Process algebra for dynamic system modeling[*]

J.C.M. Baeten[1], D.A. van Beek[2], J.E. Rooda[2]
[1]Department of Mathematics and Computer Science
[2]Department of Mechanical Engineering
Eindhoven University of Technology, P.O.Box 513
5600 MB Eindhoven, The Netherlands
{j.c.m.baeten,d.a.v.beek,j.e.rooda}@tue.nl

Abstract

Process algebra is the study of distributed or parallel systems by algebraic means. Originating in computer science, process algebra has been extended in recent years to encompass not just discrete event, reactive systems, but also continuously evolving phenomena, resulting in so-called hybrid process algebras. A hybrid process algebra can be used for the specification, simulation, control and verification of embedded systems in combination with their environment, and for any dynamic system in general. As the vehicle of our exposition, we use the hybrid process algebra χ (Chi). The syntax and semantics of χ are discussed, and it is explained how equational reasoning can simplify, among others, tool implementations for simulation and verification. Finally, a bottle filling line example is introduced to illustrate system analysis by means of equational reasoning.

1 Introduction

1.1 Definition

Process algebra is the study of distributed or parallel systems by algebraic means. The word ‘process’ here refers to *behavior* of a *system*. A system is anything showing behavior, such as the execution of a software system, the actions of a machine or even the actions of a human being. Behavior is the total of events or actions that a system can perform, the order in which they can be executed and maybe other aspects of this execution such as timing, probabilities, or continuous aspects. Always, the focus is on certain aspects of behavior, disregarding other aspects, so an abstraction or idealization of the ‘real’ behavior is considered. Instead of considering behavior, we may consider an *observation* of behavior, where an action is the chosen unit of observation. As the origin of process algebra is in computer science, the actions are usually thought to be discrete: occurrence is at some moment in time, and different actions are separated in time. This is why a process is sometimes also called a *discrete event system*.

The word ‘algebra’ denotes that the approach in dealing with behavior is algebraic and axiomatic. That is, methods and techniques of universal algebra are used. A process algebra can be defined as any mathematical structure satisfying the axioms given for the basic operators. A process is an element of a process algebra. By using the axioms, we can perform *calculations* with processes. Often, though, process algebra goes beyond the strict bounds of universal algebra: sometimes multiple sorts and/or binding of variables are used.

[*]Invited chapter to CRC Handbook on Dynamic System Modeling, ed. Paul Fishwick.

The simplest model of behavior is to see behavior as an input/output function. A value or input is given at the beginning of the process, and at some moment there is a value as outcome or output. This model was used to advantage as the simplest model of the behavior of a computer program in computer science, from the start of the subject in the middle of the twentieth century. It was instrumental in the development of (finite state) *automata theory*. In automata theory, a process is modeled as an automaton. An automaton has a number of *states* and a number of *transitions*, going from a state to a state. A transition denotes the execution of an (elementary) action, the basic unit of behavior. Also, there is an initial state (sometimes, more than one) and a number of final states. A behavior is a run, i.e. a path from initial state to final state. An important aspect is when to consider two automata equal, expressed by a notion of equivalence. On automata, the basic notion of equivalence is 'language equivalence', which considers equivalence in terms of behavior, where a behavior is characterized by the set of executions from the initial state to a final state. An algebra that allows equational reasoning about automata is the algebra of regular expressions, see e.g. (Linz 2001).

Later on, this model was found to be lacking in several situations. Basically, what is missing is the notion of *interaction*: during the execution from initial state to final state, a system may interact with another system. This is needed in order to describe parallel or distributed systems, or so-called *reactive* systems. When dealing with interacting systems, the phrase *concurrency theory* is used. Thus, concurrency theory is the theory of interacting, parallel and/or distributed systems. When referring to process algebra, we usually consider it as an approach to concurrency theory, so that a process algebra usually (but not necessarily) has parallel composition as a basic operator.

Thus, a usable definition is that process algebra is the study of the behavior of parallel or distributed systems by algebraic means. It offers means to describe or *specify* such systems, and thus it has means to specify parallel composition. Besides this, it can usually also specify alternative composition (choice) and sequential composition (sequencing). Moreover, it is possible to reason about such systems using algebra, i.e. equational reasoning. By means of this equational reasoning, *verification* becomes possible, i.e. it can be established that a system satisfies a certain property.

What are these basic laws of process algebra? In this chapter, we do not present collections of such laws explicitly. Rather, it is shown how calculations can proceed.

To repeat, it can be said that any mathematical structure with operators of the right number of arguments satisfying the given basic laws is a process algebra. Often, these structures are formulated in terms of *transition systems*, where a transition system has a number of states (including an initial state and a number of final states) and transitions between them. The notion of equivalence studied is usually not language equivalence. Prominent among the equivalences studied is the notion of *bisimulation*. Often, the study of transition systems, ways to define them and equivalences on them are also considered part of process algebra, even in the case no equational theory is present.

1.2 Calculation

One form of calculation is verification by means of automated methods (called *model checking*, see e.g. (Clarke, Grumberg, and Peled 2000)) that traverse all states of a transition system and check that a certain property is true in each state. The drawback is that transition systems grow very large very quickly (in fact, often they become infinite). For instance, a system having 10 interacting components, each of which has 10 states, has a total number of 10 000 000 000 states. It is said that model checking techniques suffer from the *state explosion* problem.

At the other end, reasoning can take place in logic, using a form of deduction. Also here, progress is made, and many *theorem proving* tools exist (Bundy 1999). The drawback here is that finding a proof needs user assistance (as the general problem is undecidable), which requires a lot of knowledge about the system.

Equational reasoning on the basis of an algebraic theory takes the middle ground. On the one hand, the next step in the procedure is usually clear, since it is more rewriting than equational reasoning. Therefore, automation can be done in a straightforward way. On the other hand, representations are compact and allow the presence of parameters, so that an infinite set of instances can be verified at the same time.

1.3 History

Process algebra started in the late seventies of the twentieth century. At that point, the only part of concurrency theory that existed was the theory of Petri nets, as discussed in another chapter in this volume.

The question was raised how to give semantics to programs containing a parallel composition operator. It was found that this was difficult using the semantical methods used at that time. The idea of a behavior as an input/output function needed to be abandoned. A program could still be modeled as an automaton, but the notion of language equivalence was no longer appropriate. This is because the interaction a process has between input and output influences the outcome, disrupting functional behavior. Secondly, the notion of *global* variables needed to be overcome. Using global variables, a state of an automaton used as a model was given as a valuation of the program variables, that is, a state was determined by the values of the variables. The independent execution of parallel processes makes it difficult or impossible to determine the values of global variables at a given moment. It turned out to be simpler to let each process have its own local variables, and to denote exchange of information explicitly.

After some preliminary work by others, three main process algebra theories were developed. These are CCS (Calculus of Communicating Systems) by Robin Milner (Milner 1980; Milner 1989), CSP (Communicating Sequential Processes) by Tony Hoare (Hoare 1985), and ACP (Algebra of Communicating Processes) by Jan Bergstra and Jan Willem Klop, see (Bergstra and Klop 1984; Baeten and Weijland 1990).

Comparing these best-known process algebras CCS, CSP and ACP, we can say there is a considerable amount of work and applications realized in all three of them. In that sense, there seem to be no fundamental differences between the theories with respect to the range of applications. Historically, CCS was the first with a complete theory. Different from the other two, CSP has a least distinguishing equational theory. More than the other two, ACP emphasizes the algebraic aspect: there is an equational theory with a range of semantical models. Also, ACP has a more general communication scheme: in CCS, communication is combined with abstraction, in CSP, there is also a restricted communication scheme.

The language we consider in this chapter is most closely related to the ACP approach. Over the years, other process algebras were developed, and many extensions were realized. Most interesting for this volume is the extension to hybrid systems.

1.4 Hybrid systems

Process algebra started out in computer science, and is especially geared to describing discrete event systems such as computer programs and software systems. With the growing importance of embedded systems, which are software systems that are integrated in the machine or device that they control, it was considered to use process algebra also to model and reason about the controlled physical environment of the software. However, specifications of physical systems not only require discrete-event models, but also differential algebraic equations, leading to hybrid models.

In recent years, several attempts were made to incorporate such aspects into process algebra. In this chapter, we report on one of these, based on the χ language. Other hybrid process algebras are HyPA (Cuijpers and Reniers 2005), process algebra for hybrid systems $\mathrm{ACP}^{\mathrm{srt}}_{\mathrm{hs}}$ (Bergstra and Middelburg 2005), and the ϕ-Calculus (Rounds and Song 2003). The history of the χ formalism dates back quite some time. It was originally mainly used as a modeling and simulation language for discrete-event systems. The first simulator (Naumoski and Alberts 1998) was successfully applied to a large number of industrial cases, such as integrated circuit manufacturing plants, breweries, and process industry plants (van Beek, van den Ham, and Rooda 2002). Later, the hybrid language and simulator were developed (Fábián 1999; van Beek and Rooda 2000). Recently, the χ language has been completely redesigned. The result is a hybrid process algebra with a formal semantics as defined in (van Beek, Man, Reniers, Rooda, and Schiffelers 2006). This chapter informally defines the most important elements of the syntax and semantics of the χ process algebra. It also extends the formal definitions of (van Beek, Man, Reniers, Rooda, and Schiffelers 2006) with a more user friendly syntax, including the specification of data types.

2 Syntax and informal semantics of the χ process algebra

2.1 Model syntax

In this section, the syntax of χ models is defined using a Backes-Naur (BNF) like notation. The symbol | defines choice, notation $\{Z\}^*$ denotes a sequence of zero or more Z's, and notation $\{Z\}^0$ defines Z as being optional. A χ model is of the following form:

$$\chi_{\text{model}} ::= \ \mathsf{model}\ id(D_{\text{m}}) = [\![\ D :: \ p\ ]\!]$$

where *id* is an identifier that represents the name of the model, and D_{m} denotes the model parameters as defined below. The model parameter declaration may be also be empty. Furthermore, D denotes the declaration of variables and/or channels of the model. This type of declaration is also used to declare the local variables and channels of scope operators. Finally, p denotes a statement, also known as process term. The scope operator and statement p are both defined in Section 2.2. To simplify the syntax definitions, we assume the declaration part D not to be empty. The syntax of the declarations D_{m} and D is:

$$\begin{array}{rlll}
D_{\text{m}} ::= & \mathsf{val}\ \ S\ \{, S\}^* \ \mid\ D_{\text{m}}, D_{\text{m}} & & \text{value parameter declaration} \\[4pt]
D ::= & \mathsf{chan}\ \ S\ \{, S\}^* & & \text{channel declaration} \\
\mid & (\mathsf{var}\ \mid \mathsf{cont}\ \mid \mathsf{alg}\ )\ \ IS\ \{, IS\}^* & & \text{variable declaration} \\
\mid & D, D & &
\end{array}$$

$$\begin{array}{rll}
S ::= & id\ \{, id\}^* : t & \text{declaration without initialization} \\[4pt]
IS ::= & id\ \{, id\}^* : t = e\ \mid\ S & \text{declaration with optional initialization}
\end{array}$$

Here, t denotes the type of a variable or channel, e denotes an initialization expression, and *id* denotes an identifier. An executable model instantiation for a model declared as $\mathsf{model}\ M(\mathsf{val}\ x_1 : t_1, \ldots, x_n : t_n)$ is obtained by $M(c_1, \ldots, c_n)$, where c_i denotes a value for the corresponding model parameter x_i. The following items can be declared in D:

- Channels, such as in $\mathsf{chan}\ h : \mathsf{real},\ \mathit{close} : \text{void}$, which declares a communication channel h, that communicates values of type real, and a synchronization channel *close* (no data exchange).
- Discrete variables, such as in $\mathsf{var}\ k, n : \mathsf{int},\ v_{\text{set}} : \mathsf{real} = 1.0$. This declares two uninitialized variables k, n of type int (integer), and a variable v_{set} that has an initial value 1.0. The values of discrete variables remain constant when model time progresses.
- Continuous variables, such as in $\mathsf{cont}\ x : \mathsf{real} = 1.0$. Continuous variables are the only variables for which dotted variables (derivatives) can be used in models. Therefore, the declaration $\mathsf{cont}\ x : \mathsf{real} = 1.0$ implies that x and its dotted version $\dot{x}$, can both be used in the model. The values of continuous variables may change according to a continuous function of time when model time progresses. The values of continuous variables are further restricted by delay predicates (defined in the next section), that may occur in the form of differential algebraic equations.
- Algebraic variables, such as in $\mathsf{alg}\ y, z : \mathsf{real}$. These variables behave in a similar way as continuous variables. The differences are that algebraic variables may change according to a discontinuous function of time, and algebraic variables are not allowed to occur as dotted variables.

Besides the variables mentioned in the model defined above, the existence of the predefined reserved global variable time which denotes the current time, the value of which is initially zero, is assumed. This variable cannot be declared. It can only be used in expressions in statements p.

2.2 Statement syntax

Statements can be divided in two classes: the atomic statements, that represent the smallest statement units; and the compound statements, that are constructed from one or more (atomic) statements by means of operators. The syntax of the atomic χ statements, is as follows:

p_{atom}	$::=$	skip			non-delayable internal action
	$\mid$	$\mathbf{x} := \mathbf{e}$			non-delayable (multi-)assignment
	$\mid$	$\{\mathbf{x}\} : r \gg l_{\text{a}}$			non-delayable action predicate
	$\mid$	$h\,??\,\mathbf{x}$	$\mid$	$h??$	non-delayable receive
	$\mid$	$h\,!!\,\mathbf{e}$	$\mid$	$h!!$	non-delayable send
	$\mid$	[skip]			delayable internal action
	$\mid$	$[\mathbf{x} := \mathbf{e}]$			delayable (multi-)assignment
	$\mid$	$[\{\mathbf{x}\} : r \gg l_{\text{a}}]$			delayable action predicate
	$\mid$	$h\,!\,\mathbf{e}$	$\mid$	$h!$	delayable send
	$\mid$	$h\,?\,\mathbf{x}$	$\mid$	$h?$	delayable receive
	$\mid$	Δd			delay
	$\mid$	u			delay predicate,

where $\mathbf{x}$ and $\mathbf{e}$ denote comma separated variables $x_1, \ldots, x_n$ and expressions $e_1, \ldots, e_n$, respectively, for $n \geq 1$, r denotes a predicate (boolean condition) as defined in Section 2.4.1, l_{a} denotes an action label, h denotes a channel, and d denotes an expression of type real. Delay predicate u denotes a predicate over variables (including the variable time) and dotted continuous variables (derivatives). Delay predicates may occur in the form of differential algebraic equations, such as $\dot{x} = y,\ y = n$, or in the form of a constraint or invariant, such as $x \geq 1$. The comma in delay predicates denotes conjunction. E.g. u_1, u_2 means $u_1 \wedge u_2$. Also, both $e_1 \leq \dot{x} \leq e_2$ and $\dot{x} \in [e_1, e_2]$

can be used instead of $e_1 \leq \dot{x}$, $\dot{x} \leq e_2$, and likewise for strict inequalities and open intervals. Note that the non-delayable send statements $h \,!\, \mathbf{e}$ and $h!$ can also be written as $[h \,!!\, \mathbf{e}]$ and $[h!!]$, and likewise for the delayable receive statements.

The syntax of the compound χ statements is as follows:

$p ::=$		p_{atom}	atomic
	$\mid$	$p;\ p$	sequential composition
	$\mid$	$b \to p$	guard operator
	$\mid$	$p \ [\!]\ p$	alternative composition
	$\mid$	$p \parallel p$	parallel composition
	$\mid$	$*p$	loop statement
	$\mid$	$b \overset{*}{\to} p$	while statement
	$\mid$	$[\![\ D :: \ p\ ]\!]$	variable and channel scope operator
	$\mid$	$id(\mathbf{e})$	process instantiation
	$\mid$	p_{R}	recursion scope operator,

where b denotes a predicate over variables. To simplify the semantics of χ models, the use of continuous or algebraic variables in guards is restricted. In particular, the guards b in $b \to u$, $b \to \Delta d$, and $b \to [\![\ D :: \ p\ ]\!]$ are not allowed to change while delaying, which is ensured by using only discrete variables in such guards. For delay predicate u, instead of $b \to u$, where b contains continuous variables, $b \Rightarrow u$ can be used. Here $\Rightarrow$ denotes logical implication, and $b \Rightarrow u$ is therefore also a (delay) predicate. Note that in the formal semantics of hybrid χ as defined in (van Beek, Man, Reniers, Rooda, and Schiffelers 2006) there are no restrictions on the use of continuous or algebraic variables in the guards.

The operators are listed in descending order of their binding strength as follows:

$$\{*,\ \overset{*}{\to}\ ,\ \to\},\ ;\ ,\{\parallel\ ,\ [\!]\}.$$

The operators inside the braces have equal binding strength. For example, $x := 1;\ y := x\ [\!]\ x := 2;\ y := 2x$ means $(x := 1;\ y := x)\ [\!]\ (x := 2;\ y := 2x)$. Parentheses may be used to group statements. To avoid confusion, parenthesis are obligatory when alternative composition and parallel composition are used together. E.g. $p\ [\!]\ q \parallel r$ is not allowed and should either be written as $(p\ [\!]\ q) \parallel r$, or as $p\ [\!]\ (q \parallel r)$.

The recursion scope operator statement p_{R} may appear in two forms:

$$\begin{aligned} p_{\text{R}} ::= \ & [\![\ R\ \{, R\}^* :: \ X\ ]\!] \\ \mid \ & [\![\ R\ \{, R\}^* :: \ p^+\ ]\!], \end{aligned}$$

where X denotes a recursion variable, and recursion definition R is defined as:

$$R ::= \ \{\mathsf{mode}\}^0\ X = (p^+),$$

where statements p^+ consist of statements p to which recursion variables X are added:

$$p^+ ::= \ p \ \mid \ p; X \ \mid \ p^+\ [\!]\ p^+ \ \mid \ p; p^+$$

The syntax enforces any recursion variable X to occur only at the end of a sequential composition. An additional restriction is that each recursion scope operator must be ‘complete’. This means that in the two forms of the recursion scope operator

$$[\![\ \mathsf{mode}\ X_1 = (p_1^+),\ \ldots, \mathsf{mode}\ X_n = (p_n^+) :: \ X_k\ ]\!] \quad \text{and}$$

$$[\![\ \mathsf{mode}\ X_1 = (p_1^+),\ \ldots, \mathsf{mode}\ X_n = (p_n^+) :: \ p_{n+1}^+ \ ]\!],$$

all occurrences of free recursion variables in p_i^+ $(1 \leq i \leq n+1)$ must be defined in the recursion scope operator itself: $\cup_{j=1}^{n+1}\text{freervar}(p_j^+) \subseteq \{X_1,\ \ldots,\ X_n\}$, where freervar$(p_j^+)$ is a function that returns the set of free (unbound) occurrences of recursion variables in p_j^+. Furthermore, the recursion variables $X_1, \ldots, X_n$ may occur only at the 'top level' in p_i^+, that is, not nested in other scope operators. These restrictions enforce structured use of recursion: only one recursion variable X_i with corresponding statement p_i^+ can be executed at the same time, the first statement to be executed is p_{n+1}^+, and termination of any of the statements p_i terminates the scope operator itself. This structured use of recursion simplifies analysis of χ models, it simplifies the translation to the normal form as discussed in Section 3, and it simplifies tool support for χ.

Although recursion variables cannot be placed in parallel directly, two recursion scope operators can occur in parallel, as in:

$$[\![\ \mathsf{mode}\ X_1 = (p_1) :: \ X_1 \ ]\!] \parallel [\![\ \mathsf{mode}\ X_2 = (p_2) :: \ X_2 \ ]\!],$$

where the two recursion variables X_1 and X_2 can of course have the same name, without changing the meaning of the model.

2.3 Semantical framework

In this chapter, the meaning (semantics) of a χ model is informally defined in terms of delay behavior and action behavior, based on the formal semantics as presented in (van Beek, Man, Reniers, Rooda, and Schiffelers 2006). Delay behavior involves passing of time, where the semantics defines for each variable how its value changes as a function of time. Action behavior is instantaneous: time does not progress, and the semantics defines for each variable the relation between its value before and after the action.

Atomic statements can be disabled or enabled. Actions and delays are done by *enabled atomic* statements, with one exception only: an enabled guarded statement $b \to p$ with a guard that is false can do any delay. Atomic statements terminate by doing an action. They never terminate by doing a delay. A statement that terminates becomes disabled by doing so.

Compound statements combine (sub-)statements by means of operators. The operator defines the relation between enabling, disabling and termination of the compound statement and its sub-statements. Enabling or disabling a compound statement is defined in terms of enabling or disabling its sub-statements. Enabling a compound statement implies enabling one or more of its sub-statements. E.g. enabling a sequential composition $p_1; \ldots; p_n$ implies enabling the first statement p_1, whereas enabling a parallel composition $p_1 \parallel \ldots \parallel p_n$ implies enabling all statements $p_1 \ldots p_n$.

Execution of a χ model M, defined as $\mathsf{model}\ M(D_0) = [\![\ D_1 :: \ p_0 \ ]\!]$, takes place by executing a sequence of delays and actions in the following way:

- At the start, statement p_0 is enabled.
- Any enabled skip statement, assignment statement or action predicate (delayable or non-delayable) can do an action.
- An enabled pair of a send and a receive statement on the same channel that are placed in parallel can simultaneously do a send and a receive action, followed by joint termination. The result, in terms of values of variables, of simultaneous execution of a send statement (h !! $\mathbf{e}$ or h ! $\mathbf{e}$) and a receive statement (h ?? $\mathbf{x}$ or h ? $\mathbf{x}$) is comparable to the (distributed) execution

of a multi-assignment $\mathbf{x} := \mathbf{e}$. E.g. execution of the communication action in $h\,!\,1 \parallel h\,?\,x$ is comparable to execution of the assignment $x := 1$.

- The model can do delays only when and for as long as:
  - All enabled statements can delay. The delayable versions of the skip statement, assignment, action predicate, and send and receive statements can always delay (the non-delayable versions can never delay). A delay statement Δd can delay for as long as its internal timer is not expired (see Section 2.4.3), and the set of all enabled delay predicates can delay for as long as they have a solution. Such a solution defines the values of the variables as a function of time for the period of the delay.

    Note that the set of enabled statements may change while delaying. The reason for this is the guarded statement $b \rightarrow p$, because the value of the guard can change while delaying, due to changes in the values of the continuous or algebraic variables.
  - No parallel pair of a send and a receive statement on the same channel is enabled or becomes enabled. This is because, by default, channels in χ are urgent: communication or synchronization cannot be postponed by delaying.
- When different actions and/or delays are possible, any of these can be chosen. This is referred to as nondeterministic choice. Note that delays may always be shorter than the maximum possible length.

The values of the discrete and continuous variables are stored in memory. The values of the algebraic variables are not stored. This means that the starting point of the trajectory of the discrete and continuous variables equals their last value stored in memory. The starting point of the trajectory of the algebraic variables can be any value that is allowed by the enabled equations.

In models of physical systems, the delay behavior of the continuous and algebraic variables is usually uniquely determined: there is usually only one solution of the set of enabled differential algebraic equations. Multiple delays / solutions can be caused by under-specified systems of equations, where there are less equations than variables, or by delay predicates that allow multiple solutions, such as 'true' or $\dot{x} \in [0, 1]$.

The action behavior of the discrete, continuous and algebraic variables is as follows:

- The discrete and continuous variables do not change as a result of actions unless the change is explicitly specified, for example by means of an assignment, or by receiving a value via a channel.
- The algebraic variables do not have a memory. Therefore, their value can in principle change arbitrarily in actions. In most models, their value is defined by delay predicates.

2.4 Semantics of atomic statements

2.4.1 Skip, multi-assignment and action predicate

An enabled skip statement can do an internal action, and then terminates.

An enabled *multi-assignment* statement $\mathbf{x}_n := \mathbf{e}_n$ for $n \geq 1$ can do an internal action that changes the values of the variables $x_1, \ldots, x_n$ in one step to the values of expressions $e_1, \ldots, e_n$, respectively, and then terminates. For $n = 1$, this gives a normal assignment $x := e$.

The action predicate $\{\mathbf{x}\} : r \gg l_\mathrm{a}$ is a kind of generalized assignment statement, consisting of a set of variables $\{\mathbf{x}\}$, a predicate r over variables and '${}^-$' superscripted variables, and an action label l_a. The values of the variables of set $\{\mathbf{x}\}$ are allowed to change so that their new values satisfy the predicate r. A '$^-$' superscripted variable, such as x^- refers to the value of the variable immediately before execution of the action predicate. E.g. a multi assignment $x, y := y, x$, that swaps the values of x and y, can be written as action predicate: $\{x, y\} : x = y^-, y = x^- \gg \tau$, where τ denotes the predefined internal action.

2.4.2 Delay predicate

An enabled *delay predicate u* can perform delays but no actions. Delay predicates restrict the allowed trajectories of the variables while delaying in such a way that at each time point during the delay the delay predicate holds (its value must be true), when all variables and dotted variables in the predicate are replaced by their current value.

Delay predicates also restrict the action behavior of χ models, because the enabled delay predicate must also hold before and after each action. In fact, the enabled delay predicates of a χ model must hold at all times. This is referred to as the 'consistent equation semantics'.

The relation between the trajectory of a continuous variable x and the trajectory of its 'derivative' $\dot{x}$ is given by the Caratheodory solution concept: $x(t) = x(0) + \int_0^t \dot{x}(s)ds$. This allows a non-smooth (but continuous) trajectory for a differential variable in the case that the trajectory of its 'derivative' is non-smooth or even discontinuous, as in, for example, model $M() = [\![\ \mathsf{cont}\ y : \mathsf{real} = 0.0 :: \dot{y} = \mathsf{step}(\mathsf{time} - 1)\ ]\!]$, where $\mathsf{step}(x)$ equals 0 for $x \leq 0$ and 1 for $x > 0$.

2.4.3 Delay statement

A delay statement Δd behaves as a local timer. The timer can either delay, or it can terminate by means of an action. It can be activated in two ways:

- If the value c of expression d is bigger than zero when the timer is enabled for the very first time, or when it is re-enabled after having been disabled, the timer can start running (delaying) from the value c. After having delayed for a total of c time-units, the timer has expired. An expired timer cannot delay anymore; it can only terminate by means of an action.
- If the value c of expression d is zero when the timer is enabled for the very first time, or when it is re-enabled after having been disabled, the timer can immediately expire and terminate by means of an action.

The total period of time c, that must pass before the timer expires, can be split up into separated delays, that may be interleaved by actions of other statements in a parallel context, as long as the timer (delay statement) stays enabled. When the timer / delay statement is disabled, re-enabling it causes re-evaluation of expression d when the timer starts running again. Note that the value c can change each time the timer is re-enabled. E.g. in $*(h\,?\,d;\ \Delta d) \parallel *(h\,!\,1;\ h\,!\,2)$, the first delay of the timer is 1, the second delay is 2, and then the cycle is repeated.

2.5 Semantics of compound statements

2.5.1 Sequential composition

In a *sequential composition* $p_1; \ldots; p_n$ $(n \geq 1)$, only one statement p_i, $1 \leq i \leq n$, can be enabled at the same time. Enabling a sequential composition $p_1; \ldots; p_n$ implies enabling its first statement p_1. When statement p_i $(1 \leq i \leq n - 1)$ terminates (and is therefore also disabled), the next statement p_{i+1} becomes enabled. The sequential composition terminates upon termination of its last statement p_n.

2.5.2 Guard operator

Enabling of a guarded statement enables its guard b. Behavior of a guarded statement $b \rightarrow p$ depends on the value of the guard b:

- Statement p is enabled while the value of the guard is true. Execution of the first *action* by p disables the guard. Thus, after this first action, the value of the guard becomes irrelevant.
- Statement p is disabled while the value of the guard is false. The guarded statement $b \rightarrow p$ can, in principle, do any delay while the value of the guard is false; only at the start point and end point of such a delay, the value of the guard may be true.

When a guarded statement occurs in parallel with another statement, as in $q \parallel b \rightarrow p$, the value of the guard can change due to actions of statement q, which may cause statement p to change from being disabled to enabled or vice versa. E.g. $b := \text{false}; (\Delta 1; b := \text{true} \parallel b \rightarrow \text{skip})$

When in $q \parallel b \rightarrow p$, the guard b contains continuous or algebraic variables, and q contains one or more enabled delay predicates, the value of the guard may change during a delay, causing statement p to change from being disabled to enabled or vice versa. E.g. $\dot{x} = 1 \parallel x \geq 1 \rightarrow x := 0$.

2.5.3 Alternative composition

Enabling $p_1 \,[\!]\, \ldots \,[\!]\, p_n$ enables the statements $p_1, \ldots, p_n$. Execution of an action by any one of the statements $p_1 \ldots p_n$ disables the other statements. In this way, execution of the first action makes a choice. When one of the statements $p_1, \ldots, p_n$ terminates, the alternative composition $p_1 \,[\!]\, \ldots \,[\!]\, p_n$ also terminates.

2.5.4 Parallelism

Enabling $p_1 \parallel \ldots \parallel p_n$ enables the statements $p_1, \ldots, p_n$. When a statement p_i, $1 \leq i \leq n$, executes an action, the other statements remain enabled. The parallel composition $p_1 \parallel \ldots \parallel p_n$ terminates when the statements $p_1, \ldots, p_n$ have all terminated.

Informally, we often refer to the statements $p_1, \ldots, p_n$ occurring in $p_1 \parallel \ldots \parallel p_n$ as parallel processes. Parallel processes interact by means of shared variables or by means of synchronous point-to-point communication or synchronization via a channel. Communication in χ is the sending of values of one or more expressions by one parallel process via a channel to another parallel process, where the received values are stored in variables. In case no values are sent and received, we refer to synchronization instead of communication.

2.5.5 Loop and while statement

Loop statement $*p$ represents the infinite repetition of statement p. When $*p$ is enabled, p is enabled. Termination of p results in re-enabling of p.

The while statement $b \overset{*}{\rightarrow} p$ can be interpreted as “while b do p”. Enabling of $b \overset{*}{\rightarrow} p$ when b is true enables p (after an internal action), and enabling of $b \overset{*}{\rightarrow} p$ when b is false, leads to immediate termination (by means of an internal action).

2.5.6 Variable and channel scope operator

A variable and channel scope operator may introduce new variables and new channels. Enabling of a variable and channel scope statement $[\![\ D :: \ p\ ]\!]$, where the local declaration part D introduces new variables and or channels according to the specification of D in Section 2.1, performs the variable initializations specified in D and enables statement p. Termination of p terminates the scope statement $[\![\ D :: \ p\ ]\!]$. Any occurrence of a variable or channel in p that is declared in D refers to that local variable or channel and not to any more global declaration of the variable or channel with the same name, if such a more global declaration should exist.

2.5.7 Recursion scope operator and recursion variable

Statement X denotes a recursion variable (identifier) that is defined and used in a recursion scope operator statement of the form $[\![\ X_1 = (p_1^+),\ \ldots, X_n = (p_n^+) :: \ X_i\ ]\!]$ or $[\![\ X_1 = (p_1^+),\ \ldots, X_n = (p_n^+) :: \ p_{n+1}^+\ ]\!]$, where the keyword mode may be added.

The meaning of recursion scope operators satisfying the restrictions defined in Section 2.2, is as follows. Enabling the recursion scope operator $[\![\ X_1 = (p_1^+),\ \ldots, X_n = (p_n^+) :: \ X_i\ ]\!]$ or $[\![\ X_1 = (p_1^+),\ \ldots, X_n = (p_n^+) :: \ p_{n+1}^+\ ]\!]$, enables the statement X_i or p_{n+1}^+, respectively. When a recursion variable X_i is enabled (or disabled), its defining statement p_i is enabled (or disabled) instead. When a defining statement p_i $(1 \leq i \leq n)$ terminates, the recursion scope operator terminates.

2.5.8 Process definition and instantiation

To simplify the explanation, process instantiation $id(\mathbf{e})$ is rewritten in a more specific form $id(\mathbf{x}_n, \mathbf{z}_m, \mathbf{h}_l, \mathbf{e}_k)$, where id denotes a process name, $\mathbf{x}_n$ denotes the continuous variables $x_1, \ldots, x_n$, $\mathbf{z}_m$ denotes the algebraic variables $z_1, \ldots, z_m$, $\mathbf{h}_l$ denotes the channels $h_1, \ldots, h_l$, and $\mathbf{e}_k$ denotes the expressions $e_1, \ldots, e_k$. Process instantiation enables (re)-use of a process definition. A process definition is specified once, but it can be instantiated many times, usually with different parameters.

The meaning of process instantiation

$$id(\mathbf{x}_n, \mathbf{z}_m, \mathbf{h}_l, \mathbf{e}_k)$$

with corresponding process definition

$$\mathsf{proc}\ id(\mathsf{cont}\ \mathbf{x}'_n : \mathbf{t}_\mathrm{c},\ \mathsf{alg}\ \mathbf{z}'_m : \mathbf{t}_\mathrm{a},\ \mathsf{chan}\ \mathbf{h}'_l\{!|?\}^0 : \mathbf{t}_\mathrm{h},\ \mathsf{val}\ \mathbf{v}_k : \mathbf{t}_\mathrm{v}) = [\![\ D :: \ p\ ]\!]$$

is obtained by the syntactical substitution of the process instantiation by

$$[\![\ D, \mathsf{var}\ \mathbf{v}_k : \mathbf{t}_\mathrm{v} = \mathbf{w}_k :: \ p\ ]\!]\ [\mathbf{x}_n, \mathbf{z}_m, \mathbf{h}_l, \mathbf{e}_k / \mathbf{x}'_n, \mathbf{z}'_m, \mathbf{h}'_l, \mathbf{w}_k],$$

where notation $\mathbf{t}_{\ldots}$ denotes a data type, and $\mathbf{h}'_l\{!|?\}^0$ denotes the channels $h'_1\{!|?\}^0, \ldots, h'_l\{!|?\}^0$, where each channel is optionally postfixed with ! or ?. The meaning of $h!$ or $h?$ in the formal parameter list is that the use of channel h in p is limited to sending ($h!$.. or $h!!$..) or receiving ($h?$.. or $h??$..), respectively. Notation $r[b_1, \ldots, b_i / a_1, \ldots, a_i]$ denotes the statement obtained from r by syntactic substitution of the free occurrences of the variables/channels $a_1, \ldots, a_i$ in r by $b_1, \ldots, b_i$, respectively. Free variables (or channels) in r are variables (or channels) that are not declared in r.

The substitution replaces the process instantiation $id(\mathbf{x}_n, \mathbf{z}_m, \mathbf{h}_l, \mathbf{e}_k)$ by the process body $[\![\ D :: \ p\ ]\!]$, whereafter the value parameters $\mathbf{v}_k$ are added to the local declarations D as discrete variables that are initialized to the values of the new variables $\mathbf{w}_k$. Finally, the (free) variables $\mathbf{x}'_n$, $\mathbf{z}'_m$, $\mathbf{w}_k$ and the (free) channels $\mathbf{h}'_l$ occurring in the body are substituted by $\mathbf{x}_n$, $\mathbf{z}_m$, $\mathbf{e}_k$, $\mathbf{h}_l$, respectively. If substitution would cause new bindings, the local variable or local channel that a variable or channel from $\mathbf{x}_n$, $\mathbf{z}_m$, $\mathbf{e}_k$, or $\mathbf{h}_l$ would become bound to, is renamed into a fresh variable or fresh channel before the substitution takes place. In Section 3.3, substitution of process instantiations is illustrated by means of an example.

2.6 Assembly line example

An assembly process A assembles three different parts that are supplied by three suppliers G. The order at which the parts are supplied is unknown, but each part should be received by the assembly process as soon as possible. When all three parts have been received, assembly may start. Assembly takes t_A units of time. When the products have been assembled, they are sent to an exit process E. Figure 1 shows the iconic model of the assembly line, which is modeled as a discrete-event system. For the χ model of the assembly line, first two types are declared. The type

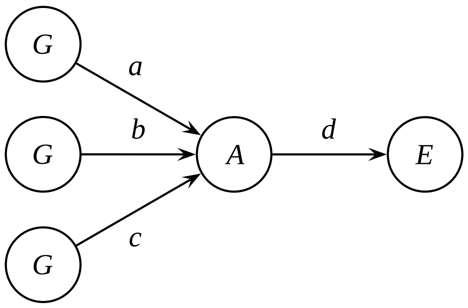

Figure 1: Iconic model of an assembly line.

'part', representing a part as a natural number, and the type 'assy', representing an assembled unit as a 3-tuple of parts:

$$\begin{array}{l} \mathsf{type}\ \mathrm{part} = \mathsf{nat} \\ ,\quad \mathrm{assy} = (\mathrm{part}, \mathrm{part}, \mathrm{part}) \end{array}$$

The χ model consists of parallel instantiations of the three generator processes G, the assembly process A and the exit process E:

$$\begin{array}{l} \mathsf{model}\ \mathit{AssemblyLine}(\mathsf{val}\ t_0, t_1, t_2 : \mathsf{real},\ t_\mathrm{A} : \mathsf{real}) = \\ [\![\ \mathsf{chan}\ a, b, c : \mathrm{part},\ d : \mathrm{assy} \\ ::\ G(a, 0, t_0) \parallel G(b, 1, t_1) \parallel G(c, 2, t_2) \parallel A(a, b, c, d, t_\mathrm{A}) \parallel E(d) \\ ]\!] \end{array}$$

Each generator G sends a part n every t time units:

$$\mathsf{proc}\ G(\mathsf{chan}\ a! : \mathrm{part},\ \mathsf{val}\ n : \mathsf{nat},\ t : \mathsf{real}) = [\![\ *(\ a!n;\ \Delta t\ )]\!]$$

The assembly process receives the parts by means of the parallel composition $(a\,?\,x \parallel b\,?\,y \parallel c\,?\,z)$. This ensures that each part is received as soon as possible. The parallel composition terminates when all parts have been received.

$$\begin{array}{l}\text{proc } A(\text{chan } a?, b?, c? : \text{part},\ d! : \text{assy},\ \text{val } t : \text{real}) = \\ [\![\text{ var } x, y, z : \text{part} \\ :: *(\ (\ a\,?\,x \parallel b\,?\,y \parallel c\,?\,z\ )\ ;\ \Delta t;\ d\,!(x, y, z)\ ) \\ ]\!] \end{array}$$

The exit process is simply:

$$\text{proc } E(\text{chan } a? : \text{assy}) = [\![\text{ var } x : \text{assy} :: *\, a\,?\,x\ ]\!]$$

3 Algebraic reasoning and verification

3.1 Introduction

The χ process algebra has strong support for modular composition by allowing unrestricted combination of operators such as sequential and parallel composition, by providing statements for scoping, by providing process definition and instantiation, and by providing different interaction mechanisms, namely synchronous communication and shared variables.

The fact that the χ process algebra is such a rich language potentially complicates the development of tools for χ, since the implementations have to deal with all possible combinations of the χ atomic statements and the operators that are defined on them. This is where the process algebraic approach of equational reasoning, that allows rewriting models to a simpler form, is essential.

To illustrate the required implementation efforts, consider the following implementations that are developed: a Python implementation for rapid prototyping; a C implementation for fast model execution; and an implementation based on the MATLAB Simulink S-functions (The MathWorks, Inc 2005), where a χ model is translated to an S-function block. Furthermore, there is an implementation for real-time control (Hofkamp 2001). In (Bortnik, Trčka, Wijs, Luttik, van de Mortel-Fronczak, Baeten, Fokkink, and Rooda 2005) it has been shown that different model checkers each have their own strengths and weaknesses. Therefore, for verification, translations to several tools are defined. In particular, for hybrid models a translation to the hybrid I/O automaton based PHAver (Frehse 2005) model checker is defined. For timed models the following translations are defined: (1) a translation to the action-based process algebra μCRL (Groote 1997), used as input language for the verification tool CADP (Fernandez, Garavel, Kerbrat, Mounier, Mateescu, and Sighireanu 1996); (2) a translation to PROMELA, a state-based, imperative language, used as input language for the verification tool SPIN (Holzmann 2003); and (3) a translation to the timed automaton based input language of the UPPAAL (Larsen, Pettersson, and Yi 1997) verification tool. In future, for verification of hybrid models, additional translations may be considered to tools such as HYTECH (Alur, Henzinger, and Ho 1996), or one of the many other hybrid model checkers.

Instead of defining the implementations mentioned above on the full χ language as defined in Section 2, the process algebraic approach of equational reasoning makes it possible to transform χ models in a series of steps to a (much simpler) normal form, and to define the implementations on the normal form. The original χ model and its normal form are bisimilar, which ensures that relevant model properties are preserved. The normal form has strong syntactical restrictions, no parallel composition operator, and is quite similar to a hybrid automaton. Currently, correctness proofs are developed, and in the near future, implementations will be redesigned based on the normal form.

The steps to the normal form are as follows. First of all, the process instantiations are eliminated, by replacing them by their defining bodies, and replacing the formal parameters by actual variables. Next, parallel composition is eliminated by using laws of process algebra, in particular a so-called *expansion law* (not given here). An example of a process algebra law in χ specifying that the guard distributes over alternative composition is $b \rightarrow (p \;[]\; q) \;=\; b \rightarrow p \;[]\; b \rightarrow q$. Finally, the normal form may be simplified further, taking advantage of the fact that it no longer contains parallel composition. Note that it is possible to construct models for which the normal form cannot be (easily) generated. These exceptions are not discussed in this chapter, since they do not restrict translation to the normal form for practical purposes.

The syntax for the normal form in χ is given by a model with on the outer level a global variable and channel declaration D, as defined in Section 2.1, on the inner level a local variable and channel declaration D, and one recursion scope operator statement. To simplify the syntax definitions, we assume the declaration parts not to be empty.

$$\chi_{\text{norm}} ::= \;\; \mathsf{model}\; id(D_{\text{m}}) = [\![\; D :: [\![\; D :: [\![\; R_{\text{norm}} :: X \;]\!]]\!]]\!],$$

with recursion definitions R_{norm} according to the normal form defined as:

$R_{\text{norm}} ::=$	$X = (p_{\text{norm}})$	recursion definition
$\mid$	$X = (u \curvearrowright p_{\text{norm}})$	recursion definition with initialization
$\mid$	R, R	multiple recursion definitions

The signal emission operator $u \curvearrowright p$, defined in (van Beek, Man, Reniers, Rooda, and Schiffelers 2006), is required only when a variable scope operator is used that initializes algebraic variables. The global and local variable and channel declarations differ only with respect to the visibility of the declared variables and channels in the transition system. Globally declared variables and channels are visible, locally declared variables and channels are not, due to abstraction.

The normalized statements p_{norm}, used to define the recursion variables X, may consist of undelayable normalized atomic statements p_{na}. Such an normalized atomic statement may be prefixed by a guard b, and/or it may be made delayable (e.g. $b \rightarrow p_{\text{na}}$ and $[p_{\text{na}}]$). Delay predicates u, that may be guarded, are also allowed. Sequential composition is allowed only in the form of such (guarded, and/or delayable) atomic statements followed by a recursion variable. Finally, all of these statements may be part of alternative composition:

$p_{\text{norm}} ::= \; p_{\text{nga}}$		(guarded) atomic action
$\mid\; u \;\mid\; b \rightarrow u$		(guarded) delay predicate
$\mid\; p_{\text{nga}};\; X$		atomic action followed by recursion variable
$\mid\; p_{\text{norm}} \;[]\; p_{\text{norm}}$		alternative composition,

where the normalized guarded atomic action statements p_{nga} are defined by:

$p_{\text{nga}} ::= \; p_{\text{na}}$	non-delayable atomic action statement
$\mid\; b \rightarrow p_{\text{na}}$	guarded non-delayable atomic action statement
$\mid\; [p_{\text{na}}]$	delayable atomic action statement
$\mid\; b \rightarrow [p_{\text{na}}]$	guarded delayable atomic action statement

and the normalized atomic action statements p_{na} are defined by:

$p_{\text{na}} ::=$	$\{\mathbf{x}\} : r \gg l_{\text{a}}$	action predicate
$\mid$	$\{\mathbf{x}\} : r \gg h$	synchronization via channel h
$\mid$	$\{\mathbf{x}\} : r \gg h(\mathbf{e})$	communication via channel h

In the bottle filling example discussed in the following sections, the skip statement is not rewritten to $\{\} : \text{true} \gg \tau$ in the normalized χ models in Section 3.4, for better readability. Also, statement h is used in these models as an abbreviation of $\{\} : \text{true} \gg h$. The synchronization statement $\{\mathbf{x}\} : r \gg h$ and communication statement $\{\mathbf{x}\} : r \gg h(\mathbf{e})$ are required because of the fact that there is no parallel composition in the normalized form. The parallel composition $h!! \parallel h??$ is normalized to $\{\} : \text{true} \gg h$, and $h \,!!\, \mathbf{e} \parallel h \,??\, \mathbf{x}$ is normalized to $\{\mathbf{x}\} : \mathbf{x} = \mathbf{e}^- \gg h(\mathbf{e})$. The statement $\{\} : \text{true} \gg h$ is comparable to the skip statement, and the statement $\{\mathbf{x}\} : \mathbf{x} = \mathbf{e}^- \gg h(\mathbf{e})$ is comparable to the multi-assignment statement $\mathbf{x} := \mathbf{e}$. The effect on the values of the variables is the same. There is only a small difference with respect to the occurrence of channel h in the transition system. As an example consider the statement $(h \,!!\, 0 \parallel h \,??\, x);\ \Delta 1$ which is first rewritten as $(h \,!!\, 0 \parallel h \,??\, x);\ [\![\ \mathsf{var}\ t : \mathsf{real} = \mathsf{time} + 1 :: \mathsf{time} \geq t \to \mathsf{skip}\ ]\!]$ and then normalized to

$$
\begin{array}{l}
[\![\ \mathsf{var}\ t : \mathsf{real} \\
::\ [\![\ X_0 = (\ \{x, t\} : x = 0, t = \mathsf{time} + 1 \gg h(0);\ X_1\ ) \\
\quad ,\ X_1 = (\ \mathsf{time} \geq t \to \mathsf{skip}\ ) \\
\quad ::\ X_0 \\
\quad]\!] \\
]\!]
\end{array}
$$

The normal form makes it easy to analyze system behavior and it simplifies tool implementations in the following way. When a model is defined as

$$
\begin{array}{l}
\mathsf{model}\ M(\mathsf{val}\ \mathbf{x} : \mathbf{t}) = \\
[\![\ D_0 \\
::\ [\![\ D_1 \\
\quad ::\ [\![\ X_1 = (p_{\text{norm}_1}), \ldots, X_n = (p_{\text{norm}_n}) :: X_i\ ]\!] \\
\quad]\!] \\
]\!],
\end{array}
$$

$M(\mathbf{c})$ defines a particular model instantiation. At each point of execution of this model instantiation, exactly one recursion variable X_i is enabled, so that the set of all possible next steps is determined by the term p_{norm_i} only. In addition, the term p_{norm_i} defines for each action the recursion variable (if any) that is enabled after execution of the action. Process definition, process instantiation, parallel composition, send and receive statements, the loop statement, while do statement, and delay statement are no longer present. Also scoping has been eliminated, apart from one top level variable and channel scope operator, and one top level recursion scope operator.

Note that it is also possible to partly normalize a model. For instance, top level parallelism could be kept intact. In this way, the statement $p_1 \parallel \ldots \parallel p_n$ could be normalized to

$$
\begin{array}{l}
[\![\ D \\
::\ [\![\ X_{11} = (p_{\text{norm}_{11}}), \ldots, X_{1k} = (p_{\text{norm}_{1k}}) :: X_{1i}\ ]\!] \\
\parallel\ \ldots \\
\parallel\ [\![\ X_{n1} = (p_{\text{norm}_{n1}}), \ldots, X_{nm} = (p_{\text{norm}_{nm}}) :: X_{nj}\ ]\!] \\
]\!]
\end{array}
$$

For simulation, normalization can be further simplified. A simulation executes only one trace out of the many traces that may be allowed by the model. Therefore, a considerable reduction in the number of recursion definitions of a ‘normal form’ for simulation is possible. The use of the normal form is further illustrated in the next sections by means of an example.

3.2 Bottle filling line example

Figure 2 shows a bottle filling line consisting of a storage tank that is continuously filled with a flow Q_{in}, a conveyor belt that supplies empty bottles, and a valve that is opened when an empty bottle is below the filling nozzle, and is closed when the bottle is full. When a bottle has been filled, the conveyor starts moving to put the next bottle under the filling nozzle, which takes one unit of time. When the storage tank is not empty, the bottle filling flow Q equals Q_{set}. When the storage tank is empty, the bottle filling flow equals the flow Q_{in}. The system should operate in such a way that overflow of the tank does not occur. Furthermore, it is preferred that the tank does not get empty when filling a bottle. We assume $Q_{\text{in}} < Q_{\text{set}}$.

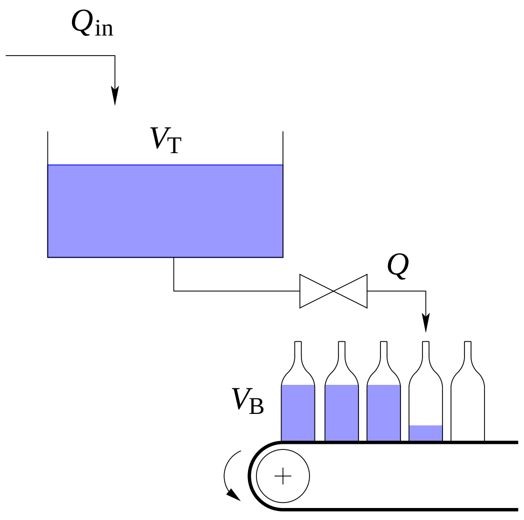

Figure 2: Filling Line

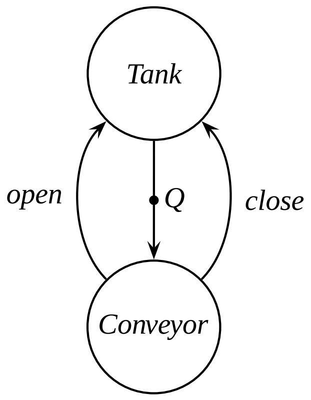

Figure 3: Iconic model of the filling line.

Figure 3 shows an iconic representation of the model of the filling line. It consists of the processes *Tank* and *Conveyor* that interact by means of the channels *open* and *close*, and shared variable Q. The model is defined below. It has two parameters: the initial volume V_{T0} of the storage tank, and the value Q_{in} of the flow that is used to fill the storage tank. The constants Q_{set}, V_{Tmax}, and V_{Bmax} define the maximum value of the bottle filling flow Q, the maximum volume of the storage tank, and the filling volume of the bottles, respectively. The model *FillingLine* consists of the algebraic variable Q, the channels *open* and *close*, and the parallel composition of the process instantiations for the tank and the conveyor.

```
const $Q_{\text{set}}$ : real   $= 3.0$
,     $V_{\text{Tmax}}$ : real $= 20.0$
,     $V_{\text{Bmax}}$ : real $= 10.0$
```

```
model $FillingLine$(val $V_{\text{T0}}, Q_{\text{in}}$ : real) $=$
$[\![$ alg $Q$ : real, chan $open, close$ : void
$::$ $Tank(Q, open, close, V_{\text{T0}}, Q_{\text{in}}) \parallel Conveyor(Q, open, close)$
$]\!]$
```

The tank process has a local continuous variable V_{T} that is initialized to V_{T0}. Its process body is a recursion scope consisting of three modes: closed, opened, and openedempty that correspond to the valve being open, the valve being closed, and the valve being opened while the storage tank is empty. In the mode opened, the storage tank is usually not empty. When the storage tank is empty in mode opened, the delayable skip statement [skip] may be executed causing the next mode to be openedempty. Due to the consistent equation semantics, the skip statement can be executed only if the delay predicate in the next mode openedempty holds. This means, among others, that $V_T = 0.0$ must hold. Therefore, the transition to mode openedempty can be taken only when the storage tank is empty.

```
proc $Tank(\mathsf{alg}\ Q : \mathsf{real},\ \mathsf{chan}\ open?, close? : \mathrm{void},\ \mathsf{val}\ V_{T0}, Q_{in} : \mathsf{real}) =$
$[\![$ cont $V_T$ : real $= V_{T0}$
:: $[\![$ mode closed =
        ( $\dot{V}_T = Q_{in},\ Q = 0.0,\ V_T \le V_{Tmax}\ [\!]\ open?;\ \mathrm{opened}$ )
  , mode opened =
        ( $\dot{V}_T = Q_{in} - Q,\ Q = Q_{set},\ 0.0 \le V_T \le V_{Tmax}$
        $[\!]$ [skip]; openedempty
        $[\!]$ $close?;$ closed
        )
  , mode openedempty =
        ( $V_T = 0.0,\ Q = Q_{in}\ [\!]\ close?;\ \mathrm{closed}$ )
  :: closed
  $]\!]$
$]\!]$
```

Process *Conveyor* supplies an empty bottle in 1 unit of time ($V_B := 0.0;\ \Delta 1.0$). Then it synchronizes with the storage tank process by means of the send statement *open*!, and it proceeds in mode filling. When the bottle is filled in mode filling ($V_B \ge V_{Bmax}$), the process synchronizes with the storage tank to close the valve and returns to mode moving. The initial mode is moving.

```
proc $Conveyor(\mathsf{alg}\ Q : \mathsf{real},\ \mathsf{chan}\ open!, close! : \mathrm{void}) =$
$[\![$ cont $V_B$ : real $= 0.0$
:: $[\![$ mode moving = ( $V_B := 0.0;\ \Delta 1.0;\ open!;\ \mathrm{filling}$ )
  , mode filling  = ( $V_B \ge V_{Bmax} \to close!;\ \mathrm{moving}$ )
  :: moving
  $]\!]$
$\|$ $\dot{V}_B = Q$
$]\!]$
```

Figure 4 shows the result of a simulation run of the model *FillingLine*(5.0, 1.5), that is with model parameters $V_{T0} = 5.0$ and $Q_{in} = 1.5$.

3.3 Elimination of process instantiation

Elimination of the process instantiations for the *Tank* and *Conveyor* processes, as defined in Section 2.5.8, leads to the following model:

```
model $FillingLine(\mathsf{val}\ V_{T0}, Q_{in} : \mathsf{real}) =$
$[\![$ alg $Q$ : real, chan $open, close$ : void
:: $[\![$ cont $V_T$ : real $= V_{T0}^{L}$
  , var $V_{T0}^{L}$ : real $= V_{T0},\ Q_{in}^{L}$ : real $= Q_{in}$
  :: $[\![$ mode closed =
```

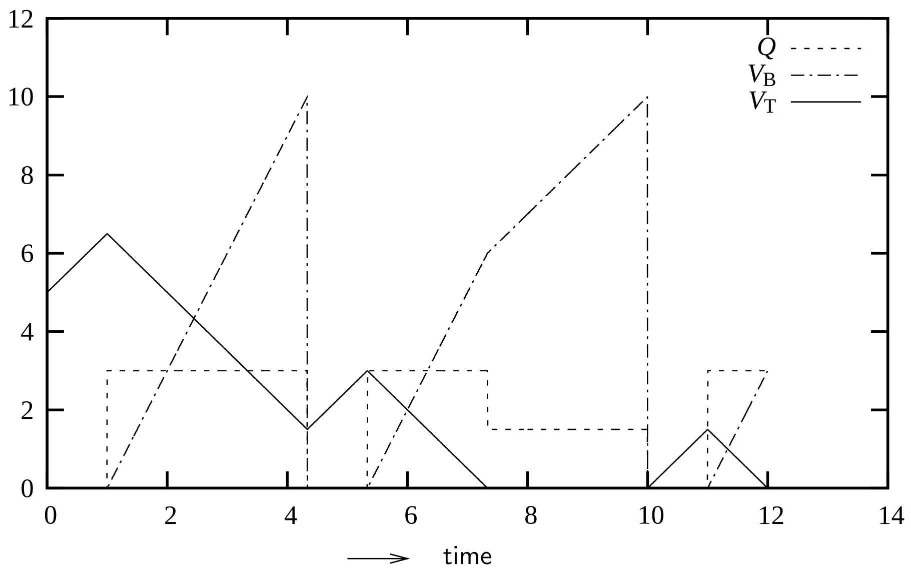

Figure 4: Simulation results of model *FillingLine*.

```
        ( $\dot{V}_T = Q^L_{in},\ Q = 0.0,\ V_T \le V_{Tmax}$ [] $open?$; opened )
    , mode opened =
        ( $\dot{V}_T = Q^L_{in} - Q,\ Q = Q_{set},\ 0.0 \le V_T \le V_{Tmax}$
        [] [skip]; openedempty
        [] $close?$; closed
        )
    , mode openedempty =
        ( $V_T = 0.0,\ Q = Q^L_{in}$ [] $close?$; closed )
    :: closed
    ]|
  ]|
|| |[ cont $V_B$ : real = 0.0
  :: |[ mode moving = ( $V_B := 0.0$; $\Delta 1.0$; $open!$; filling )
    , mode filling  = ( $V_B \ge V_{Bmax} \to close!$; moving )
    :: moving
    ]|
  || $\dot{V}_B = Q$
  ]|
]|
```

To avoid naming conflicts between the formal parameters V_{T0} and Q_{in} declared in the process definition for process *Tank*, and the actual arguments V_{T0} and Q_{in} in the process instantiation *Tank*(Q, *open*, *close*, V_{T0}, Q_{in}), the newly defined local discrete variables that are used to hold the values of the last two parameters of the process instantiation, are renamed to V^L_{T0} and Q^L_{in}.

3.4 Elimination of parallel composition

Elimination of parallel composition and translation to the normal form as discussed in Section 3.1 leads to the model:

```
model $\mathit{FillingLine}(\mathsf{val}\ V_{\mathrm{T0}},\ Q_{\mathrm{in}} : \mathsf{real}) =$
$\|[$ alg $Q$ : real, chan $\mathit{open}, \mathit{close}$ : void
:: $\|[$ cont $V_{\mathrm{T}} : \mathsf{real} = V^{\mathrm{L}}_{\mathrm{T0}},\ V_{\mathrm{B}} : \mathsf{real} = 0.0$
  , var $t$ : real, $V^{\mathrm{L}}_{\mathrm{T0}} : \mathsf{real} = V_{\mathrm{T0}},\ Q^{\mathrm{L}}_{\mathrm{in}} : \mathsf{real} = Q_{\mathrm{in}}$
  :: $\|[$ moving_closed =
         ( $\dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}},\ Q = 0.0,\ V_{\mathrm{T}} \leq V_{\mathrm{Tmax}},\ \dot{V}_{\mathrm{B}} = Q$
         $[]$ $\{V_{\mathrm{B}}, \mathsf{time}\} : V_{\mathrm{B}} = 0.0,\ t = \mathsf{time} + 1.0 \gg \tau$; moving$_0$_closed
         )
     , moving$_0$_closed =
         ( $\dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}},\ Q = 0.0,\ V_{\mathrm{T}} \leq V_{\mathrm{Tmax}},\ \dot{V}_{\mathrm{B}} = Q$
         $[]$ $\mathsf{time} \geq t \rightarrow$ skip; moving$_1$_closed
         )
     , moving$_1$_closed =
         ( $\dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}},\ Q = 0.0,\ V_{\mathrm{T}} \leq V_{\mathrm{Tmax}},\ \dot{V}_{\mathrm{B}} = Q$
         $[]$ $\mathit{open}$; filling_opened
         )
     , filling_opened =
         ( $\dot{V}_{\mathrm{T}} = Q^{\mathrm{L}}_{\mathrm{in}} - Q,\ Q = Q_{\mathrm{set}},\ 0.0 \leq V_{\mathrm{T}} \leq V_{\mathrm{Tmax}},\ \dot{V}_{\mathrm{B}} = Q$
         $[]$ [skip]; filling_openedempty
         $[]$ $V_{\mathrm{B}} \geq V_{\mathrm{Bmax}} \rightarrow \mathit{close}$; moving_closed
         )
     , filling_openedempty =
         ( $V_{\mathrm{T}} = 0.0,\ Q = Q^{\mathrm{L}}_{\mathrm{in}},\ \dot{V}_{\mathrm{B}} = Q$
         $[]$ $V_{\mathrm{B}} \geq V_{\mathrm{Bmax}} \rightarrow \mathit{close}$; moving_closed
         )
     :: moving_closed
     $]\|$
  $]\|$
$]\|$
```

3.5 Substitution of constants and additional elimination

The model below is the result of substitution of the globally defined constants by their values. Furthermore, the discrete variables $Q^{\mathrm{L}}_{\mathrm{in}}$ and $V^{\mathrm{L}}_{\mathrm{T0}}$, that were introduced by elimination of the process instantiations, are eliminated. Also, the presence of the undelayable statements $\{V_{\mathrm{B}}, \mathsf{time}\} : V_{\mathrm{B}} = 0.0,\ t = \mathsf{time} + 1.0 \gg \tau$ and *open* in modes moving_closed and moving$_1$_closed, respectively, allows elimination of the differential equations in these modes.

Most hybrid automaton based model checkers, such as PHAver (Frehse 2005) and HyTech (Henzinger, Ho, and Wong-Toi 1995), do not (yet) have urgent transitions that can be combined with guards. Therefore, the urgency in the guarded statements is removed by making the statements that are guarded delayable, and adding the closed negation of the guard as an additional delay predicate (invariant). E.g. $\mathsf{time} \geq t \rightarrow$ skip is rewritten as $\mathsf{time} \leq t\ []\ \mathsf{time} \geq t \rightarrow$ [skip].

```
model $\mathit{FillingLine}(\mathsf{val}\ V_{\mathrm{T0}},\ Q_{\mathrm{in}} : \mathsf{real}) =$
$\|[$ alg $Q$ : real, chan $\mathit{open}, \mathit{close}$ : void
:: $\|[$ cont $V_{\mathrm{T}} : \mathsf{real} = V_{\mathrm{T0}},\ V_{\mathrm{B}} : \mathsf{real} = 0.0$
  , var $t$ : real
  :: $\|[$ moving_closed =
         ( $V_{\mathrm{T}} \leq 20.0,\ Q = 0.0$
            [] $\{V_\mathrm{B}, \mathsf{time}\} : V_\mathrm{B} = 0.0,\ t = \mathsf{time} + 1.0 \gg \tau;$ moving$_0$_closed
            )
        , moving$_0$_closed =
            ( $\dot{V}_\mathrm{T} = Q_\mathrm{in},\ Q = 0.0,\ V_\mathrm{T} \leq 20.0,\ \dot{V}_\mathrm{B} = 0.0,\ \mathsf{time} \leq t$
            [] $\mathsf{time} \geq t \rightarrow$ [skip]; moving$_1$_closed
            )
        , moving$_1$_closed =
            ( $V_\mathrm{T} \leq 20.0,\ Q = 0.0$
            [] *open*; filling_opened
            )
        , filling_opened =
            ( $\dot{V}_\mathrm{T} = Q_\mathrm{in} - 3.0,\ Q = 3.0,\ 0.0 \leq V_\mathrm{T} \leq 20.0,\ \dot{V}_\mathrm{B} = 3.0,\ V_\mathrm{B} \leq 10.0$
            [] [skip]; filling_openedempty
            [] $V_\mathrm{B} \geq 10.0 \rightarrow$ [*close*]; moving_closed
            )
        , filling_openedempty =
            ( $V_\mathrm{T} = 0.0,\ Q = Q_\mathrm{in},\ \dot{V}_\mathrm{B} = Q,\ V_\mathrm{B} \leq 10.0$
            [] $V_\mathrm{B} \geq 10.0 \rightarrow$ [*close*]; moving_closed
            )
        :: moving_closed
        ]|
    ]|
]|
```

Figure 5 shows a graphical representation of the model. By means of straightforward mathematical analysis of the model, it can be shown that overflow never occurs if $Q_\mathrm{in} \leq 30/13$.

3.6 Tool based verification

As a final step, for the purpose of tool-based verification, the model is translated to the input language of the hybrid IO automaton based tool PHAVer (Frehse 2005). Since most hybrid automata, including PHAVer, do not know the concept of an algebraic variable, first the algebraic variables are eliminated from the χ model. Because of the consistent equation semantics of χ, as defined in (van Beek, Man, Reniers, Rooda, and Schiffelers 2006), each occurrence of an algebraic variable in the model can simply be replaced by the right hand side of its defining equation. The urgency due to unguarded undelayable statements is in principle translated by defining the corresponding flow clause as false. The resulting PHAVer model follows below. Note that an additional variable x is introduced and all derivatives need to be defined in all locations, because of the current inability of PHAVer to define false as flow clause.

```
automaton filling_line
  state_var: Vt,Vb,t,time,x;
  parameter: Vt0,Qin;
  synclabs : open,close,tau;
  loc moving_closed:
    while Vt <= 20 & x==0 wait {x==1 & Vb==0 & Vt==0 & t==0 & time==1};
    when true sync tau do {Vt'==Vt & Vb'==0 & t'==time+1 & time'==time & x'==0}
      goto moving0_closed;
  loc moving0_closed:
    while Vt <= 20 & time <= t wait {Vb==0 & t==0 & time==1 & Vt==30/13};
    when time >= t sync tau do {Vt'==Vt & Vb'==Vb & t'==t & time'==time & x'==0}
      goto moving1_closed;
```

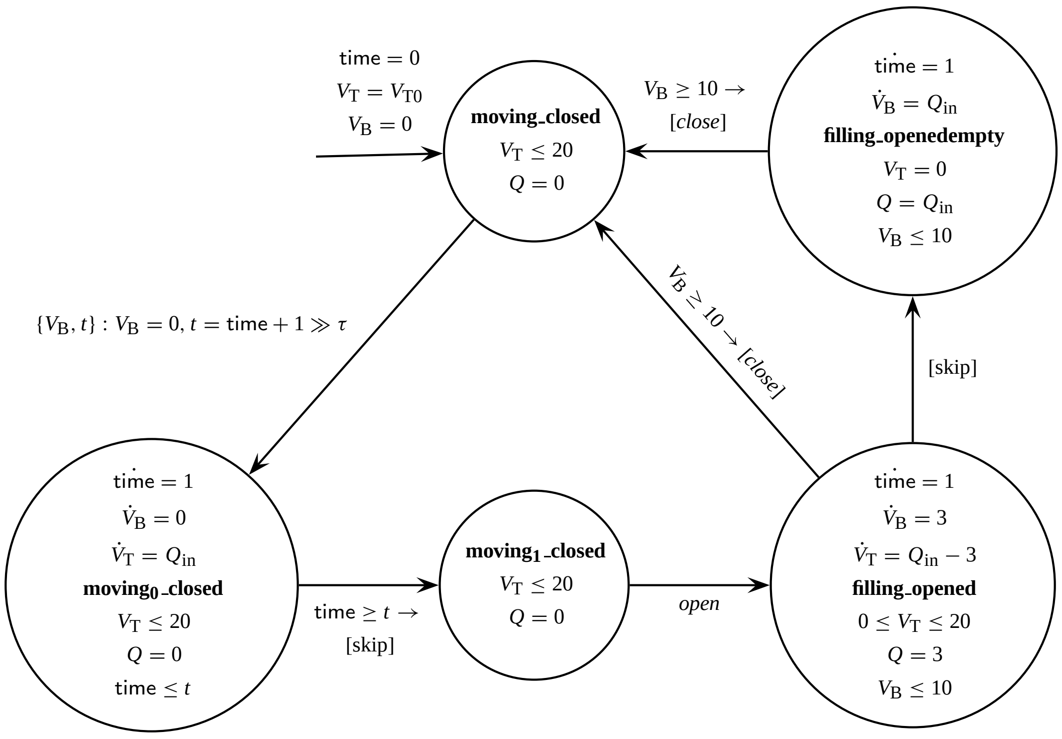

Figure 5: Graphical representation of the normalized χ model.

```
  loc moving1_closed:
    while Vt <= 20 & x==0 wait {x==1 & Vb==0 & Vt==0 & t==0 & time==1};
    when true sync open do {Vt'==Vt & Vb'==Vb & t'==t & time'==time}
      goto filling_opened;
  loc filling_opened:
    while Vt >= 0 & Vt <= 20 & Vb <= 10 wait {Vb==3 & t==0 & time==1 & Vt==30/13-3};
    when Vt==0 sync tau do {Vt'==Vt & Vb'==Vb & t'==t & time'==time}
      goto filling_openedempty;
    when Vb >= 10 sync close do {Vt'==Vt & Vb'==Vb & t'==t & time'==time & x'==0}
      goto moving_closed;
  loc filling_openedempty:
    while Vt == 0 & Vb <= 10 wait {Vb==30/13 & t==0 & time==1};
    when Vb >= 10 sync close do {Vt'==Vt & Vb'==Vb & t'==t & time'==time & x'==0}
      goto moving_closed;
  initially moving_closed & t==0 & Vt == Vt0 & Vb==0 & x==0;
end
```

The following properties were derived: if $Q_{in} = 30/13$ and $0 \le V_{T0} \le V_{Tmax} - 30/13$, overflow does not occur, and the storage tank does not become empty when filling a bottle. The volume of the storage tank then remains in the region $V_{T0} \le V_T \le V_{T0} + 30/13$. If $Q_{in} > 30/13$, eventually overflow occurs. If $Q_{in} < 30/13$, the container becomes empty every time a bottle is filled. In this small example, these properties can also be derived by means of straightforward mathematical analysis of the χ models of Section 3.4 or 3.5.

4 Conclusions

Process algebra originated in the domain of theoretical computer science, where it was designed for the purpose of reasoning about the behavior of concurrent discrete-event systems. Recently, process algebra theory has been extended to include also continuous-time systems, and combined discrete-event / continuous-time, or hybrid systems. The χ process algebra, that has been used as an example in this chapter, illustrates that process algebra is not only suited to verification, but also very well suited to high level modeling and simulation of complex dynamical systems. The compositional semantics of a process algebra facilitates modular composition of processes and statements using not only parallel composition, but also sequential composition, and in fact any kind of combination of statements by means of the process algebra operators. The equational reasoning, that is characteristic of process algebra, allows rewriting of complex specifications to a straightforward normal form, where parallel composition has been eliminated. For the χ process algebra, the normal form is very similar to a hybrid automaton, and thus simplifies the use and development of tools for simulation and verification.

Acknowledgments

The authors thank Albert Hofkamp for providing the main functionality of the χ toolset, and for many helpful comments on drafts of this text. They thank Rolf Theunissen for his preparative work on the bottle filling example, and for analysis of the properties of the resulting hybrid automaton using PHAVer. Finally, they thank Ramon Schiffelers for enabling hybrid simulation of χ models.

References

Alur, R., T. A. Henzinger, and P. H. Ho (1996). Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering 22*(3), 181–201.

Baeten, J. C. M. and W. P. Weijland (1990). *Process Algebra*, Volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge, United Kingdom: Cambridge University Press.

Bergstra, J. A. and J. W. Klop (1984). Process algebra for synchronous communication. *Information and Control 60*(1/3), 109–137.

Bergstra, J. A. and C. A. Middelburg (2005). Process algebra for hybrid systems. *Theoretical Computer Science 335*(2/3), 215–280.

Bortnik, E. M., N. Trčka, A. J. Wijs, B. Luttik, J. M. van de Mortel-Fronczak, J. C. M. Baeten, W. J. Fokkink, and J. E. Rooda (2005). Analyzing a Chi model of a turntable system using Spin, CADP and Uppaal. *Journal of Logic and Algebraic Programming 65*(2), 51–104.

Bundy, A. (1999). A survey of automated deduction. In M. Wooldridge and M. Veloso (Eds.), *Artificial Intelligence Today. Recent Trends and Developments*, Volume 1600 of *Lecture Notes in Computer Science*, pp. 153–174. Springer Verlag.

Clarke, E. M., O. Grumberg, and D. A. Peled (2000). *Model Checking*. MIT Press.

Cuijpers, P. J. L. and M. A. Reniers (2005). Hybrid process algebra. *Journal of Logic and Algebraic Programming 62*(2), 191–245.

Fábián, G. (1999). *A Language and Simulator for Hybrid Systems*. Ph. D. thesis, Eindhoven University of Technology.

Fernandez, J. C., H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu (1996). CADP - a protocol validation and verification toolbox. In *Proceedings 8th Conference on Computer Aided Verification (CAV'96)*, Volume 1102 of *Lecture Notes in Computer Science*, pp. 437–440.

Frehse, G. (2005). PHAVer: Algorithmic verification of hybrid systems past HyTech. In M. Morari and L. Thiele (Eds.), *Hybrid Systems: Computation and Control, 8th International Workshop*, Volume 3414 of *Lecture Notes in Computer Science*, pp. 258–273. Springer-Verlag.

Groote, J. F. (1997). The syntax and semantics of timed μCRL. Technical Report SEN-R9709, CWI, The Netherlands.

Henzinger, T. A., P.-H. Ho, and H. Wong-Toi (1995). A user guide to HYTECH. In *First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, Lecture Notes in Computer Science 1019, pp. 41–71. Springer Verlag.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Englewood-Cliffs: Prentice-Hall.

Hofkamp, A. T. (2001). *Reactive machine control, a simulation approach using χ*. Ph. D. thesis, Eindhoven University of Technology.

Holzmann, G. J. (2003). *The SPIN Model Checker: Primer and Reference Manual*. Boston: Addison Wesley Professional.

Larsen, K. G., P. Pettersson, and W. Yi (1997). UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer 1*(1–2), 134–152.

Linz, P. (2001). *An Introduction to Formal Languages and Automata*. Jones and Bartlett.

Milner, R. (1980). *A Calculus of Communicating Systems*, Volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

Naumoski, G. and W. Alberts (1998). *A Discrete-Event Simulator for Systems Engineering*. Ph. D. thesis, Eindhoven University of Technology.

Rounds, W. C. and H. Song (2003). The ϕ-Calculus: A language for distributed control of reconfigurable embedded systems. In O. Maler and A. Pnueli (Eds.), *Hybrid Systems : Computation and Control, 6th International Workshop*, Lecture Notes in Computer Science 2623, pp. 435–449. Springer-Verlag.

The MathWorks, Inc (2005). *Writing S-functions, version 6*. http://www.mathworks.com.

van Beek, D. A., K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers (2006). Syntax and consistent equation semantics of hybrid Chi. *Journal of Logic and Algebraic Programming*. to appear.

van Beek, D. A. and J. E. Rooda (2000). Languages and applications in hybrid modelling and simulation: Positioning of Chi. *Control Engineering Practice 8*(1), 81–91.

van Beek, D. A., A. van den Ham, and J. E. Rooda (2002). Modelling and control of process industry batch production systems. In *15th Triennial World Congress of the International Federation of Automatic Control*, Barcelona. CD-ROM.