy algebraic variables. The application of this technique is also illustrated below.

In this section, a PID controller is used as an example for a high index system. A horizontal force F is applied to a body of mass m on a flat surface, without friction. The position of the mass is denoted by x. The control objective is to let the object follow a given trajectory x_{set}, where x_{set} is given as a function of time. The PID controller can be described in five equations in the five unknowns: x and v are the position and the speed of the body, respectively, u is the control signal, e is the error

and i is the integral of the error. Variables F and x_{set} are input variables, i.e., they are functions of time, m, k_P, k_D and k_I are constants.

$$\dot{x} = v \tag{9.2a}$$
$$\dot{v} = \frac{F-u}{m} \tag{9.2b}$$
$$\dot{i} = e \tag{9.2c}$$
$$e = x - x_{set} \tag{9.2d}$$
$$u = k_P e + k_D \dot{e} + k_I i \tag{9.2e}$$

The first three equations give expressions for $\dot{x}, \dot{v}$ and $\dot{i}$. Equation (9.2d) specifies a relationship between differential variables and an input variable, therefore e and x are dependent differential variables (see Section 2.2.1 for definition of dependent differential variables). This means that e and x cannot be initialized independently. Either the value of x is chosen freely, and the value of e is calculated from (9.2d), or e is chosen freely and x is calculated. Therefore, the number of degrees of freedom is reduced.

In order to reduce this system into an ODE, first (9.2d) is differentiated:

$$\dot{e} = \dot{x} - \dot{x}_{set} \tag{9.3}$$

which reveals a hidden constraint on $\dot{e}$. (Equation (9.3) can equally be considered as a hidden constraint on $\dot{x}$.) After substituting this into (9.2e), and using (9.2a) we get an algebraic equation for u:

$$u = k_P e + k_D(v - \dot{x}_{set}) + k_I i. \tag{9.4}$$

By differentiating this equation, we obtain

$$\dot{u} = k_P \dot{e} + k_D \dot{v} - k_D \ddot{x}_{set} + k_I \dot{i} = k_P(v - \dot{x}_{set}) + k_D(\frac{F-u}{m}) - k_D \ddot{x}_{set} + k_I e.$$

The index of (9.2) is therefore 2.

In order to be able to solve this system by numerical solvers, its index needs to be reduced to one. Furthermore, in order to be able to calculate the initial state of this system, the dependency between e and x, (and $\dot{e}$ and $\dot{x}$) must be revealed.

Please note that equation (9.2) is a semi-explicit index 2 system. This specific kind of index 2 systems can be solved directly be changing the error test of an existing DAE solver [Lötstedt and Petzold, 1986b]. However, such an altered DAE solver is not available in χ. This example has been chosen for its simplicity and the illustrated technique can be used to reduce the index of any high index system.

9.5.1 Prime-substitution

The new information used above for the index reduction is that $\dot{e} = \dot{x} - \dot{x}_{set}$. This is specified in χ by a substitute equation. Whether $\dot{e}$ or $\dot{x}$ is calculated by substitution can freely be chosen. By choosing $\dot{e}$, the above equation set is specified in χ as

$$
\begin{aligned}
& x' = v \\
, \; & v' = (F - u)/m \\
, \; & e = x - x_{set} \\
, \; & i' = e \\
, \; & u = k_P e + k_D e' + k_I i \\
, \; & e' \leftarrow x' - x'_{set}
\end{aligned}
$$

Variable e is a so-called prime substituted differential variable. The actual set of equations solved by solvers after substitution is

$$\dot{x} = v \tag{9.5a}$$
$$\dot{v} = \frac{F - u}{m} \tag{9.5b}$$
$$e = x - x_{set} \tag{9.5c}$$
$$\dot{i} = e \tag{9.5d}$$
$$u = k_P e + k_D(\dot{x} - \dot{x}_{set}) + k_I i. \tag{9.5e}$$

Note that for the solvers, $\dot{e}$ is not present in the equations. With other words, the same set of equations would actually be solved if a new algebraic variable were introduced, say a, and the following model were specified in χ

$$
\begin{aligned}
& x' = v \\
, \; & v' = (F - u)/m \\
, \; & e = x - x_{set} \\
, \; & i' = e \\
, \; & u = k_P e + k_D a + k_I i \\
, \; & a \leftarrow x' - x'_{set}
\end{aligned}
$$

It is more beneficial however to write a substitute equation for $\dot{e}$ than for a new algebraic variable a, because then in the χ model e is a differential variable (since its time derivative occurs in the equations). As a consequence, $\dot{e}$ can be used in the discrete-event part of the process, and wherever it is used it evaluates to $\dot{x}$. The algorithm of Mattson [Mattson and Söderlind, 1992] can be used to determine the set of equations that need to be differentiated and to determine the derivatives that have to be substituted. In this algorithm, parts of the DAEs are differentiated analytically, and appended to the original system. For each additional equation, a derivative is selected to be replaced by a new algebraic variable.

The system actually solved by solvers (equation (9.5)) is an index 1 system. Note, that in this equation set, (therefore for the solvers only!), variable e is an algebraic variable, so that there is no dependency among differential variables. Since x is a differential variable, its value can freely be chosen. After this, the value of e is calculated from the equations, so that e is correctly initialized to the value of x.

9.5.2 Base-prime substitution

The index can also be reduced by removing variable e from the equations. In this case, both e and $\dot{e}$ are calculated by substitution. In this case, the χ specification of the equations is as follows

$$
\begin{aligned}
& x' = v \\
, \; & v' = (F-u)/m \\
, \; & i' = e \\
, \; & u = k_P e + k_D e' + k_I i \\
, \; & e \leftarrow x - x_{set} \\
, \; & e' \leftarrow x' - x'_{set}
\end{aligned}
$$

Variable e is a differential base-prime substituted variable. Wherever e and $\dot{e}$ occur in the non-substituted equations, they are substituted by the right-hand-side expressions of the respective substitute equations. The equation set actually solved by numerical solvers after substitution is

$$
\begin{aligned}
\dot{x} &= v \\
\dot{v} &= \frac{F-u}{m} \\
\dot{i} &= x - x_{set} \\
u &= k_P e + k_D(\dot{x} - \dot{x}_{set}) + k_I i.
\end{aligned}
$$

Variable e has disappeared from the actual equations. What is left is four equations in four unknowns. This is an index 1 problem. Wherever the value of e or $\dot{e}$ is used in the discrete-event part of the process, it is calculated by substitution. Together with the index reduction, the problem of the hidden constraint on $\dot{e}$ is solved as well. In this system, the value of x can freely be chosen after which e is automatically initialized to x via substitution. Also, the value of $\dot{e}$ is set automatically to $\dot{x}$.

9.5.3 Consistent initialization

In the previous two approaches, the problems of high index DAEs and hidden constraint in the equations have been solved simultaneously. In this section, the situation is addressed where only the problem of a hidden constraint is present. This is the case when the system of DAEs is of index 1 and there are dependent differential variables. Again, the solution is substitution.

According to the mathematical model described in Chapter 4, the differential variables can freely be initialized in χ models, and the initial values of the algebraic variables are calculated from the equations. In most index 1 systems of DAEs, this approach is correct. There are, however, index 1 systems of DAEs where the system contains hidden constraints, so that not all differential variables can be initialized freely. To

illustrate this, take the PID example (9.2). Its index can also be reduced if variable u is replaced by a derivative of a dummy variable z. The set of equations now is

$$\dot{x} = v \tag{9.6a}$$
$$\dot{v} = \frac{F - \dot{z}}{m} \tag{9.6b}$$
$$e = x - x_{set} \tag{9.6c}$$
$$\dot{i} = e \tag{9.6d}$$
$$\dot{z} = k_P e + k_D \dot{e} + k_I i. \tag{9.6e}$$

This is an index 1 problem, because after differentiating equation (9.6c), the equations can be re-arranged into an ODE. Yet, e and x remain dependent differential variables so that they cannot be freely initialized, and as a consequence, there is a hidden constraint if equation (9.6c) is differentiated. The problem can easily be solved in χ as before, by using a substitution either for $\dot{e}$ only, or for both e and $\dot{e}$.

With this example, it is shown that index 1 systems that contain hidden constraints can easily be modelled in χ.

9.6 Final remarks

Several other hybrid χ models can be found in the literature. A χ model of a plant for the biochemical production of ethanol is described in [van Beek et al., 1995]. Detailed models of a conveyor line transportation system with actuators and sensors and the associated control system are presented in [van Beek et al., 1996] and [van Beek et al., 1997]. Finally, a model of a bottle factory and other examples are presented in [van Beek and Rooda, 1999].

Chapter 10

Conclusions

Combined discrete-event/continuous-time modelling is being increasingly applied in several engineering fields. Such modelling requires a hybrid specification formalism. Also, it is desirable to be able to execute hybrid models. In this thesis, the hybrid χ language and the design of the hybrid χ simulator have been presented. In this chapter, this work is evaluated and suggestions for future work are summarized.

10.1 The χ language

The χ language is suitable to specify purely discrete, purely continuous and mixed discrete-event/continuous-time systems. It has a small number of language constructs, which makes the language simple and easy to use. It uses LaTeX symbols for specifications which makes specifications compact and, after getting acquainted with the symbols, easy to read. The χ language has been designed from the beginning as a hybrid language: where possible, the discrete and the continuous language elements are based on similar concepts, and follow a similar language philosophy.

The discrete-event part of the language is based on CSP. In CSP, parallelism, communication between discrete sub-systems, modularity and time-events can be expressed. In addition, in χ, random distribution functions can be used. This makes the χ language suitable to describe hybrid systems, where purely discrete sub-systems are present. Such systems are, for example, production systems and embedded systems.

In χ, the continuous-time behaviour is specified symbolically by means of DAEs. Connection of continuous variables define a non-causal equality relation. Currently, first order DAEs of index 0 and 1 can be solved directly. The state of the system can be (re-)initialized using default continuity assumptions or it can be (re-)initialized to its steady-state using an alternative continuity assumption. The continuous and hybrid capabilities of the χ language have been demonstrated in this thesis by means of examples of mechanical systems and continuous production systems.

In this thesis, the hybrid aspects of the language semantics have been defined. The most important features of the semantics are:

- Differential and algebraic variables are distinguished: only differential variables can be initialized freely. This categorization emphasizes the physical role of the variables and thereby helps to develop physically correct models. The distinction allows specification of steady-state initialization as well.
- Connections of continuous variables define an algebraic equation among the variables. These equations are semantically equivalent to all other equations defined in the processes. This choice simplifies the language semantics.
- Discrete phases are divided into sub-phases, each ending with a consistent initial state calculation. Within each sub-phase, processes work with local variables only, and interaction between sub-models is restricted to explicit communication. This facilitates reasoning about model behaviour.
- Consistent initial state calculations take place when all processes are blocked. Therefore, no additional language constructs are needed to indicate when initial state calculation is required; this keeps the language small. Furthermore, continuous variables that are being changed - either by means of assignment statements, or as a result of the initial state calculation - cannot be used in other processes because: *a)* variables are local to processes and *b)* when the initial state calculation actually changes the variables, all processes are blocked.
- State-event specifications always block the execution of the processes, and are evaluated after a new consistent initial state has been established. This semantics is important in situations when two processes access connected variables simultaneously, one for writing, and the other one to evaluate a state-event specification. In this case, the writing access succeeds first always. This ensures that in state-event specifications, always a new and consistent state is considered.
- In the χ language, a new language construct is introduced: the substitute equation. This construct can be used to reduce the index of DAEs and to reveal hidden constraints in the equations in an elegant way. Furthermore, it can also be used to handle discontinuities efficiently.

In this thesis, several areas have been identified for future research. The most important task for the future is to develop formal methods and techniques for the formal analysis of hybrid χ models. The contribution of this thesis to this goal is that the hybrid semantics is identified in an operational form.

The continuous capabilities of the language can further be enhanced by the following language extensions.

- dynamic allocation and de-allocation of continuous variables and equations
  This allows one to model some dynamic systems where individual entities enter and leave the operation area in an elegant way. Examples are production lines and systems that have autonomous entities.

- instantaneous equations
  Such equations can be used to specify instantaneous changes in physical systems, where the new state of the system after the change is (partially) determined by physical conservation principles. Examples are collisions in mechanical systems.

- high order time derivatives
  Many physical systems can be most naturally described by high order time derivatives. Currently, when a high order derivative is necessary, a dummy variable is introduced (e.g. $\ddot{x} = a$ is specified as $\dot{x} = v$, $\dot{v} = a$).

- high index DAEs
  The mathematical models of many physical systems are high index DAEs. In the literature, several algorithms can be found for index reduction; see Section 9.5. These algorithms can be used for automatic index reduction. However, allowing high index DAEs is a radical step, because, in high index system not all differential variables can be initialized freely, as it is currently assumed. Therefore, this approach requires the language semantics to be reconsidered.

- type system extension with dimensionality
  Regarding the types of the continuous variables, currently, a limited type system is defined and implemented in the χ simulator. A new concept is presented in Appendix A. This concept supports the development of physically correct models by specifying operations on variables representing physical quantities.

10.2 The hybrid χ simulator

In this thesis, the design of the hybrid χ simulator has been described. This simulator is an extension of an existing discrete-event simulator. It is written in C++ , using object-oriented technology, which has proved to be beneficial, allowing reuse of code and fast and smooth extension.

The simulator integrates two numerical solvers: NLEQ is used to calculate a consistent initial state after discontinuities and DASSL integrates the DAEs in continuous phases. When discrete phases occur closer to each other than the minimum step-size of DASSL, this is detected by the simulator and the two phases are scheduled after each other, at the same time point.

In the χ simulator, the continuous variables are mapped onto a smaller set of global variables that are used internally by the numerical solvers. Furthermore, state-event specifications are mapped onto switching functions of the DAE solver.

Currently, the χ simulator does not implement complex continuous data types, such as arrays of continuous variables or arrays of continuous channels. In the future, these extensions should be made. Furthermore, the performance of the simulator can be improved, and the range of the problems it can handle can be extended by the following techniques.

- symbolic analysis and manipulation of equations
  Symbolic analysis and manipulation of the equations can be used for several tasks. The most important ones are:
  - to identify high index problems and reduce the index
  - to reduce the size of the problem that needs to be solved by the numerical solvers
  - to identify the type of the equations and choose a suitable numerical algorithm to solve them
  - to derive additional information about the equations (e.g., Jacobian matrix) analytically
- use of more DAE and ODE solvers
  Currently, the χ simulator only uses DASSL to solve the DAEs. The current solver interface can easily be used to integrate other DAE and ODE solvers as well. By having several solvers available, users or the simulator itself may choose the solver most suitable to the problem. New solvers may also be selected to enlarge the problem domain and to enable the use of new technology when they become available.

Bibliography

[Abadi and Lamport, 1992] Abadi, M. and Lamport, L. (1992). An old-fashioned recipe for real time. In de Bakker, J., Huizing, K., de Roever, W.-P., and Rozenberg, G., editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag.

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Massachusetts.

[Alur et al., 1993] Alur, R., Courcoubetis, C., and Dill, D. L. (1993). Model checking in dense real time. *Information and Computation*, 104(1):2–34.

[Alur et al., 1995] Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Ho, P., Olivero, X. N. A., Sifakis, J., and Yovine, S. (1995). The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34.

[Alur and Henzinger, 1992] Alur, R. and Henzinger, T. A. (1992). Logics and models of real time: a survey. In de Bakker, J., Huizing, K., de Roever, W.-P., and Rozenberg, G., editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer-Verlag.

[Alur and Henzinger, 1993] Alur, R. and Henzinger, T. A. (1993). Real-time logics: complexity and expressiveness. *Information and Computation*, 104(1):35–77.

[Alur and Henzinger, 1994] Alur, R. and Henzinger, T. A. (1994). A really temporal logic. *Journal of the ACM*, 41(1):181–204.

[Alur et al., 1996] Alur, R., Henzinger, T. A., and Ho, P.-H. (1996). Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201.

[Arends, 1996] Arends, N. W. A. (1996). *A Systems Engineering Specification Formalism*. PhD thesis, Eindhoven University of Technology, The Netherlands.

[ASCEND, 1997] ASCEND (1997). *ASCEND IV Manual*. Carnegie Mellon University.

[Augustin et al., 1967] Augustin, D. C., Fineberg, M. S., Johnson, B. B., and Linebarger, R. N. (1967). The SCi continuous system simulation language CSSL. *Simulation*, 9:281–303.

[Bachmann et al., 1990] Bachmann, R., Brüll, L., Mrziglod, T., and Pallaske, U. (1990). On methods for reducing the index of differential algebraic equations. *Computers and Chemical Engineering*, 14:1271–1273.

[Bail et al., 1992] Bail, J. L., Alla, J. H., and David, R. (1992). Hybrid petri nets. In *1st European Control Conference*, pages 1472–1477, Grenoble.

[Barton, 1992] Barton, P. I. (1992). *The Modelling and Simulation of Combined Discrete/Continuous Processes*. PhD thesis, University of London.

[Barton and Pantelides, 1994] Barton, P. I. and Pantelides, C. C. (1994). Modeling of combined discrete/continuous processes. *AIChE*, 40(6):966–979.

[Bos and Kleijn, 1999] Bos, V. and Kleijn, J. J. T. (1999). Structured operational semantics of Chi. Technical report, Eindhoven University of Technology, Eindhoven.

[Branicky, 1995] Branicky, M. S. (1995). *Studies in Hybrid Systems: Modeling, Analysis, and Control*. PhD thesis, Massachuesetts Institute of Technology, Cambridge, MA.

[Brenan, 1983] Brenan, K. E. (1983). *Stability and Convergence of Difference Approximations for Higher index Differential-Algebraic Systems with Applications in Trajectory Control*. PhD thesis, University of California at Los Angeles.

[Brenan et al., 1996] Brenan, K. E., Campbell, S. L., and Petzold, L. R. (1996). *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM's Classics in Applied Mathematics. Siam, Philadelphia.

[Brenan and Petzold, 1988] Brenan, K. E. and Petzold, L. R. (1988). The numerical solution of higher index differential/algebraic equations. *Mathematics of Computation*, 51:659–676.

[Cardelli, 1996] Cardelli, L. (1996). Type systems. In Tucker, A. B., editor, *Handbook of Computer Science and Engineering*. CRC Press.

[Cellier, 1986] Cellier, F. E. (1986). Combined continuous/discrete simulation applications, techniques, and tools. In Wilson, J., Henriksen, J., and Roberts, S., editors, *Proceedings of the 1986 Winter Simulation conference*, pages 24–33.

[Cellier and Elmqvist, 1993] Cellier, F. E. and Elmqvist, H. (1993). Automated formula manipulation supports object-oriented continuous system modeling. *IEEE Control System Magazine*, pages 28–38.

[Cellier et al., 1993] Cellier, F. E., Elmqvist, H., Otter, M., and Taylor, J. H. (1993). Guidelines for modeling and simulation of hybrid systems. In *IFAC 12th Triennial World Congress*, Sydney, Australia.

[Chaochen et al., 1996] Chaochen, Z., Ji, W., and Ravn, A. P. (1996). A formal description of hybrid systems. In Alur, R., Henzinger, T. A., and Sontag, E. D., editors, *Hybrid Systems III - Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*, pages 511–530. Springer-Verlag.

[Cosmo, 1995] Cosmo, R. D. (1995). *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Birkhäuser, Boston.

[Dahlquist and Björk, 1974] Dahlquist, G. and Björk, A. (1974). *Numerical Methods*. Prentice Hall, Englewood Cliffs, New Jersey. Translated by N. Anderson.

[David and Alla, 1990] David, R. and Alla, H. (1990). Autonomous and timed continuous petri nets. In *11th Int. Conf. on Application and Theory of Petri Nets*, pages 367–386, Paris.

[Deshpande et al., 1997] Deshpande, A., Göllü, A., and Varaiya, P. (1997). SHIFT: A formalism and a programming language for dynamic networks of hybrid automata. In *Hybrid systems V.*, LNCS. Springer-Verlag.

[Deuflhard, 1974] Deuflhard, P. (1974). A modified newton method for the solution of ill-conditioned systems of nonlinear equations with application to multiple shooting. *Numer. Math*, 22:289–315.

[Duff et al., 1986] Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford.

[Edgar and Himmelblau, 1989] Edgar, T. F. and Himmelblau, D. M. (1989). *Optimization of Chemical Processes*. McGraw-Hill.

[Elmqvist et al., 1996] Elmqvist, H., Brück, D., and Otter, M. (1996). *Dymola - User's manual*. Dynasim AB, Lund, Sweden.

[Elmqvist et al., 1993] Elmqvist, H., Cellier, F., and Otter, M. (1993). Object-oriented modeling of hybrid systems. In *Proceedings of European Simulation Symposium, ESS'93*. The Society of Computer Simulation.

[Elmqvist and Mattson, 1997] Elmqvist, H. and Mattson, S. E. (1997). MODELICA - the next generation modeling language - an international design effort. In *Proceedings of the 1st World Congress on System Simulation , WCSS'97*, Singapore.

[Fábián and Janson, 1996] Fábián, G. and Janson, P. (1996). Simulator for combined continuous-time/discrete-event models. Technical report, Stan Ackermans Instituut, Eindhoven University of Technology, The Netherlands.

[Fahrland, 1970] Fahrland, D. A. (1970). Combined discrete event continuous systems simulation. *Simulation*, 14(61).

[Filippov, 1964] Filippov, A. F. (1964). Differential equations with discontinuous right-hand sides. *ACM Transl.*, 42:199–231.

[Gear, 1971] Gear, C. W. (1971). The simultaneous numerical solution of differential algebraic equations. *IEEE Trans. Circuit Theory*, CT-18:89–95.

[Gear, 1988] Gear, C. W. (1988). Differential-algebraic equation index transformations. *SIAM. J. Sci. Stat. Comp.*, 9:39–47.

[Gear et al., 1981] Gear, C. W., Hsu, H. H., and Petzold, L. R. (1981). Differential-algebraic equations revisited. In *Proc. ODE Meeting*, Oberwolfach.

[Gear et al., 1985] Gear, C. W., Leimkuhler, B., and Gupta, G. K. (1985). Automatic integration of Euler-Lagrange equations with constraints. *J. Comput. Appl. Math.*, 12(13):77–90.

[Gear and Petzold, 1984] Gear, C. W. and Petzold, L. R. (1984). ODE methods for the solution of differential/algebraic systems. *SIAM Journal on Numerical Analysis*, 21:716–728.

[Griepentrog and März, 1986] Griepentrog, E. and März, R. (1986). *Differential Algebraic Equations and Their Numerical Treatment*, volume 88. Teubner-Texte zur Math., Leipzig.

[Hairer et al., 1980] Hairer, E., Lubich, C., and Roche, M. (1980). *The Numerical Solution of Differential-Algebraic Systems by Runge-Kutta Methods*, volume 1409 of *Lecture Notes in Mathematics*. Springer-Verlag.

[Hairer and Wanner, 1991] Hairer, E. and Wanner, G. (1991). *Solving Ordinary Differential Equations II. Stiff and Differential Algebraic Problems*. Springer-Verlag.

[Harel et al., 1990] Harel, E., Lichtenstein, O., and Pnueli, A. (1990). Explicit-clock temporal logic. In *Proceedings of the Fith Annual Symposium on Logic in Computer Science*, pages 402–413. IEEE Computer Society Press.

[Henzinger et al., 1995] Henzinger, T. A., Kopke, P., Puri, A., and Varaiya, P. (1995). What's decidable about hybrid automata. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 373–382.

[Henzinger et al., 1992] Henzinger, T. A., Manna, Z., and Pnueli, A. (1992). What good are digital clocks. In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming (ICALP 1992)*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer-Verlag.

[HindMarsh, 1980] HindMarsh, A. C. (1980). LSODE and LSODI, two new initial value ordinary differential equations solvers. *ACM SIGNUM Newsletter*, 15:10–11.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall, Englewood-Cliffs.

[Hoffmann and Engell, 1998] Hoffmann, I. and Engell, S. (1998). Chaos in simple logistic systems. In *Proceedings of the 9th Symposium on Information Control in Manufacturing*, Nancy-Metz, France.

[Holmes, 1995] Holmes, J. (1995). *Object-oriented compiler construction*. Prentice Hall Inc., New Jersey.

[Hooman, 1991] Hooman, J. (1991). *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Iwasaki et al., 1995] Iwasaki, Y., Farquhar, A., Saraswat, V., Bobrow, D., and Gupta, V. (1995). Modeling time in hybrid systems: How fast is 'Instantaneous'? In *Proceedings of the 1995 International Conference on Qualitative Reasoning*, pages 94–103, Amsterdam.

[Jahanian and Mok, 1986] Jahanian, F. and Mok, A. K. (1986). Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904.

[Jarvis, 1993] Jarvis, R. B. (1993). *Robust Dynamic Simulation of Chemical Engineering Processes*. PhD thesis, University of London.

[Jarvis and Pantelides, 1992] Jarvis, R. B. and Pantelides, C. C. (1992). DASOLV– a differential algebraic equation solver. Technical report, Centre for Process Systems Engineering, Imperial College, London.

[Jifeng, 1994] Jifeng, H. (1994). From CSP to hybrid systems. In Roscoe, A., editor, *A Classical Mind, Essays in Honour of C.A.R. Hoare*, Prentice Hall International Series in Computer Science, pages 171–189. Prentice Hall.

[Kaldewaij, 1990] Kaldewaij, A. (1990). *Programming: The Derivation of Algorithms*. International Series in Computer Science. Prentice Hall.

[Karnopp et al., 1990] Karnopp, D. C., Margolis, D. L., and Rosenberg, R. C. (1990). *System Dynamics: A Unified Approach*. John Wiley & Sons.

[Kleijn et al., 1998] Kleijn, J. J. T., Reniers, M. A., and Rooda, J. E. (1998). A process algebra based verification of a production system. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 90–99, Brisbane.

[Koenig et al., 1967] Koenig, H. E., Tokad, Y., and Kesavan, H. (1967). *Analysis of Discrete Systems*. McGraw-Hill.

[Koymans et al., 1987] Koymans, R., Kuiper, R., and Zijlstra, E. (1987). Specifying message passing and real-time systems with real-time temporal logic. In *Exprit 87 Results and Achievements*, volume 2(4). North Holland.

[Lafferriere et al., 1999] Lafferriere, G., Pappas, G. J., and Yovine, S. (1999). A new class of decidable hybrid systems. In Vaandrager, F. W. and van Schuppen, J. H., editors, *Hybrid Systems: Computation and Control, Second International Workshop, HSCC'99*, volume 1569 of *Lecture Notes in Computer Science*, pages 137–1151. Springer-Verlag.

[Leimkuhler et al., 1991] Leimkuhler, B., Petzold, L. R., and Gear, C. W. (1991). Approximation methods for the consistent initialization of differential-algebraic equations. *SIAM Journal on Numerical Analysis*, 28:205–226.

[Lewis, 1990] Lewis, H. R. (1990). A logic of concrete time intervals. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 380–389. IEEE Computer Society Press.

[Liniger, 1979] Liniger, W. (1979). Multistep and one-leg methods for implicit mixed differential algebraic systems. *IEEE Trans. Circuits Systems*, pages 755–762.

[Lötstedt and Petzold, 1986a] Lötstedt, P. and Petzold, L. R. (1986a). Numerical solution of nonlinear differential equations with algebraic constraints I: Convergence results for backward differentiation formulas. *Mathematics of Computation*, 46:491–516.

[Lötstedt and Petzold, 1986b] Lötstedt, P. and Petzold, L. R. (1986b). Numerical solution of nonlinear differential equations with algebraic constraints II: practical applications. *Mathematics of Computation*, 46:491–516.

[Lynch et al., 1996] Lynch, N. A., Segala, R., Vaandrager, F. W., and Weinberg, H. B. (1996). Hybrid I/O automata. In Alur, R., Henzinger, T. A., and Sontag, E. D., editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer-Verlag.

[Maler et al., 1992] Maler, O., Manna, Z., and Pnueli, A. (1992). From timed to hybrid systems. In de Bakker, J. W., Huizing, C., de Roever, W. P., and Rozenberg, G., editors, *Proceedings of the REX Workshop Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 447–484. Springer-Verlag.

[Manna and Pnueli, 1993] Manna, Z. and Pnueli, A. (1993). Verifying hybrid systems. In Grossman, R., Nerode, A., Ravn, A., and Rischel, H., editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 4–35. Springer-Verlag.

[Marquardt, 1991] Marquardt, W. (1991). Dynamic process simulation – recent progress and future challenges. *Chemical Process Control*, pages 131–180.

[Mattson, 1988] Mattson, S. E. (1988). On model structuring concepts. In *Preprints of the 4th IFAC Symposium on Computer-Aided Design in Control Systems*, pages 209–214, Beijing.

[Mattson, 1989] Mattson, S. E. (1989). Modelling of interactions between submodels. In Iazeolla, G., Lehmann, A., and van den Herik, J. M., editors, *Simulation methodologies, languages and architectures and AI and graphics for simulation*, pages 63–68, Rome. SCS.

[Mattson et al., 1993] Mattson, S. E., Andersson, M., and Åström, K. J. (1993). Object-oriented modelling and simulation. In Linkens, editor, *CAD for Control Systems*, chapter 2, pages 31–69. Marcel Dekker Inc, New York.

[Mattson and Söderlind, 1992] Mattson, S. E. and Söderlind, G. (1992). A new technique for solving high-index differential-algebraic equations using dummy derivatives. In *1992 IEEE Symposium on Computer-Aided Control System Design, CACSD'92*, Napa, California.

[Mitchell and Gauthier, 1976] Mitchell, E. E. L. and Gauthier, J. S. (1976). Advanced continuous simulation language (ACSL). *Simulation*, 26(3):72–78.

[Modelica, 1997] Modelica (1997). Modelica – a unified object-oriented language for physical systems modeling. Available at `http://www.Dynasim.se/Modelica`.

[Moller and Tofts, 1990] Moller, F. and Tofts, C. (1990). A temporal calculus of communicating systems. In Baeten, J. and Klop, J., editors, *Proceedings of Concur'90*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag.

[Mosterman, 1999] Mosterman, P. J. (1999). An overview of hybrid simulation phenomena and their support by simulation packages. In Vaandrager, F. W. and van Schuppen, J. H., editors, *Hybrid Systems: Computation and Control, Second International Workshop, HSCC'99*, volume 1569 of *Lecture Notes in Computer Science*, pages 165–177. Springer-Verlag.

[Mosterman and Biswas, 1996] Mosterman, P. J. and Biswas, G. (1996). A theory of discontinuities in physical system models. *Journal of the Franklin Institute*, 40(6):966–979.

[Mosterman et al., 1998] Mosterman, P. J., Zhao, F., and Biswas, G. (1998). Sliding mode model semantics and simulation for hybrid systems. In Henzinger, T. A. and Sastry, S., editors, *Hybrid Systems V.*, volume 1386 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Naumoski and Alberts, 1998] Naumoski, G. and Alberts, W. (1998). *A Discrete-Event Simulator for Systems Engineering*. PhD thesis, Eindhoven University of Technology. The Netherlands.

[Nicollin et al., 1990] Nicollin, X., Richier, J. L., Sifakis, J., and Voiron, J. (1990). ATP: an algebra for timed processes. In *Proceedings of the IFIP TC 2 Working Conference of Programming Concepts and Methods*, Sea of Galilee, Israel.

[Nicollin et al., 1993] Nicollin, X., Sifakis, J., and Yovine, S. (1993). From ATP to timed graphs and hybrid systems. *Acta Informatica*, pages 181–202.

[Novak and Weiman, 1991] Novak, U. and Weiman, L. (1991). A family of newton codes for systems of highly nonlinear equations. Technical report, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

[Okuda and Ushio, 1990] Okuda, K. and Ushio, T. (1990). Petri net based qualitative simulation. In *Proc. IAESTED Int. Symp. on Expert Systems Theory and Applications*, Los Angeles, California.

[Ostroff, 1989] Ostroff, J. S. (1989). Temporal logic for real-time systems. In *Advanced Software Development Series*. Research Studies Press (John Wiley & Sons), Taunton England.

[Pantelides, 1988] Pantelides, C. C. (1988). The consistent initialization of differential-algebraic systems. *SIAM. J. Sci. Stat. Comput.*, 9(2):213–231.

[Pantelides and Barton, 1993] Pantelides, C. C. and Barton, P. I. (1993). Equation-oriented dynamic simulation current status and future perspectives. *Computers Chem. Eng.*, 17:s263–s285.

[Park and Barton, 1996] Park, T. and Barton, P. I. (1996). State event location in differential-algebraic models. *ACM Transactions on Modeling and Computer Simulation*, 6(2):137–165.

[Parsons, 1992] Parsons, T. W. (1992). *Introduction to compiler construction.* Computer Science Press, New York.

[Perkins, 1986] Perkins, J. D. (1986). Survey of existing systems for the dynamic simulation of industrial processes. *Modeling, Identification and Control*, 7(71).

[Perkins and Sargent, 1982] Perkins, J. D. and Sargent, R. W. H. (1982). SPEEDUP: a computer program for steady-state and dynamic simulation and design of chemical processes. In *AIChE Symposium Series*, volume 78.

[Petzold, 1982] Petzold, L. R. (1982). Differential/algebraic equations are not ODEs. *SIAM J. Sci. Statist. Comput.*, 3:367–384.

[Petzold, 1983] Petzold, L. R. (1983). A description of DASSL: A differential/algebraic system solver. *Scientific Computing*, pages 65–68.

[Petzold, 1986] Petzold, L. R. (1986). Order results for implicit Runge-Kutta methods applied to differential algebraic systems. *SIAM. J. Numer. Anal.*, 23:837–852.

[Piela et al., 1991] Piela, P., Epperly, T., Westerberg, K., and Westerberg, A. (1991). ASCEND: An object-oriented computer environment for modeling and analysis: the modeling language. *Computers and Chemical Engineering*, 15(1):53–72.

[Pnueli and Harel, 1988] Pnueli, A. and Harel, E. (1988). Applications of temporal logic to the specification of real-time systems. In Joseph, M., editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 331 of *Lecture Notes in Computer Science*, pages 84–98. Springer-Verlag.

[Reed and Roscoe, 1986] Reed, G. M. and Roscoe, A. W. (1986). A timed model for communicating sequential processes. In *Proceedings of the 13th International Colloquium on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer-Verlag.

[Reed and Roscoe, 1987] Reed, G. M. and Roscoe, V. W. (1987). Metric spaces as models for real-time concurrency. In *Mathematical Foundations of Programming*, volume 298 of *Lec. Notes in Computer Science*, pages 331–343. Springer-Verlag.

[Rulkens et al., 1998] Rulkens, H. J. A., van Campen, E. J. J., van Herk, J., and Rooda, J. E. (1998). Batch size optimization of a furnace and pre-clean area by using dynamic simulations. In *SEMI/IEEE Advanced Semiconductor Manufacturing Conference ASMC*, Boston.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall Inc., New Jersey.

[Shampine and Gordon, 1975] Shampine, L. and Gordon, M. K. (1975). *Computer Solution of Ordinary Differential Equations*. W.H. Freeman and Co., San Francisco, CA.

[Simulink, 1993] Simulink (1993). *Simulink User's Guide*. The MathWorks.

[Sincovec et al., 1981] Sincovec, R. F., Erisman, A. M., Yip, E. L., and Epton, M. A. (1981). Analysis of descriptor systems using numerical algorithms. *IEEE Trans. Automat. Control*, pages 139–147.

[Speckhart and Green, 1976] Speckhart, H. and Green, W. H. (1976). *A Guide to Using CSMP*. Prentice Hall.

[Stewart, 1990] Stewart, D. (1990). A high accuracy method for solving ODEs with discontinuous right-hand side. *Numer. Math.*, 58:299–328.

[SystemBuild, 1984] SystemBuild (1984).
At `http://www.isi.com/Products/MATRIXx/SystemBuild`.

[van Beek and Rooda, 1998a] van Beek, D. and Rooda, J. (1998a). Languages and applications in hybrid modelling: Positioning of Chi. In *Proc. 9th Symposium on Information Control in Manufacturing*, pages 77–82, Nancy.

[van Beek et al., 1995] van Beek, D. A., Gordijn, S. H. F., and Rooda, J. E. (1995). Integrating continuous-time and discrete-event concepts in process modelling, simulation and control. In *Proceedings of the First World Conference on Integrated Design and Process Technology*, pages 197–204.

[van Beek et al., 1997] van Beek, D. A., Gordijn, S. H. F., and Rooda, J. E. (1997). Integrating continuous-time and discrete-event concepts in modelling and simulation of manufacturing machines. *Simulation Practice and Theory*, 5:653–669.

[van Beek and Rooda, 1997] van Beek, D. A. and Rooda, J. E. (1997). Design of discrete controllers for continuous systems using hybrid Chi. In *Proceedings of IFAC 7th Symposium on Computer Aided Control Systems Design (CACSD'97)*, pages 9–14, Gent.

[van Beek and Rooda, 1998b] van Beek, D. A. and Rooda, J. E. (1998b). Specification of discontinuities in hybrid models. In *Les Syst mes Dynamiques Hybrides / Hybrid Dynamical Systems—Proc. of 3rd International Conference on Automation of Mixed Processes*, pages 415–420, Reims.

[van Beek and Rooda, 1999] van Beek, D. A. and Rooda, J. E. (1999). Production and process systems modelling. Lecture notes, Eindhoven University of Technology, Department of Mechanical Engineering, The Netherlands.

[van Beek et al., 1996] van Beek, D. A., Rooda, J. E., and Gordijn, S. H. F. (1996). Hybrid modelling in discrete-event control system design. In *CESA'96 IMACS Multiconference, Symposium on Discrete Events and Manufacturing Systems*, pages 596–601, Lille.

[van de Mortel-Fronczak, 1995] van de Mortel-Fronczak, J. M. (1995). Operational semantics of Chi. Technical Report WPA 420062, Department of Mechanical Engineering, Eindhoven University of Technology.

[van de Mortel-Fronczak and Rooda, 1996] van de Mortel-Fronczak, J. M. and Rooda, J. E. (1996). On the integral modelling of control and production management systems. In *Proceedings of APMS'96*, pages 171–176, Kyoto, Japan.

[van de Mortel-Fronczak and Rooda, 1997] van de Mortel-Fronczak, J. M. and Rooda, J. E. (1997). A case study in the design of control systems for flexible production cells. In *Proceedings of MIM'97*, pages 243–248, Vienna, Austria.

[van de Mortel-Fronczak et al., 1995] van de Mortel-Fronczak, J. M., Rooda, J. E., and van den Nieuwelaar, N. J. M. (1995). Specification of a flexible manufacturing system using concurrent programming. *The International Journal of Concurrent Engineering: Research & Applications*, pages 187–194.

[Wijckmans, 1996] Wijckmans, P. M. E. J. (1996). *Conditioning of Differential Algebraic Equations and Numerical Solution of Multibody Dynamics*. PhD thesis, Eindhoven University of Technology, The Netherlands.

[Wood et al., 1984] Wood, R. K., Thambynayagam, R. K. M., Noble, R. G., and Sebastian, D. J. (1984). DPS – a digital simulation language for the process industries. *Simulation*, pages 221–233.

Appendix A

On dimensions and units

Models written in a hybrid specification language must not only be correct in a sense of correctness in computer programs, but also they have to be *physically consistent*. Therefore, in hybrid languages it is desirable to be able to express physical properties of quantities (dimensionality, for example), and to be able to verify physical consistency. This is an important step towards supporting physically correct models.

For this purpose, variables that represent *physical quantities* should be treated different than pure numbers. It is not sufficient to express the absolute value of such variables, but the unit of measurement is also required to be able to interpret this value. For example, a variable v that represents length may have value 5, that can be 5 kilometers or 5 meters, depending on its unit of measurement. In this chapter, physical quantities are treated as pairs, consisting of a value and a unit of measurement.

Physical quantities belong to quantity types, called *dimensions*. These are, for example, length, mass, time, force, electrical charge, etc. Each dimension may have more *units of measurement*, (or shortly units), so that the same quantity can be expressed in different units. For example, 5 kilometers may be expressed as 5000 meters. It is therefore, more appropriate to define compatibility of variables based on their dimensions, than based on their units of measurement.

Physical consistency can be formulated in terms of dimensions. To be able to carry out consistency checks, all variables used in continuous context must have a dimension. This does not only include continuous variables, but also discrete variables used for example as constants in equations. The following language constructs need to be dimensionally correct.

- *equations*
  The dimensions of the two sides of an equation must be the same.
- *connections*
  Only variables of the same dimension can be connected.
- *functions*
  Functions used in continuous context (in equations for example), have dimension

information. They must be declared and used correctly with respect to dimensions.

- *discrete-event statements*
  All statements and expressions that involve variables with dimension must be dimensionally correct.

Dimensions and units are used in several modelling languages (Omola [Mattson et al., 1993], Modelica [Modelica, 1997], ASCEND [Piela et al., 1991], gPROMS [Barton, 1992]). They are attributes of the types of continuous variables. These attributes are often optional, or default values can be used in case a user do not want to fully specify them. To a varying extent, the dimension concept has been introduced and, except for Modelica and gPROMS, physical consistency checks are carried out in these languages. However, physical consistency is often not well defined in the case when physical quantities are used in discrete-event statements. This, as will be shown, allows inconsistent models.

The challenge in hybrid languages is the specification of physical consistency, particularly in discrete-event statements, and the smooth integration of these requirements into the language semantics. The means to enforce compatibility in a computer language is the type system. Therefore, the type system of a hybrid language must express the compatibility of physical quantities and it must specify their correct use. Particularly, the use of physical quantities together with numerical and other kinds of quantities must be well defined.

A.1 Related work

Definitely, one of the biggest obstacles to using units of measurement in simulation packages is that currently several different systems of units are in use. Many of these are even national systems that are difficult to unify. A simulation package that is intended to be used internationally, should therefore not commit to one particular system of units but should allow (for example by means of libraries) to specify a particular set of units to be used. There are international efforts to make the current situation more clear. There are two international standards, ISO 31-1992 "General principles concerning quantities, units and symbols" and ISO 1000-1992 "SI units and recommendations for the use of their multiples and of certain other units". These two standards give guidelines for naming and specifying units and quantities. Furthermore, there exist programs that can answer compatibility problems, and can calculate the base SI representation of quantities. For example, the GNU units program converts quantities expressed in various scales to their equivalents in other scales.

In [Mattson, 1989] Mattson proposed to check the compatibility of connections in modelling languages, and to introduce a proper scale factor in the connection equations. However, not many simulation packages implement this concept.

Among the hybrid simulation languages, ASCEND [ASCEND, 1997] implements maybe the most sophisticated dimension concept. There are 10 base dimensions, from which new dimensions can be derived. The displaying unit of measurement of each dimension can be chosen by users. Alternatively, in case of derived dimensions, the displaying units can be derived from the units of the base dimensions. Libraries of different sets of units may be used as well. Dimensionality is checked in equations and statements. However, new dimensions cannot be introduced. Also, the dimension may be left undefined for some types, and the actual dimension is later deduced from the use of the variables of the particular type. The author considers this an error prone approach.

A.2 Dimension

Dimensions are the types of physical quantities. Examples are force, mass, flow, etc. By using dimension operators, the relation among different kinds of physical quantities can be expressed at an abstract level. For example, pressure = force/area.

According to the SI standard, the base dimensions are length, mass, time, electrical current, temperature, quantity, and luminous intensity. The advantage of using the SI system is that it is metric. That means that other dimensions can be derived from the base dimensions using three operators: multiplication, division and exponentiation. In SI, each base dimension has a unique unit of measurement, listed in Table A.1. Taking into account the metric character, the derived dimensions have unique units

Dimension	Unit	Unit name
length	m	meter
mass	kg	kilogram
time	s	second
electrical current	A	ampere
temperature	K	kelvin
quantity	mol	mole
luminuous intensity	cd	candela

Table A.1: The base SI units.

of measurement too. To see this, take an arbitrary unit expression. By counting the exponents of each base unit, the unit expression can be brought into a *canonical form*. For example, the canonical base unit representation of $[\mathrm{kg} \cdot \frac{\mathrm{m} \cdot \mathrm{s}^2}{\mathrm{kg} \cdot \mathrm{K}} \cdot \frac{\mathrm{m}^3}{\mathrm{s}}]$ is $[\mathrm{m}^4 \cdot \mathrm{s} \cdot \mathrm{K}^{-1}]$. As a convention, we enclose concrete unit expressions between square brackets. Some examples for dimensions, units and physical quantities are listed in Table A.2.

Note, that the names of the dimensions do not need to be declared in a hybrid model in order to be able to decide whether physical quantities are compatible with each other.

Dimension	Unit	Physical quantity
length	km, m, cm	5 km, 10 cm
speed	km/h, m/s	5 km/h, 10 m/s
plane angle	rad, deg	5 rad, 10 deg
currency	$, £	5 $, 10 £

Table A.2: Examples for dimensions, units and physical quantities.

This can be done by comparing the units of the quantities. If the units differ by a scale only, then the quantities are of the same dimension, thus they are compatible. We will return back to this later. According to this approach, dimensions are implicitly defined by units. For example, if a variable is declared with meter as unit, in χ this is written as $v :: [\mathrm{m}]$, then its dimension is automatically length. Variable v is then compatible with all other variables and quantities that can be expressed in meter. For example, $y :: [\mathrm{km}]$. To be able to deduce this, 'length' as dimension does not have to be explicitly declared in the model.

Definition 2 *A dimension is defined by its canonical base unit representation, where the base units are the fundamental SI units and the user defined base units.*

Definition 3 *Physical quantities are compatible with each other if their dimensions are equal, i.e., if their base unit representations are the same.*

The consequence of the above definitions is that all quantities may internally be represented in SI units, and conversion to other units needs to take place only whenever they are displayed. This is the case in ASCEND.

A.2.1 One-dimension, one-unit

There are several quantities, usually described in the literature as 'adimensional' the unit of which cannot be derived from the fundamental SI units. Examples are, depending on the problem domain, plane angle, solid angle, pH, currency, etc. These quantities are usually measured in *one-unit* that we denote as $[-]$. The ISO recommendation for the one-unit (ISO-31-1992) is '1'. However, this would collide in χ with the numerical literal 1, which is not the same as the one-unit. The one-unit is the one element of the multiplication operation, that is, for each unit $\mathbf{u}$: $\mathbf{u} \cdot [-] = \mathbf{u}$. A familiar appearance of the one-unit in applications is as the unit of angular displacement. Then, the angular speed is measured in $[-/\mathrm{s}] = [\mathrm{s}^{-1}]$ (in literature 1/s). However, quantities that represent solid angle, plane angle and pH are not compatible with each other, therefore, they should not belong to the same dimension. The solution is to introduce a new dimension for each of them. A hybrid language should facilitate the introduction of a new dimension along with a new base unit of measurement. The one-unit should still be available in a hybrid language, for other purposes. For example, the dimension of a mass concentration is mass/mass that

is measured in one-unit. The dimension of the physical quantities that are measured in one-unit is called *one-dimension*.

A.2.2 Formal representation

Let $\mathbf{u_1}, \mathbf{u}_2, ...\mathbf{u}_n$ denote the base units, and $a_1, a_2, ...a_n$ be scalars. The canonical base unit representation of a dimension $\mathbf{u}$ is $\mathbf{u} = \mathbf{u}_1^{a_1} \cdot \mathbf{u}_2^{a_2} \cdot ... \cdot \mathbf{u}_n^{a_n}$, where a component that has a zero exponent vanishes. For example, having only the fundamental SI units as base units, $[\mathrm{kg} \cdot \mathrm{m/s^2}] = [\mathrm{m}^1 \cdot \mathrm{kg}^1 \cdot \mathrm{s}^{-2} \cdot \mathrm{A}^0 \cdot \mathrm{K}^0 \cdot \mathrm{mol}^0] = [\mathrm{m}^1 \cdot \mathrm{kg}^1 \cdot \mathrm{s}^{-2}]$. The one-unit $[-]$ is then defined as

$$[-] \quad = \quad [\mathrm{m}^0 \cdot \mathrm{kg}^0 \cdot \mathrm{s}^0 \cdot \mathrm{A}^0 \cdot \mathrm{K}^0 \cdot \mathrm{mol}^0].$$

A.2.3 Operations on dimensions

Let $\mathbf{u_1}, \mathbf{u}_2, ...\mathbf{u}_n$ denote the base units, and $a_1, a_2, ...a_n, b_1, b_2, ...b_n$ and b be scalars. The following operations are defined on dimensions.

1. *multiplication* and *division*

$$\begin{aligned} \mathbf{u}_1^{a_1} \cdot \mathbf{u}_2^{a_2} \cdot ... \cdot \mathbf{u}_n^{a_n} \cdot \mathbf{u}_1^{b_1} \cdot \mathbf{u}_2^{b_2} \cdot ... \cdot \mathbf{u}_n^{b_n} &= \mathbf{u}_1^{a_1+b_1} \cdot \mathbf{u}_2^{a_2+b_2} \cdot ... \cdot \mathbf{u}_n^{a_n+b_n} \\ \mathbf{u}_1^{a_1} \cdot \mathbf{u}_2^{a_2} \cdot ... \cdot \mathbf{u}_n^{a_n} / \mathbf{u}_1^{b_1} \cdot \mathbf{u}_2^{b_2} \cdot ... \cdot \mathbf{u}_n^{b_n} &= \mathbf{u}_1^{a_1-b_1} \cdot \mathbf{u}_2^{a_2-b_2} \cdot ... \cdot \mathbf{u}_n^{a_n-b_n} \end{aligned}$$

2. *exponentiation*

$$(\mathbf{u}_1^{a_1} \cdot \mathbf{u}_2^{a_2} \cdot ... \cdot \mathbf{u}_n^{a_n})^b \quad = \quad \mathbf{u}_1^{a_1+b} \cdot \mathbf{u}_2^{a_2+b} \cdot ... \cdot \mathbf{u}_n^{a_n+b}$$

The dimension concept is related to a vector space, where the basis vectors are the base units, and the scalars used as exponents in unit expressions are the scalars over the vector space. This is a one-to-one transformation.

$$\mathbf{u}_1^{a_1} \cdot \mathbf{u}_2^{a_2} \cdot ... \cdot \mathbf{u}_n^{a_n} \longleftrightarrow a_1\mathbf{u}_1 \cdot a_2\mathbf{u}_2 \cdot ... \cdot a_n\mathbf{u}_n$$

The transform of the multiplication is the + operation, and the transform of the exponentiation is a multiplication by scalar in the vector space. Furthermore, the transform of the one-unit is the null vector in the vector space.

A.3 Units

A dimension may have several units of measurement. Units provide information about the display of the variables. They are analogous to base numbers in which a real number may be displayed. To simplify matters, we allow units of the same dimension to differ from each other only by a scale. The reason for this choice is explained later.

A.3.1 Unit declaration

In χ, the base units are the six fundamental SI units. These units, together with the one-unit are built-in language elements, that is, they do not need to be declared. New units, base or compound are declared with the keyword unit. In compound unit declarations, the following operators can be used: multiplication, division, exponentiation and multiplying by a scalar. For example,

```
unit km      = 10^3 m
,    Newton = kg · m/s^2
,    Pa     = Newton/m^2
,    kPa    = 10^3 m
```

A unit that differs by a scale from its base unit representation is called a *scaled unit*. For example km, and kPa are scaled units. An arbitrary unit expression can be brought into its *canonical form* by expressing it in the base units. Formally, the canonical form of an arbitrary unit $\mathbf{z}$ is $\mathbf{z} = t\mathbf{u}$, where $t \in \mathbb{R}$ and $\mathbf{u}$ is a canonical base unit expression. For example, $[\text{km/h}] = [\frac{1}{3.6}\,\text{m} \cdot \text{s}^{-1}]$.

There are several reasons to introduce a new unit. These are:

1. A known unit name is introduced that is used as a shorthand in the rest of the model. These can be scaled units, or common derived units. For example,

   ```
   unit km     = 10^3 m
   ,    Newton = kg · m/s^2
   ```

2. A new base unit is introduced. The reason for this is that a new base dimension, for example angle is going to be used in the model. In this case, radian is declared as new base unit. Another reason can be that a unit cannot be derived with the aid of unit operators. Examples for this are Celsius and Fahrenheit. These two units cannot be introduced as alternative units of temperature next to Kelvin, because subtraction is not allowed in unit expressions. This choice is explained below. For example,

   ```
   unit rad
   ,    Fahrenheit
   ```

3. A shorthand is introduced for a complex unit expression. Many physical constants used in equations have a complex dimension. For their units a shorthand may be introduced. For example, the outgoing flow in a cylinder shape reservoir is calculated as $Q = k\sqrt{h}$, where the units of k is

   ```
   unit cylinder = m^2.5/s
   ```

A.3.2 Why scaled units

It has been pointed out that according to this dimension concept all physical quantities could internally be represented in base units. This implies, that users could also work with base units only.

To allow to use scaled units such as kilometer or hour, is more for the convenience of users. Scaled units are important when entering or displaying physical quantities. For example, in a model where all lengths are measured in km, it is rather inconvenient to write all length literals in meter. Furthermore, physical constants used in equations are determined in certain units. Users should be able to specify these constants in arbitrary, i.e., scaled units.

A.3.3 Why scaled units only

Transformation to the base units is easy, because units of the same dimension differ from each other only by a scale. This would not be the case, if for example, subtraction or addition were allowed in unit expressions. Then, Fahrenheit could be defined as

$$\mathsf{unit}\ Fahrenheit = \tfrac{9}{5}[K] - 290.78[K]$$

This definition specifies how quantities expressed in Kelvin are transformed to Fahrenheit. For the transformation from Fahrenheit to Kelvin, the inverse formula needs to be calculated. Furthermore, some analysis and a calculus is necessary to determine how to transform, for example, quantities measured in Joule/Kelvin to Joule/Fahrenheit. In this proposal, such calculations are not dealt with.

A.4 Operations on physical quantities

A physical quantity is a pair consisting of a real number and a unit. For example $2\,[\mathrm{km}]$. This in comparison with *numerical quantities* represented by variables and literals of numerical types that are real numbers only. Formally, a physical quantity is a pair $\alpha\,\mathbf{z}$, where $\alpha \in \mathbb{R}$, and $\mathbf{z}$ is a unit and $\mathbf{z} = t\mathbf{u}$ is the canonical form of $\mathbf{z}$. To get rid of the double scalars α (in this example 2) and t (in this example 10^3), physical quantities may internally be represented in base units. For example, $2\,[\mathrm{km}] = 2 \cdot 10^3\,[\mathrm{m}] = 2000\,[\mathrm{m}]$. The canonical form of a physical quantity is obtained by expressing it in the base units. This can be done, by multiplying the two scalars: $\alpha\,\mathbf{z} = (\alpha \cdot t)\mathbf{u}$. The operations are defined on physical quantities expressed in their base units. To be able to apply these rules, a transformation may be necessary before or after the operations.

Equations, discrete-event statements, and functions may perform operations on quantities. Let $\alpha\mathbf{u}, \beta\mathbf{u}$ and $\beta\mathbf{v}$ denote physical quantities in their canonical form, and γ be a scalar. The following operations are defined on quantities.

1. *multiplication* and *division*

$$\begin{aligned} \alpha\,\mathbf{u}\cdot\beta\,\mathbf{v} &= \alpha\cdot\beta\,\mathbf{u}\cdot\mathbf{v} \\ \alpha\,\mathbf{u}/\beta\,\mathbf{v} &= \alpha\cdot\beta\,\mathbf{u}/\mathbf{v} \end{aligned}$$

2. *exponentiation*

$$(\alpha\,\mathbf{u})^{\gamma} = \alpha^{\gamma}\,\mathbf{u}^{\gamma}$$

3. *multiplying by scalar*

$$\gamma\cdot(\beta\,\mathbf{u}) = \gamma\cdot\beta\,\mathbf{u}$$

4. *addition* and *subtraction* of the same dimension

$$\begin{aligned} \alpha\,\mathbf{u}+\beta\,\mathbf{u} &= (\alpha+\beta)\,\mathbf{u} \\ \alpha\,\mathbf{u}-\beta\,\mathbf{u} &= (\alpha-\beta)\,\mathbf{u} \end{aligned}$$

5. *boolean operators* applied to quantities of the same dimension

$$\alpha\,\mathbf{u}\,\{>\,|\,\geq\,|\,=\,|\,\leq\,|\,<\}\,\beta\,\mathbf{u} \equiv \alpha\,\{>\,|\,\geq\,|\,=\,|\,\leq\,|\,<\}\,\beta$$

6. *derivation*
   The dimension of a derivative is

$$\mathrm{U}(\dot{h}) = \mathrm{U}(h)/\mathrm{s}$$

   where h is a physical quantity expression that does not contain physical quantity literals, and function U specifies the unit of such an expression. The value of $\dot{h}$ depends on the components that occur in h and the equations that determine their values.

7. *exponentiation with natural base* of quantities of one-dimension

$$e^{\alpha[-]} = e^{\alpha}$$

All operations other than the above ones need to be defined as functions, stating explicitly the unit transformation they perform. See Section A.6.3 for function definitions.

A.5 Types

So far, the notions of dimension and physical quantity have been defined together with their respective operators. Also, compatibility issues have been addressed. Thus, the

ground to specify the type system which must ensure these requirements has been laid.

Compatibility of physical quantities is based on dimension. Therefore, types of variables that represent physical quantities must define a dimension together with a unit of measurement. Since dimensions are implicitly defined by units, types with dimension can simply be declared with units only. In χ, types are introduced with the keyword type. Units in type declaration are enclosed in square brackets. For example,

```
type largelength    = [km]
,    normallength = [m]
,    force          = [Newton]
,    angle          = [rad]
```

The type system of χ is based on type equivalence. In case of types with dimension, this means that if two types define the same dimension, then they are equivalent. In the above example, the first two types *largelength* and *normallength* are equivalent, because they have the same dimension.

The dimension operators apply to types with dimension. These are multiplication, division and exponentiation. Unfortunately, raising to power is already used in χ as a type operator. It is the constructor of array types. Therefore, below the exponent is underscored to make a syntactic difference between the two type operators. For example,

```
type largearea   = largelength^2
,    normalarea = normallength^2
,    pressure    = force/normalarea
```

Again, the types *largearea* and *normalarea* are equivalent.

A.5.1 Types with dimension versus types without dimension

Variables of hybrid languages are usually divided into discrete and continuous variables. Continuous variables always represent physical quantities, so that their type must always have a dimension. However, discrete variables may also represent physical quantities. These are for example, constants used in equations. Take, as an example, the equation describing the volume of a substance in a tank represented by variable V, where the bottom area of the tank is denoted by A and the height of the substance by h: $Ah = V$. Variables V and h are continuous and A is discrete. Suppose, that the type of V is [m^3] and the type of h is [m]. The equation is only correct, if the dimension of A is [m^2].

With dimensions, a new division is introduced among variables and types: there are types and variables with dimension (representing physical quantities) and without dimension (representing numerical and other kinds of quantities). Continuous

variables always have a dimension. (The one-dimension is also considered a dimension.) Table A.3 summarizes the situation. In χ, discrete variables are declared using single colon (':'), whereas continuous variables are declared using double colon ('::'). This makes the syntactic difference between the continuous and discrete variable declarations when both have dimensions.

	continuous variable	discrete variable
with dimension	V:: [m^3] h :: [h]	A: [m^2] g : [m/s^2]
without dimension	-	r : real n: nat

Table A.3: Variables with dimension vs. variables without dimension

The main issue in the type system design is how to fit variables with dimension together with variables without dimension, more precisely, how to use physical and numerical quantities together. First, compatibility must be specified. For example, whether a variable with dimension can be assigned the value of a variable without dimension. Or, whether a function defined on types with dimension can be called with a parameter without dimension. Second, those operations, in which physical and numerical quantities can be used together must be defined, and the meaning of the operation must be specified.

The following approach is examined. Numerical types that do not have dimension (in χ, these are the types natural, integer and real) are scalars. They are not compatible with types that have a dimension. Variables and literals of these types can be used in exponentiation and multiplication as scalars. Other approaches to fit physical and numerical quantities together are examined at the end of the chapter, in Section A.9.

A.6 Physical consistency

In this section, the interpretation and the correctness requirements of hybrid language elements are revisited in the highlight of physical consistency.

A.6.1 Equations

An equation is only correct if the dimensions of the left and the right side expressions are the same. Furthermore, equations define equality of physical quantities. The consequence is that quantities of the same dimension but different units can be specified on the two sides of equations. For example, if two variables v_1 and v_2 are defined as v_1 :: [km], v_2 :: [m], the equation

$$v_1 = v_2,$$

is correct.

The benefit of this interpretation is that users do not have to specify scales. Without this, scaling of equations is necessary when some quantities must be calculated in scaled units, or if constants are given in scaled units. Take, as an example, the specification of the outgoing flow in a reservoir that has the shape of a cylinder. The diameter of the reservoir is D, the diameter of the outlet is d. The outgoing flow is $Q_{out} = k\sqrt{h}$, where $k = 0.785\,d^2\sqrt{\frac{2g}{1-\left(\frac{d}{D}\right)^4}}$, and in this formula d and D are measured in meter. Suppose, that $D = 75\,\text{cm}$, $d = 10\,\text{cm}$, and Q needs to be calculated in [liter/s]. Without scaling, that is in a language where D and d are variables of type real, if all variables are specified in SI units, the equation that calculates Q must be scaled as

$$Q = 10^{-3}k\sqrt{h}$$

This approach is error prone and if such a model is reused with Q measured in $[\text{m}^3/s]$, the scale in the equation has to be changed. Compare it with the flexibility and expressiveness of the χ model.

$$\begin{array}{l}
\mathsf{const}\ g : [\text{m/s}^2] = 9.87\,[\text{m/s}^2] \\
\mathsf{unit}\ \text{cm} \ \ = 10^{-2}\,\text{m} \\
,\quad \text{liter} = 10^{-3}\,\text{m}^3
\end{array}$$

$$\begin{array}{l}
\mathsf{proc}\ Res() = \\
|\ [\ d, D : [m],\ k : [\text{m}^2 \cdot (\text{m/s}^2)^{\frac{1}{2}}],\ h :: [\text{m}],\ Q_{out} :: [\text{liter/s}] \\
\sim|\ d := 10\,[cm]\,;\ D := 75\,[cm]\,;\ k := 0.785\,d^2\sqrt{\frac{2g}{1[-]-\left(\frac{d}{D}\right)^4}} \\
=|\ Q = k\sqrt{h} \\
]\ |
\end{array}$$

Note also that above d and D are specified in centimeter, yet the value of k is correctly calculated. In this model, the original form of the equation is preserved. It is easy to read and it is much less error prone. If Q needs to be calculated in m^3/s, then only the declaration of Q has to be changed.

A.6.2 Connections

Only variables of the same dimension can be connected, and a connection defines equality of physical quantities. Since in χ a connection defines an equation between the connected variables, the interpretation of connections follows from the interpretation of equations. Take as an example the following two processes.

$$\textsf{unit}\ \mathrm{km} = [10^3\,\mathrm{m}]$$

$$\begin{array}{ll}
\textsf{proc}\ P_1(v :: -\circ[\mathrm{km}]\ldots) = & \textsf{proc}\ P_2(v :: -\circ[\mathrm{m}]\ldots) = \\
\mid [\ v_1 :: [\mathrm{km}] & \mid [\ v_2 :: [\mathrm{m}] \\
\dashv\ v -\circ v_1 & \dashv\ v -\circ v_2 \\
\quad\vdots & \quad\vdots \\
]\mid &]\mid
\end{array}$$

The above connection defines that the two physical quantities, $P_1.v_1$ and $P_2.v_2$ are equal. Such interpretation of connections supports re-use of models, because models using different units of measurement can be connected without alteration as is illustrated in the above example.

A.6.3 Functions

Functions may perform unit transformation, that is, their domain and range may have components with dimension. For example, the *sin* function may be specified as

$$\textsf{func}\ \sin(a : [rad]) \longrightarrow [-]$$

In χ, the formal parameters of a function match the actual parameters if their types are equivalent. As a consequence, functions that operate on physical quantities define operations on dimensions independent of the actual unit of the quantities. Suppose, that degree is defined as another unit for angle.

$$\textsf{const}\ \pi : \mathrm{real} = 3.14159265$$
$$\textsf{unit}\ \mathrm{deg} = 180/\pi\,\mathrm{rad}$$

By defining variables r, d and s as

$$r :: [\mathrm{rad}],\ d :: [\mathrm{deg}],\ s : [-]$$

the following statements are all correct.

$$s := sin(r); s := sin(\pi\,[\mathrm{rad}]); s := sin(d); s := sin(45\,[\mathrm{deg}])$$

Polymorphic dimension functions

There are several functions that can be used on different dimensions which perform a well defined dimension transformation. All functions related to exponentiation are of this kind. One of them is the function *safesqrt*, which defines the square root of negative numbers to be 0. Using polymorphism, it is defined as

```
func safesqrt(v : [any²]) → [any] =
| [ [ v ≥ 0 [any²] ⟶ ↑ √v
  ⟦ v < 0 [any²] ⟶ ↑ 0 [any]
  ]
] |
```

Despite the exponent 2 in the parameter type, it is not required that in the base unit representation of the actual type all base units have an even exponent. Rather, the above definition is equivalent to the definition

```
func safesqrt(v : [any]) → [any^½]
```

The reason for this is that the exponents in the base unit representation of many physical constants are fractions. An example is variable k in Section A.6.1. Its dimension is $[\mathrm{m}^2 \cdot \sqrt{\frac{\mathrm{m}}{\mathrm{s}^2}}] = [\mathrm{m}^{\frac{5}{2}} \cdot \mathrm{s}^{-1}]$.

Conversion functions

Conversion functions can be used to convert between physical quantities and numerical quantities, and to convert between physical quantities of different dimensions.

An example for a conversion function that converts physical quantities to real numbers is *dim2real*. How such a function should be defined is a subject of agreement. It can, for example, return the value of the physical quantity measured in base units. For example, *dim2real*$(5\,[\mathrm{km}]) = 5000$. Another implementation of *dim2real* can be that it returns the value of quantities measured in their units. That is, it returns 5 in case of $5\,[\mathrm{km}]$. In this case, an agreement is necessary to determine the units of complex expressions, like $2\,[\mathrm{m}] + 5\,[\mathrm{km}]$.

A general function that converts numbers to physical quantities requires that the unit is specified as a parameter. In χ, this is not possible because a type name or unit expression itself can not be used as a value. A simple solution instead is to use multiplication as in

$$l ::= x \cdot 1[\mathrm{km}]$$

where the variables are defined as x : real, l :: [km]. For converting real numbers to physical quantities by a function, the user has to define explicit conversion functions. As an example, assume that measurement data is available in a file. This data is a sequence of real numbers, that represents lengths measured in centimeter. A model that uses this data, reads from the file a sequence of real numbers and converts them into quantities of type length, using the following conversion function.

```
func real2cm(e : real) → [m] =
| [ ↑ e · 10⁻² [m] ] |
```

An often used conversion function is *real2one* that converts real numbers to physical quantities of one-dimension. This function can be built-in.

Conversion can be defined between quantities of different dimensions. The first example below calculates the area of a circle from the radius, the second one converts Fahrenheit to Kelvin.

$$\mathsf{const}\ \pi : \text{real} = 3.14159265$$

$$\mathsf{func}\ \text{circlearea}(r : [\text{m}]) \rightarrow [\text{m}^2] =$$
$$|\ [\ \uparrow \pi \cdot r^2\ ]\ |$$

$$\mathsf{func}\ \text{F2K}(f : [\text{Fahrenheit}]) \rightarrow [\text{K}] =$$
$$|\ [\ \uparrow \tfrac{5}{9}[\tfrac{\text{K}}{\text{Fahrenheit}}] \cdot f - 290.78\,[\text{K}]\ ]\ |$$

A.6.4 Discrete-event statements

Next to the continuous part, the discrete part of hybrid models must also be physically consistent, otherwise, the whole model becomes inconsistent. In general, all expressions that are defined on physical quantities must be dimensionally correct, and must be evaluated according to the operation definitions given in Section A.4. To see why consistency must be ensured in the continuous as well as in the discrete-event part, take for example, the following process.

$$\mathsf{proc}\ P() =$$
$$|\ [\ v_1 :: [\text{m/s}],\ v_2 :: [\text{km/h}],\ k : real$$
$$\sim|\ k := 2.5$$
$$=|\ v_1 = k \cdot v_2$$
$$\quad|\ \ldots \nabla\, v_1 > v_2 \ldots$$
$$]\ |$$

Suppose, that v_1 and v_2 are positive quantities. Then, if the equation is interpreted as a relation on physical quantities, then $v_1 > v_2$ (since $k > 1$). Therefore, the state-event specification should also be evaluated according to the definition of boolean operators on physical quantities, in order to ensure that it evaluates to true.

Time-out statements specify an amount of time that a process must wait. In these statements, the amount must be a physical quantity with dimension time. For example, $\Delta 2\,[\text{s}]$ or $\Delta 10[\text{h}]$. Furthermore, simulation time, represented in χ by the special variable τ, is also a physical quantity of dimension time. Its unit is by default [s]. This is further explained in Section A.8.

Correctness of discrete-event statements are specified using type equivalence. In the case of types that have a dimension, type equivalence means identical dimensions. Below, we re-formulate the type requirements of discrete-event statements for types with dimension.

1. *assignment statements*
   The two sides of an assignment statement must have the same dimension.

2. *send and receive actions via discrete channels*
   Physical quantities may only be sent over discrete channels that have types with the same dimension. This requires that discrete channels can be defined with types with dimension. Such kinds of channels can be used for example, to send samples of physical quantities or constants.

3. *read actions from input files*
   A physical quantity may be read from an input file and stored in a variable that has the same dimension as that of the quantity.

4. *function calls*
   The dimensions of the formal and the actual parameters must be the same.

5. *return statement*
   In functions, the dimension of the returned quantity must be the same as that of the return type.

As has been indicated earlier, the actual unit of a variable (i.e., the unit, in which a variable is declared) is only important for displaying physical quantities in output and it may be used in conversion functions. In χ, output can be written to file or to the standard output. An agreement must be made about the output format.

A.7 Numerical aspects

Numerics plays an important role in hybrid models, therefore, the numerical aspects of the presented concept must be addressed. For the representation of the variables, it is important that their range can be assessed. This allows to scale variables when the equations are solved numerically by solvers. Scaling in this case means that the absolute and the relative tolerances of individual variables are given to solvers.

One possibility is to represent all variables in SI units. In this case, the scale of their units can be used as range. For example, if lightyears are used to measure length

$$\mathsf{unit}\ \text{lightyear} = [60 \cdot 60 \cdot 24 \cdot 356 \cdot 3 \cdot 10^8 \text{m}] = [9.22 \cdot 10^{15}\ \text{m}]$$

then all variables of unit lightyear are in the range 10^{16}. This approach assumes that the units of the variables are chosen such that the values of the variables lie around 1-10 [lightyear], perhaps with a wider interval.

More accurate assessment can be made, if the types of the variables provide more numerical information. For example, in gPROMS, the type of the continuous variables include the default value, and the upper and lower bounds on the value of the variables of this type.

A.8 Unit of simulation time

In the current χ language specification, time-out statements use a value of type real to denote the amount of time a process has to wait. This value is interpreted as number of seconds. This is based on the implicit assumption that simulation time is measured in seconds. In other words, each tick of the global clock is one second. Practically, when executing the model by a simulator, all time dependent quantities are represented in seconds.

It may however not always be the most convenient unit of time. Suppose, that a model describes the behavior of a system on bases of hours. In such a model, the statement $\Delta 1\,[\mathrm{h}]$ is simulated by 3600 ticks of the global clock. Also, time derivatives are calculated in seconds. Take, for example, a variable v that measures speed in [km/h]. Suppose, that its time derivative is $1\,[\mathrm{km/h^2}]$ specified by the equation

$$v' = 1\,[\mathrm{km/h^2}]$$

Following the calculus of physical quantities, the value of v' measured in $[\mathrm{m/s^2}]$ is $value(v') \cdot \frac{1}{3.6^2}$. In fact, the equation $v' = \frac{1}{3.6^2}$ is integrated in continuous phases by numerical solvers. Whether this simulation is less efficient then as if the simulation time was measured in hours depends on the numerical solvers used.

A simulation language should support to set the simulation time unit. Practically, the model need not be changed, only internally all time dependent physical quantities have to be represented in the simulation time unit. To summarize, physically correct modelling means that when the unit of simulation time is changed, the model need not be altered, since time dependent quantities specify the timing of the model independently of the simulation time unit.

A.9 Alternatives

The decision of how to use variables with dimension together with variables without dimension in a hybrid language is a crucial one. It has consequences for the whole language. Therefore, it needs to be a conceptually well based decision, such that consequences of a particular choice can systematically be evaluated.

In this chapter numerical quantities has been treated as scalars that are not compatible with physical quantities. The most striking consequence of this choice is that dimensions must always be specified whenever a physical quantity is treated. An extreme example is the calculation of variable k in Section A.6.1:

$$k := 0.785\, d^2 \sqrt{\frac{2g}{1\,[-] - \left(\frac{d}{D}\right)^4}}$$

Here, the one-dimension must be specified in order to be able to execute subtraction. On the other hand, models are clear and there are no ambiguities. The type system is clearly divided into types with dimension and types without dimension, each with its respective operators. The operations on the physical quantities and the numerical quantities are well defined. Furthermore, operations on purely dimensionless quantities remain unchanged.

Numerical quantities as physical quantities of one-dimension

Another approach is to treat numerical quantities as being physical quantities having one-dimension. The attractive feature of this choice is that in calculations like the assignment of variable k above, the one-dimension need not be specified. Variable k can be assigned as

$$k := 0.785\, d^2 \sqrt{\frac{2g}{1-\left(\frac{d}{D}\right)^4}}$$

Also, multiplication by a numerical quantity has the same result as in the first choice, since multiplying by a scalar and multiplying by a quantity of one-dimension have apparently the same result. Compare the following two expressions $3 \cdot 5\,[\text{km}]$ and $3\,[-] \cdot 5\,[\text{km}]$. In the first expression 3 is a scalar so it is a multiplication by scalar, in the second one 3 is a physical quantity of one-dimension, so it is a multiplication by a physical quantity. The outcome in both cases is 15 [km].

However, numerical quantities can be used in many ways that are not defined for physical quantities. For example, they can be exponents of physical quantities as in $(2\,[\text{m}])^2$. This operation is only defined if 2 is a scalar and not a physical quantity $2\,[-]$. The interpretation of numerical quantities therefore, depend on the context that may lead to ambiguities in the language. Alternatively, all operations defined on numerical quantities may be redefined for quantities of one-dimension.

Dimension and unit determined from context

There is a choice to use numerical quantities as short notation for physical quantities in situations where the dimension and units can easily be determined from the context. Take, for example, the following assignment

$$v ::= 2;$$

where variable v is declared as $v :: [\text{km/h}]$. It can be argued that 2 represents $2\,[\text{km/h}]$. However, if the same variable is assigned the value $2\,[\text{m/s}] + 5$, it is not clear what the unit of 5 should be.

In a similar approach, the dimensions and units of variables can be determined in certain situations from their use. In ASCEND, for example, the dimension can be

left undefined in variable declarations. Later, the dimensions of such variables are deduced from their use. This is rather error prone, because the translating tool can choose a dimension for a variable such that an otherwise incorrect expression becomes correct. In this way, errors can easily go undetected.

In short, the simplicity gained by allowing incomplete specifications is not worth the language ambiguity and the erroneous models resulting from this approach.

Appendix B

Hybrid simulation

The χ simulator creates an executable file called `model.exe` for the χ model specified in file `model.chi`. The model can be simulated by executing `model.exe`.

B.1 Simulation output

The results of the simulation are the file `cvar.log`, an optional trace file `trace.log` and user output. User output can be generated by the model itself, either by writing to output files or by writing to the standard output. In the following, fractions are used from the output generated by executing the control model specified in Section 5.5.

The file `cvar.log`

This file contains the value of the continuous variables logged at timesteps. How these timesteps are selected is defined by the *stepsize* option. Furthermore, the values of the variables are logged before and after discrete phases. This file can be used to plot diagrams. The contents of `cvar.log` looks, for example, like

```
t        S.h      Tank.V  Tank.Qo
0        4        10      2       1
5        4        10      2       0
10       4        10      2       1
10       16       40      4       1
15       9.00001  22.5    3       0
17.7526  5.99999  15      2.44949 1
17.7526  5.99999  15      2.44949 1
20       5.99999  15      2.44949 1
20       18       45      4.24264 1
25       10.5147  26.2868 3.24264 0
```

In the first column the time is printed. In the first row, the names of the variables are listed. In consecutive rows, the values of the variables are printed, separated by tabulars. The last column is optional. Rows ending with 0 belong to intermediate steps, whereas rows ending with 1 show the values of the variables before or after a discrete phase.

The file `trace.log`

This is a file from which the complete simulation can be reconstructed. It records the execution of statements and the continuous state of the system at points that may be of interest. It records when events occur, and enlists the events that are awaited at a certain time point. The generation of this file is optional.

At the beginning of the file the simulation options and the continuous variable structure is printed. The values of the continuous variables and their time derivatives are printed each time an initial state is calculated. This is not the case if the short trace option is set. An example for the beginning of this file is

```
=====================================================
Trace file generated by the Chi simulator vers 0.5
   March 24 1999  11:06:45
=====================================================
Simulation parameters:

Stepsize:                      5
End time:                      100
Time tolerance:                -6
==DDASRT==
Output precision:              6
Smallest number printed:       1e-100
Biggest number printed:        1e+100
Relative tolerance:            -6
Absolute tolerance:            -6
==Initial state solver==
Relative tolerance:            -10
Scaling lower threshold:        -1
Nonlinearity:                  3
Maximum number of iterations:  20
Starting damping factor:       1
Minimum dumping factor:        0
Nonlinear solver number:       2


================= Begin simulation =================
```

```
sched:: Activate Tank
sched:: Activate Supply
sched:: Activate Control

Continuous variables:
0       S.h
                Tank.h
                Control.h
1       Tank.V
                Tank.V
2       Tank.Qo
                Tank.Qo

Differential variables:
        Tank.V

Algebraic variables:
        S.h
        Tank.Qo

Substitute variables:

After initialization:
At 0.000000
Next delta = -
Number of nablas: 0
name    value   prime
S.h     4       0
Tank.V  10      -1.35975e-29
Tank.Qo 2       0

sched:: Run active processes!
sched:: process: Tank
  executes statement 0
sched:: process: Supply
  executes statement 1
sched:: process: Control
  executes statement 1
sched:: nabla event added: process Control number 0
sched:: Active processes finished!
sched:: New state calculated:

At 0.000000
Next delta = 10.000000
```

```
Number of nablas: 1
name    value   prime
S.h     4       0
Tank.V  10      0
Tank.Qo 2       0

next step: 5
```

In the rest of the file intermediate steps and discrete phases are registered. An example for this is:

```
======================= time 15 ======================

At 15.000000
Next delta = 20.000000
Number of nablas: 1
name    value   prime
S.h     9.00001 -1.2
Tank.V  22.5    -3
Tank.Qo 3       -0.2

No nablas found, no deltas finished.
solv:: Step
next output at:20

======================= time 17.7526 =====================

At 17.752572
Next delta = 20.000000
Number of nablas: 1
name    value   prime
S.h     5.99999 -1.2
Tank.V  15      -3
Tank.Qo 2.44949 -0.2

root is found by solver
  Solver correctly found a root
sched:: Activate Control
sched:: Run active processes!
sched:: process: Control
  executes statement 4
sched:: Activate Tank
sched:: process: Tank
  executes statement 0
```

```
sched:: process: Control
  executes statement 1
sched:: nabla event added: process Control number 0
sched:: Active processes finished!
sched:: New state calculated:

At 17.752572
Next delta = 20.000000
Number of nablas: 1
name    value   prime
S.h     5.99999 -1.2
Tank.V  15      -5.16988e-26
Tank.Qo 2.44949 -0.2

next step: 20
```

B.2 Simulation options

Simulation is tailored to each model by using simulation options. A default option file `default.opt` specifies the default values of the options. These can be re-defined by a model specific option file called `model.opt` where the model itself is specified in file `model.chi`. Finally, command line options overwrite previously specified options. Each simulation option has a short and a long name. Some options, like tolerances define a negative exponent, for example, -r 6 defines the relative tolerance to be 10^{-6}.

Timing

Simulation starts by default at time 0 and runs until the *end time* set by the *endtime* option. The value of the continuous variables can be examined at equidistant time points. This is the *simulation stepsize*, set by the *stepsize* option. The continuous variables are logged into the file `cvar.log` at each step.

-s -StepSize Whenever the stepsize is positive, the continuous variables are logged at time points that are multiples of the stepsize. Argument 0 means that output is given at time steps chosen by the DAE solver.

-e -EndTime The end time of the simulation. If no more discrete actions are left, the DAEs are integrated until the end time.

DASSL options

The minimum stepsize of DASSL changes in the course of the simulation. Instead of calculating it each time when DASSL is called, a constant *time tolerance* is used.

Discrete phases that occur closer to each other than the time tolerance are scheduled after one another, without changing the simulation time.

-r -RelTolerance The relative tolerance used by DASSL. Currently, this is used for all continuous variables. The argument is used as negative exponent.

-a -AbsTolerance The absolute tolerance used by DASSL. Currently, this is used for all continuous variables. The argument is used as negative exponent.

-tt -TimeTolerance A upper approximate of the minimum stepsize of DASSL.

NLEQ options

-nr -NlRelTol The relative tolerance used by NLEQ. Currently, this is used for all continuous variables. The argument is used as negative exponent.

-x -Xscal The lower threshold of the weight in the weighted norm used by NLEQ. The argument is used as negative exponent.

-nl -Nonlinearity The initial state problem is classified as

1. linear
2. mildly nonlinear
3. highly nonlinear
4. extremely nonlinear

-mi -MaxIt The maximum number of permitted iteration steps of NLEQ.

-sd -StartDamp The starting damping factor of NLEQ. When NLEQ finds a correct direction for a step, a steplength control algorithm calculates the steplength - the damping factor. This option sets the length of the first step.

-md -MinDamp The minimum damping factor.

-sn -SolvNumb The version of NLEQ to be used. NLEQ1 implements the standard algorithm. NLEQ2 has an additional rank reduction device to solve numerically sensitive problems. This for more computation efforts.

Output options

-t -Trace Generate trace file `trace.log`.

-vb -Verbose Generate a verbose trace file `trace.log`. More detailed information is generated about the scheduling. This information is meant for debugging.

-c -Accuracy The number of digits displayed when writing the values of continuous variables.

-nd -NoDOutput Do not record the values of the variables into file `cvar.log` at discrete phases, i.e., only at multiples of the timestep.

-nc -NoCOutput Do not record the values of the variables into file `cvar.log` at intermediate steps. That is, the variables are recorded only before and after a discrete phase.

-l -LastColumn Generate the last column of zeros and ones in file `cvar.log` that indicates whether a particular output is made at intermediate step (0), or before or after a discrete phase (1).

-st -ShortTrace Generate a short trace file, i.e., the value of the continuous variables are not printed.

-o -OutFile Set the base of the names of the files `cvar.log` `trace.log`. The '.log' extension cannot be changed.

-ss -SetSeed Seed of the random number generator.

-cl -ClipLow The smallest number printed. The argument is a negative exponent.

-ch -ClipHigh The biggest number printed. The argument is a positive exponent.

Miscellaneous options

-h -Help List the simulation options.

-v -Version Print version number.

-f -OptionFile Use the argument as option file name.

-g -GenOpt List current option settings.

Index

Samenvatting

Vele hedendaagse industriële systemen tonen gemengd discrete-event/continue-tijd gedrag. Typisch voorbeelden zijn batchprocessen van de procesindustrie, digitaal bestuurde fabrieken, auto's en vliegtuigen. Modellen spelen een belangrijke rol in het ontwerpproces van zulke systemen. Ze bieden de mogelijkheid om ontwerpbeslissingen in een vroeg stadium te evalueren. Modellen die in een specificatie formalisme geschreven zijn, bevatten alleen de relevante systeem aspecten. Deze modellen leveren een belangrijke bijdrage aan het begrip van de systemen. Hybride systemen worden steeds meer gebruikt in toepassingen, waarin veiligheid van groot belang is, waardoor de betrouwbaarheid van hybride systemen steeds belangrijker wordt. Voor verificatie doeleinden, formele semantische modellen en kaders kunnen worden gebruikt.

De χ taal is een hybride specificatieformalisme, die geschikt is voor het beschrijven van discrete-event, continue-tijd en hybride systemen. Het is een parallelle taal, waarvan het discrete-event gedeelte is gebaseerd op Communicerende Sequentiele Processen (CSP [Hoare, 1985]), en het continue-tijd gedeelte is gebaseerd op algebraïsche differentiaalvergelijkingen (DAEs). Modellen die in χ geschreven zijn, kunnen door de χ simulator uitgevoerd worden.

De bijdrage van dit proefontwerp is tweezijdig. Ten eerste, de semantiek van de hybride χ taal wordt gespecificeerd. Dit is een uitbreiding van de discrete-event semantiek die eerder ontwikkeld werd. Ten tweede, de al bestaande discrete-event χ simulator werd uitgebreid om hybride modellen te kunnen uitvoeren. Om een specificatie taal geschikt te maken voor het ontwerpen en analyseren van complexe systemen, moet de taal een nauwkeurig gedefinieerde semantiek hebben. Een hybride taalsemantiek moet de puur continue-tijd en de puur discrete-event functionaliteit behouden, en ook beide functionaliteiten integreren. In het geval van de χ taal is het ook belangrijk dat modellen efficiënt kunnen worden uitgevoerd, gebruikmakend van bestaande numerieke algoritmen.

Differentiële en algebraïsche variabelen zijn gedefinieerd met hun respectievelijke operatoren in de χ taal. Dit onderscheid legt de nadruk op de fysische rol van de variabelen waardoor het ontwerp van fysisch correcte modellen wordt ondersteund. Steady-state initialisatie wordt door de taal ondersteund. Op dit moment, kunnen DAEs met index 0 en 1 worden opgelost. Systemen met hogere indexen kunnen met substitutievergelijkingen gemodelleerd worden.

De semantiek van modelcompositie is een belangrijk onderwerp in hybride talen. Modelcompositie van puur discrete modellen is goed gedefinieerd door CSP. Dit proefont-

werp behandelt de continue-tijd aspecten van modelcompositie. In de χ taal worden de continue variabelen van sub-modellen aan elkaar gekoppeld door middel van kanalen. Nà het bestuderen van bestaande hybride talen zijn algebraïsche vergelijkingen als de betekenis van connecties tussen continue variabelen gekozen. Dit, samen met het gebruiken van lokale variabelen, leidde tot doorzichtigere en robuustere modellen. Verder worden state-event condities alleen in consistente toestanden geëvalueerd; dit leidt tot meer voorspelbare modelgedrag.

Het wiskundige model van de continuous-state berekeningen is gespecificeerd. De hybride taalsemantiek is ook gespecificeerd in de vorm van een rekenmodel. Volgens dit model worden de discrete fasen verdeeld in sub-fasen, en elke sub-fase eindigt met een consistente toestand berekening. In elke sub-fase werken processen alleen met lokale variabelen. Interactie tussen sub-modellen wordt beperkt tot expliciete communicatie. Dit maakt het redeneren over het gedrag van de modellen eenvoudiger.

De χ simulator werd uitgebreid om hybride modellen te kunnen uitvoeren door integratie met een niet-lineaire solver om het initiële toestand probleem op te lossen, en met een DAE solver om de DAEs tijdens de continue fasen op te lossen. Om de prestatie te verbeteren, worden de lokale continue variabelen van de processen dynamisch gekoppeld op een globale set variabelen waarmee de solvers werken. De variabelen kunnen in de χ simulator op een geavanceerde manier worden gelogged zodanig dat hun verloop met behulp van andere softwarepakketten gevisualiseerd kan worden. De gebruiker kan ook een gedetailleerde tracefile genereren.

In de toekomst, moeten formele methoden en technieken worden ontwikkeld, gebaseerd op de operationele semantiek die in dit proefontwerp is gedefinieerd, voor de formele analyse van hybride χ modellen. Een mogelijke uitbreiding op de χ taal is om dimensieinformatie te bewaren in de datatypen. In dit proefontwerp, wordt een voorstel gedaan om fysische correctheid te controleren op basis van die informatie. Om meer soorten fysisch systemen te kunnen modelleren in χ, is het noodzakelijk om zogenaamde instantaneous vergelijkingen te kunnen specificeren. De prestatie van de χ simulator kan verbeterd worden door de vergelijkingen te optimaliseren met symbolische analyse en manipulatie.

Összefoglaló

Sok mai ipari rendszer kevert, diszkrét/folytonos karakterű. Tipikus példaként említhető a vegyiparban használt tételenkénti feldolgozás, a digitálisan vezérelt üzemek, autók, repülőgépek. Ezen rendszerek tervezésénél a modellezésnek fontos szerepe van, hiszen már a tervezés egy korai fázisában ki lehet értékelni egy-egy tervezői megoldást. A specifikációs formalizmussal leírt modellekben csak a rendszer vonatkozó, fontos tulajdonságai jelennek meg. E modellek jelentősen hozzájárulnak a rendszer dinamikájának megértéséhez. Hibrid rendszereket növekvő mértékben alkalmaznak biztonság kritikus felhasználásokban, ezért megbízhatóságuk egyre fontosabbá válik. Helyesség bizonyításra formális szemantikai modellek és bizonyítási eljárások használhatók.

A χ nyelv egy hibrid specifikációs formalizmus, mely diszkrét idejű, valamint folytonos, és kevert idejű rendszerek leírására is alkalmas. Ez egy párhuzamos nyelv, amelyben a diszkrét rész a Kommunikáló Szekvenciális Processzek-en alapul (CSP [Hoare, 1985]), a folytonos részt pedig algebrai differenciál egyenletek segítségével lehet megadni (DAEs). A χ nyelvben írt modellek a χ szimulátor segítségével futtathatók.

A jelen doktori dolgozat kettős céllal készült. Egyrészt, a hibrid χ nyelv szemantikáját definiálja. A szemantika egy, a project elején már létező diszkrét szemantikára épül. Másrészt, egy már létező diszkrét szimulátor hibriddé lett kibővítve. Ahhoz hogy egy specifikációs nyelv összetett rendszerek tervezésére és vizsgálatára alkalmas legyen, először is jól definiált nyelvszemantikára van szükség. Hibrid nyelvek esetén a szemantikának meg kell őriznie a tisztán folytonos és a tisztán diszkrét funkciókat, míg a kettőt zökkenőmentesen kell ötvöznie. A χ nyelv esetében továbbá az is fontos, hogy jelenlegi numerikus technikákkal, a modelleket gazdaságosan, gyorsan lehessen futtani.

A χ nyelvben differenciális és algebrai változókat különböztetünk meg. E megkülönböztetés a változók fizikai szerepét emeli ki, és így fizikailag korrekt modellek specifikációját segíti elő. A nyelvben lehetőség van a rendszerek stabil kiinduló-állapotba hozására. Jelenleg egyes és kettes indexű differenciál egyenleteket lehet közvetlenül megoldani. Magasabb indexszámú egyenleteket helyettesítő egyenletek használatával lehet modellezni.

A model kompozíció szemantikája egy lényeges pont a hibrid nyelvekben. A tisztán diszkrét modellek kompozícióját a CSP definiálja. E dolgozat a model kompozíció folytonos jellegű problémaival foglalkozik. A χ nyelvben az almodellek folytonos változóit úgynevezett csatornák kötik össze. Miután több hibrid nyelvet is meg-

vizsgáltunk, úgy határoztunk, hogy az összekötött folytonos változók algebrai egyenletet definiálnak. Figyelembe véve, hogy a processzek csak lokális vátozókat használnak, e választás jelentős mértékben hozzájárult a modellek érthetőségének és kifejező képességének növeléséhez. Továbbá, az állapot-események feltételei csak konzisztens állapotban értékelődnek ki, amivel a modellek működése kiszámíthatóbbá vált.

A dolgozatban ismertetjük a folytonos állapottér kiszámításának a matematikai modelljét, valamint a hibrid nyelvszemantikát; ez utóbbit egy kiszámítási modell formájában adjuk meg. E modellben a diszkrét fázisok alfázisokra oszlanak, s minden egyes alfázis egy-egy konzisztens kiinduló–állapot számításával fejeződik be. Az egyes alfázisokban, a processzek csak lokális változókkal dolgoznak. Az almodellek közötti kölcsönhatás kizárólag nyílt kommunikáció útján lehetséges, mely megkönnyíti a modellek működésének megértését.

A χ szimulátor ki lett bővítve, úgy, hogy hibrid modelleket is tudjon futtani. Egy nem-lineáris egyenlet megoldóprogram lett beépítve, a kiinduló állapot problémájának megoldására, valamint egy differenciál egyenlet megoldóprogram, amely a differenciál egyenleteket oldja meg a folytonos fázisokban. A teljesítmény növelése céljából, a lokális folytonos változókat dinamikusan átképezzük egy globális változó halmazra, mely halmazt a megoldóprogramok használnak. A χ szimulátorban a változók értékeit különböző módokon lehet elmenteni, úgy, hogy azokat más programokkal is meg lehet jeleníteni. Felhasználói kérésre, egy részletes nyomkövető fájl készül.

A jövőben javasoljuk az itt leírt operációs szemantikán alapuló formális modellek és technikák kidolgozását, a hibrid χ modellek formális analízisére. A χ nyelv egy lehetséges bővítése, hogy az adattípusokban információt tárolunk a változók dimenziójáról. E dolgozatban egy javaslatot teszünk a modellek fizikai helyességének ellenőrzésére ezen információ alapján. A modellezhető fizikai folyamatok körét kibővítendő, a jövőben szükség lesz úgynevezett pillanatnyi egyenletekre. A χ szimulátor teljesítményét tovább lehet növelni, ha az egyenleteket szimbólikus elemzés és átrendezés útján optimalizáljuk.

Curriculum Vitae

Georgina Fábián was born on May 1, 1971 in Budapest, Hungary. She completed her secondary school education in Fazekas Mihály Grammar School, in Budapest, where she attended the class specialized in mathematics.

She received the B.Sc. degree in 1992 and the M.Sc. degree in 1994 in Computer and Information Science from Eötvös Loránd University of Science, Budapest. Her master thesis was concerned with the specification and analysis of distributed data types using the UNITY method. In the period March-June 1993, she spent a trimester at Eindhoven University of Technology, The Netherlands, as a participant in a student exchange program.

In 1994, she started the post-masters programme Software Technology at Eindhoven University of Technology. She finished this programme in 1996 with a design project on a simulator for combined continuous-time/discrete-event models at the Systems Engineering Group. She continued to work on this project; in 1996 she joined the Systems Engineering Group, where she carried out research on the hybrid semantics of the χ language and she worked on the χ simulator, which is reported in this thesis.