Behavioural Hybrid Process Calculus
Translation to Modelica

Ing. A.E. van Putten
January 14, 2007

University of Twente
Department of Electrical Engineering, Mathematics and Computer Science
(EEMCS)

Contents

CHAPTER 1

Introduction

This chapter will describe the problem that is addressed in this thesis. Also work related to the subject of this thesis, the structure of the document and terminology will be described in this chapter.

1.1 Problem description

Simulation is a well-established technique for development and analysis of dynamical systems and is widely used in industry and academia. Simulation is frequently used for development of models for designing existing systems. Often such systems exhibit discrete behaviour as well as continuous behaviour.

Hybrid systems combine discrete events and continuous behaviour. Discrete events are caused by the evolution of continuous dynamics or external stimuli. The continuous behaviour can change in response to the discrete events or the flow of time. Often such systems behaviour can be observed in embedded systems. Such systems usually observe and react on a continuous time process but the controller itself is of discrete nature.

Behavioural Hybrid Process Calculus (BHPC) is an extension of classical process algebra, based on behavioural theory that is suitable for the modelling and analysis of hybrid dynamical systems [BK05, BKU05]. It provides a natural framework for the concurrent composition of such systems, and can deal with non-determinism.

The language is recently developed and no simulation tools are currently available for this language. One of the various ways to simulate a BHPC model is to translate the model to the simulation language Modelica. This thesis describes an algorithm which translates a BHPC specification into a hybrid automaton. This hybrid automaton can be simulated by Modelica by using Modelicas StateGraph library and some additional code.

1.2 Related work

Behavioural Hybrid Process Calculus is a relatively new process algebra. Helen Schonenberg has developed a discrete simulator for this language which is described in [Sch06, KS05].

HyPA is another hybrid formalism for which R.A. Schouten has developed a simulator [Sch05]. Other hybrid simulators are CHARON [HL02], HyVisual [BCL+05] and Hybrid χ[MS06].

CHARON is a hybrid process language that comes with a simulator and a visual editor. The language has been developed to model and analyze interacting hybrid systems as communicating agents.

HyVisual is a block-diagram editor and simulator for continuous-time dynamical systems and hybrid systems. The visual modeler supports construction of hierarchical hybrid systems and uses a block-diagram representation of ordinary differential equations to define the continuous behaviour.

Hybrid χ is a hybrid formalism that differs from other hybrid formalisms in the sense of expressivity. This language tries to make the expressivity of the language high while keeping it easy to use.

HyTech [AHS96] is an automatic tool for the analysis of embedded systems. HyTech computes the condition under which a linear hybrid system satisfies a temporal requirement. Hybrid systems are specified as collections of hybrid automata with discrete and continuous components. Temporal requirements are verified by symbolic model checking. On failure HyTech is able to generate a diagnostic error trace.

The StateFlow and Simulink pair is a toolset for the modelling and design of dynamical systems. It is based on the combination of Statecharts, finite state machines and flow diagrams. The tool is a commercial product of Mathworks.

1.3 Structure of the thesis

This thesis describes the translation of a Behaviour Hybrid Process Caculus (BHPC) process to a Modelica model that can be simulated. This translation is realised in a number of steps. The structure of this document tries to follow the order of these translations steps.

Chapter 2 introduces and describes the BHPC language. The first step is the translation of the BHPC model into an internal representation. A parser is developed for this purpose. This parser and the internal representation that it generates are described in chapter 3.

The next step is to translate this BHPC model into a hybrid automaton that can be simulated by Modelica. First, chapter 4 introduces the Modelica language and the StateGraph library that is used for the simulation. This chapter also describes how a hybrid automaton can be simulated with the StateGraph library and some additional Modelica code.

Chapter 5 describes the actual translation from the BHPC model into Modelica code. Finally chapter 6 describes the conclusion drawn from this thesis.

Appendixes A and B describe respectively the ASCII version of the BHPC language and a railroad diagram of this language. Appendixes C shows an XML schema of the internal representation.

CHAPTER 2

Behavioural Hybrid Process Calculus

2.1 Introduction

The growing interest in hybrid systems both in computer science and control theory has generated a new interest in models and formalisms that can be used to specify and analyse such systems.

Process algebra is a theoretical framework for the modelling and analysis of concurrent discrete event systems that has been developed within the computer science community during the last quarter century [Mil89, Hoa85, BK84, BB87]. This framework has provided more insight in concepts such as the observable behaviour in the presence of non-determinism, system composition of concurrent systems and notions of behavioural equivalence of such systems. It is believed that the basic tenets of process algebra are highly compatible with the behavioural approach to dynamical systems [Kri06, PW98].

Behavioural Hybrid Process Calculus (BHPC) is an extension of classical process algebra, based on behavioural theory that is suitable for the modelling and analysis of hybrid dynamical systems. It provides a natural framework for the concurrent composition of such systems, and can deal with non-deterministic behaviour that may arise from the occurrence of internal switching events.

2.2 Trajectories

In control theory the traditional presentation of dynamic behaviour is a number of continuous-time input and output variables whose evolutions and influences depend on the evolution of state variables. This evolution is typically defined in terms of differential equations. In the behavioural approach on which BHPC is inspired, system behaviour is characterised by a time-dependent relation between the observable or manifest variables of a system. Input and output become derived notions that depend on the constraints that the overall relation imposes on the individual variables. This means that behaviour can be simply

seen as the set of all allowed real-time evolutions, or trajectories, of the system variables.

Trajectories are defined over bounded time intervals (0,t], and map to a signal space to define the evolution of the system. Components of the signal space correspond to the different aspects of the continuous-time behaviour. These components are associated with trajectory qualifiers that identify them.

Definition 2.2.1 (Signal space) Let $\mathcal{W}$ be a set of signal domains (typically $\subseteq \mathbb{R}$) and $\mathcal{T}$ be a set of trajectory qualifiers. A signal space is a pair

$$\mathbb{W} = (W_1 \times \cdots \times W_n, (t_1, ..., t_n))$$

with $W_i \in \mathcal{W}, t_i \in \mathcal{T}$, where t_i denotes the trajectory qualifier of W_i, and $t_i \neq t_j$ for $i \neq j$, i.e., all W_i have different trajectory qualifiers [Kri06]. □

Definition 2.2.2 (Trajectory) Let $\mathbb{W} = (W_1 \times \cdots \times W_n, (t_1, ..., t_n))$ be a signal space. Then a trajectory in a signal space $\mathbb{W}$ is a function

$$\varphi_{\mathbb{W}} : (0, t] \to W_1 \times \cdots \times W_n$$

where $t \in \mathbb{R}_+$ is the duration of the trajectory, also denoted as $\mathrm{t}(\varphi)$. We will omit subscript $\mathbb{W}$ when a signal space is clear from the context [Kri06]. □

Definition 2.2.3 (Empty trajectory) ϵ denotes an empty or completed trajectory. □

Definition 2.2.4 (Set of trajectory qualifiers) A function $\mathrm{T} : \Phi \to \mathcal{T}$, where Φ is a set of trajectories and $\mathcal{T}$ is a set of qualifiers, collects all trajectory qualifiers of the trajectory:

$$\mathrm{T}(\varphi_{\mathbb{W}}) = \{q \mid \varphi_{\mathbb{W}} : (0, u] \to W_1 \times \cdots \times W_n \wedge \mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, ..., q_n)) \wedge t \in \{q_1, ..., q_n\}\}$$

Definition 2.2.5 (Projection) Let $\varphi : (0, u] \to W_1 \times \cdots \times W_n$ be a trajectory, such that $\mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, ..., q_n))$. Then a projection of the trajectory w.r.t. a trajectory qualifier $q_i (i = 1, ..., n)$ is the trajectory

$$\pi^{q_i}(\varphi) : (0, u] \to W_i$$

in signal space $\mathbb{W}_i = (W_i, q_i)$. □

Definition 2.2.6 (Composition of trajectories) Let H be a set of synchronising trajectory qualifiers, and let $\varphi : (0, t] \to W_1'' \times \cdots \times W_n''$ in $\mathbb{W} = (W_1'' \times \cdots W_k'', (t_1'', ..., t_k''))$ and $\psi : (0, u] \to W_1' \times \cdots \times W_m'$ in $\mathbb{W}' = (W_1' \times \cdots \times W_m', (t_1', ..., t_m'))$ be trajectories such that $\mathrm{T}(\varphi) \cap \mathrm{T}(\psi) \subseteq H$ and $\pi^{\mathrm{T}(\varphi) \cap \mathrm{T}(\psi)}(\varphi) = \pi^{\mathrm{T}(\varphi) \cap \mathrm{T}(\psi)}(\psi)$. Then a composition of trajectories is a trajectory.

$$\varphi \times_H \psi : (0, u] \to W_1 \times \cdots \times W_n,$$

in $\mathbb{W} = (W_1 \times \cdots \times W_n, (t_1, ..., t_n))$ such that

$$\mathrm{T}(\varphi \times_H \psi) = \mathrm{T}(\varphi) \cup \mathrm{T}(\psi),$$
$$\pi^{\mathrm{T}(\varphi)}(\varphi \times_H \psi) = \varphi,$$
$$\pi^{\mathrm{T}(\psi)}(\varphi \times_H \psi) = \psi.$$

2.3 Hybrid transition system

Automata, state-transitions diagrams and other similar models are often used to present the dynamic behaviour of modelled systems. These diagrams consist of a set of states S and some construct defining the changes of the states, usually transitions. These transitions are given as a relation over the Cartesian product of the states ($S \times S$). Labelled transition systems are diagrams which extend state transition with actions which are related to the transitions. A hybrid transition system is a labelled transition system with two types of transitions.

Definition 2.3.1 (HTS) A hybrid transition system [Kri06] is a tuple $HTS = \langle S, \mathcal{A}, \rightarrow, \mathbb{W}, \Phi, \rightarrow_c \rangle$, where

- S is a state space;
- $\mathcal{A}$ is a set of discrete action names;
- $\rightarrow \subseteq S \times \mathcal{A} \times S$ is a discrete transition relation;
- $\mathbb{W}$ is a signal space;
- Φ is a set of trajectories $\varphi : (0, t] \rightarrow W_1 \times ... \times W_n$ for $t \in \mathbb{R}_+$;
- $\rightarrow_c \subseteq S \times \Phi \times S$ is a continuous transition relation.

  For clarity reasons we will write

$$s \xrightarrow{c} s' \Leftrightarrow (s, a, s') \in \rightarrow$$

$$s \xrightarrow{\varphi} s' \Leftrightarrow (s, \varphi, s') \in \rightarrow_c$$

The set of discrete action names includes a silent action known as τ. It does not represent a potential communication and is not directly observable. Silent action may be used to specify non-deterministic behaviour like the internal actions described in [Mil89]. □

2.4 Language and semantics

The BHPC language [Kri06] is based on hybrid transition systems. The syntax is defined as follows:

Definition 2.4.1 (BHPC syntax)

$$B ::= 0 \mid a.B \mid a(v : V).B(v) \mid [\varphi \mid \Phi].B \mid \langle Pred \rangle.B \mid$$

$$\sum_{i \in I} B_i \mid B||_A^H B \mid \text{new } w.B \mid B[\sigma] \mid P$$

- 0 is a deadlock, the process that does not show any behaviour.
- $a.B$ is an action-prefix, where $a \in \mathcal{A}$ is a discrete action name and B is a process. It first performs a and then engages in B. Action-prefix denotes discrete transitions in the hybrid transition system.
- $a(v : V).B(v) \triangleq \sum_{v \in V} a(v).B(v)$ is a parameterized action prefix.

- $[f \mid \Phi].B(f)$ is a trajectory prefix, where f is a trajectory variable and Φ is a set of trajectories. It takes a trajectory or a prefix of a trajectory in Φ. If a trajectory or a part of it was taken and there exists a continuation of the trajectory, then the system can continue with a trajectory from the trajectory continuations set. If a whole trajectory was taken, then the system may continue with B, too.
- $\langle Pred \rangle.B$ is a guarded prefix. Only when $Pred$ is satisfied the behaviour will continue as B. When $Pred$ is not satisfied no further actions can be taken resulting in a deadlock.
- $\sum_{i \in I} B_i$ is a choice of processes. To generate the set they allow arbitrary index sets I. It chooses before taking an action prefix or trajectory prefix. The binary version of the choice operator is denoted $B_1 + B_2$
- $B||_A^H B$ is a parallel composition of two processes with an interconnection set H and a synchronisation set A. The interconnection set $H \subseteq \mathcal{T}$ is a set of trajectory qualifiers for the synchronisation of trajectories and the synchronisation set $A \subseteq \mathcal{A}$ is the set of action names for the synchronisation of discrete transitions.
- new $w.B$ is a hiding operator, where w is a set of discrete action names and/or trajectory qualifiers to hide.
- $B[\sigma]$ is a renaming operator, where σ is a renaming function.
- $P \triangleq B$ is a recursive equation, where P is a process identifier. □

2.4.1 Operators

This section explains the BHPC operators in more detail.

Action prefix $a.B$

The well known action prefix from discrete process algebra. Process $a.B$ defines a process which engages in the action a and then behaves as B. The invisible action τ is also allowed, this action is not observable and cannot be used to represent potential communication. Silent action(s) can be used to specify non-deterministic behaviour.

$$a.B \xrightarrow{a} B$$

Parameterised action prefix $a(v : V).B(v)$

This is a parameterised version of the action prefix. The parameterised version of the action prefix is typically used to pass information while synchronizing with a parallel process.

$$a(v : V).B(v) \triangleq \sum_{v \in V} a(v).B(v)$$

Trajectory prefix $[\varphi \mid \Phi].B(f)$

A trajectory prefix defines the behaviour that starts with a trajectory denoted by f and is followed by the trajectory continuation or behaviour specified by B.

$$[f \mid \Phi].B(f) \xrightarrow{\varphi} [f' \mid \Phi \backslash\backslash \varphi].B(\varphi; f') \text{ for all } \varphi \in \overset{-+}{\Phi}$$

where Φ is a set of trajectories such that $\forall \varphi, \psi \in \Phi \mathrm{T}(\varphi) = \mathrm{T}(\psi)$, f, f' are trajectory variables and $\varphi; \psi \in \Phi$ or $\varphi \in \Phi$ such that $t(\varphi) \neq \infty$ and $\varphi \neq \epsilon$. If a trajectory or part of it was taken and there exists a continuation of the trajectory, then the system can continue with a trajectory from the continuations set. However, if a whole trajectory was taken, then the system may continue with the consecutive process with the substituted trajectories. $(\varphi; f')$ defines substitution of the taken trajectories in the following processes, i.e., all instances of f in B are substituted by the taken trajectory φ concatenated with its follow-up f', or if it is finished, by the whole taken trajectory φ.

The bahaviour for empty trajectories ϵ will be defined later, after the definition for cancatenation is given.

The following natation for trajectory prefix can be convenient to express restrictions and exit conditions.

$$[q_1, ..., q_m \mid \Phi \downarrow Pred \Downarrow Pred_{exit}]$$

where

- $q_1, ..., q_m$ are trajectory qualifiers, which can be used to access corresponding parts of trajectories.
- $Pred$ are restrictions for the trajectory set.
- $Pred_{exit}$ are exit conditions for the trajectory set.

Concatenation

Concatenation extends the definition of the trajectory prefix. It formalises behaviour after taking a complete trajectory. The process can choose to continue with another trajectory or an action prefix, depending on the successive process. Concatenation is formalised by the following SOS rules.

$$\frac{B(\varphi) \xrightarrow{\psi} B'}{[f \mid \Phi].B(\varphi) \xrightarrow{\varphi;\psi} B'} \varphi \in \Phi$$

$$\frac{B(\epsilon) \xrightarrow{a} B'}{[f \mid \Phi].B(f) \xrightarrow{a} B'} \epsilon \in \Phi$$

The first rule shows how to concatenate two trajectories. While the second rule defines a situation, where after taking a whole trajectory the process continues with an action prefix.

The following rule is derived from concatenation and trajectory prefix rules. If $\epsilon \in \Phi$ then

$$[f \mid \Phi].B(f) = [f \mid \Phi \backslash \epsilon].B(f) + B(\epsilon)$$

Guard $\langle Pred \rangle.B$

The guard operator can be used to check certain conditions explicitly. If they are satisfied the process is allowed to progress, if the conditions are not satisfied the process will deadlock since no transition can be taken. Its behaviour is described by the following SOS rules.

$$\frac{B \xrightarrow{a} B'}{\langle Pred \rangle.B \xrightarrow{a} B'} \models Pred$$

$$\frac{B \xrightarrow{\varphi} B'}{\langle Pred \rangle.B \xrightarrow{\varphi} B'} \models Pred$$

Choice $\sum_{i \in I} B_i$

Choice is a generalised operator on sets of behaviour expressions. To generate the set they allow arbitrary index sets I. It can the thought of as a generalisation of ordinary process algebra choice. The choice operator behaviour is defined by the following SOS rules.

$$\frac{B(w) \xrightarrow{a} B'}{\sum_{v \in I} B(v) \xrightarrow{a} B'} w \in I$$

$$\frac{B(w) \xrightarrow{\varphi} B'}{\sum_{v \in I} B(v) \xrightarrow{\varphi} B'} w \in I$$

The first rule defines the choice for action prefixes which is the same as in usual process algebras. The second rule defines that the choice for trajectories is made before taking the trajectory.

Parallel composition $B_1 \,||_A^H\, B_2$

Parallel compositions is used to model concurrent evolutions of several processes. During this evolution they can interact through discrete and continuous-time transitions. The set of trajectory qualifiers H contains the trajectory qualifiers which are subject to synchronisation. The synchronisation semantics are defined by the following SOS rules:

The set of discrete actions A contains the discrete actions which are subject to synchronisation.

$$\frac{B_1 \xrightarrow{a} B_1', B_2 \xrightarrow{a} B_2'}{B_1 \,||_A^H B_2 \xrightarrow{a} B_1' \,||_A^H B_2'} \quad a \in A$$

$$\frac{B_1 \xrightarrow{a} B_1'}{\begin{array}{c} B_1 \,||_A^H B_2 \xrightarrow{a} B_1' \,||_A^H B_2 \\ B_2 \,||_A^H B_1 \xrightarrow{a} B_2 \,||_A^H B_1' \end{array}} \quad a \notin A$$

$$\frac{B_1 \xrightarrow{\varphi} B_1', B_2 \xrightarrow{\psi} B_2'}{B_1 \,||_A^H B_2 \xrightarrow{\varphi \times_H \psi} B_1' \,||_A^H B_2'} \quad \begin{array}{l} \mathcal{T}' = \mathcal{T}(\varphi) \cap \mathcal{T}(\psi) \\ \text{and } \mathcal{T}' \subseteq H \end{array}$$

where $\varphi \times_H \psi$ is a composition of trajectories as defined in definition 2.2.6.

Hiding new $w.B$

The hiding operator is a scope restriction operator. new $w.B$ restricts the use of the names w to B. The hiding of discrete actions is the same as for ordinary process calculus. The hiding of trajectories however should be used with care since two different trajectories can become observably equivalent if only observable parts of the behaviour are visible. Hiding can influence the outcome of parallel composition. The following SOS rules define the behaviour in the case of hiding:

$$\frac{B \xrightarrow{a} B'}{\text{new } w.B \xrightarrow{\tau} \text{new } w.B'} \quad a \in w$$

$$\frac{B \xrightarrow{a} B'}{\text{new } w.B \xrightarrow{a} \text{new } w.B'} \quad a \notin w$$

$$\frac{B \xrightarrow{\varphi} B'}{\text{new } w.B \xrightarrow{\pi^{T(\varphi)\setminus w(\varphi)}} \text{new } w.B'}$$

Renaming $B[\sigma]$

The renaming operator $B[\sigma]$ where σ is a renaming function, makes the new name for the discrete action or trajectory qualifier available in B and restricts the use of the old name in B. The renaming function does not change trajectory types. The behaviour of the renaming operator is defined by the following SOS rules:

$$\frac{B \xrightarrow{a} B'}{B[\sigma] \xrightarrow{\sigma(a)} B'[\sigma]}$$

$$\frac{B \xrightarrow{\varphi} B'}{B[\sigma] \xrightarrow{\sigma(\varphi)} B'[\sigma]}$$

Recursion

Recursion allows defining processes in terms of each other, like in the equation $P \triangleq B$, where P is a process identifier. Actions and signal types of B are the only allowed actions and signal types in P. Recursion can be applied with the following SOS rules.

$$\frac{B \xrightarrow{a} B'}{P \xrightarrow{a} B'} \quad P \triangleq B$$

$$\frac{B \xrightarrow{\varphi} B'}{P \xrightarrow{\varphi} B'} \quad P \triangleq B$$

2.5 Examples

This paragraph will show two examples of BHPC models, namely the thermostat and the bouncing ball.

Example 2.5.1 (Thermostat) A thermostat monitors and controls the temperature in a room. The temperature can be controlled by switching the heating on and off. This switching can occur at any time when $l \in [tempOn, tempMin]$ or $l \in [tempOff, tempMax]$, however when the temperature is greater or equal to $tempMax$ is must switch off and when the temperature is lower or equal to $tempMin$ is must switch on. This behaviour is modelled in the following BHPC specification:

$$\mathrm{ThOff}(l_0) \triangleq [l \mid \Phi_{\mathit{off}}(l_0) \Downarrow \mathit{tempOn} \geqslant l \geqslant \mathit{tempMin}].\mathrm{on}.\mathrm{ThOn}(l)$$

$$\mathrm{ThOn}(l_0) \triangleq [l \mid \Phi_{on}(l_0) \Downarrow \mathit{tempOff} \leqslant l \leqslant \mathit{tempMax}].\mathrm{off}.\mathrm{ThOn}(l)$$

$$\Phi_{\mathit{Off}}(l_0) = \{l : (0,t] \rightarrow \mathbb{R} \mid l(0) = l_0, \dot{l} = -Kl\}$$

$$\Phi_{On}(l_0) = \{l : (0,t] \rightarrow \mathbb{R} \mid l(0) = l_0, \dot{l} = K(h-l)\}$$

The first two lines define the two processes *ThOff* and *ThOn*, for the behaviour when the heater is respectively turned off and on. Each process starts with a trajectory prefix in which the temperature changes according to the state of the heater. When the exit conditions, $\mathit{tempOn} \geqslant l \geqslant \mathit{tempMin}$ for the off state or $\mathit{tempOff} \leqslant l \leqslant \mathit{tempMax}$ for the on state, are satisfied the processes can continue with a discrete action which turns the heater on or off. After the action the process will continue with recursion into the other process and thus creating an infinite loop which keeps the temperature of the room between *tempMin* and *tempMax*. A hybrid automaton of this example is shown in figure 4.1. □

Example 2.5.2 (Bouncing ball) A bouncing ball bounces up and down, loosing a fraction of its energy on every bounce. This behaviour is modelled by the following BHPC specification:

$$\mathrm{BB}(h_0, v_0) \triangleq [h, v \mid \Phi(h_0, v_0) \Downarrow h = 0].\mathrm{BB}(0, -c * v)$$

$$\Phi(h_0, v_0) = \{h, v : (0,t] \rightarrow \mathbb{R} \mid h(0) = h_0, v(0) = v_0, \dot{h} = v, \dot{v} = -g, h \geqslant 0\}$$

The trajectory prefix describes the dynamics of the ball until the bounce. When the ball reaches the ground, the exit conditions $h = 0$ is satisfied and the behaviour will continue at the beginning of the process through the recursion operator. The recursion step updates the trajectory qualifiers. The qualifier for the height is set to 0 since the ball is on the ground and the velocity is negated as the ball needs to travel up because of the bounce, the loss of energy is handled by a multiplication with c which defines what amount of the energy remains after each bounce. □

2.6 Conclusions

BHPC is an extension of classical process algebra, based on behavioural theory that is suitable for modelling and analysis of hybrid dynamical systems. Trajectories are used to describe the continuous evolution of the models. This evolution is typically defined in terms of differential equations. In the behavioural approach, on which BHPC is inspired, input and output become derived notions that depend on the constraints that the overall relation imposes on the individual variables. This means behaviour can be seen as the set of allowed

evolutions or trajectories of the system variables. These trajectories map to a signal space to define the evolution of the system.

Hybrid transition systems are a popular way to model hybrid systems. A hybrid transition systems is a labelled transition system with two types of transitions, namely discrete and continuous ones.

The BHPC language features a set of operators consisting of the deadlock, action prefix, trajectory prefix, predicate prefix, choice, parallel, renaming, hiding and recursion. These operators can be used to define a discrete process for the model. Additionally trajectories can be defined that define describe the continuous evolution of the model.

CHAPTER 3

A parser for BHPC

3.1 Introduction

Parsing is usually defined as the process of analysing an input sequence in order to determine its grammatical structure with respect to a given formal grammar. A parser is a computer program that automates this process. In the context of this project the parser analyses a given BHPC specification, checks its integrity and generates output in the form of an internal representation.

The BHPC language was developed as a mathematical formalism, it's notation includes symbols from various alphabets which are not present in the widely used ASCII character set [Wika]. Therefore creating the BHPC models could cause some difficulty. We summarize a number of options to simplify model creation using a computer.

- *LaTeX* - LaTeX has extensive mathematical support. Control strings (like e.g. `\varphi` for φ) are used to denote mathematical and other symbols. These control strings are composed of characters from the ASCII character set. This makes it relatively easy to design a BHPC specification using any text editor. The control strings however are often relative large and thus can "cloud" the specification making it harder to read.

- *Custom designer* - A custom made application which provides the user with means to develop a BHPC specification in its mathematical form. This application can write the specification in the Unicode character set or even translate it to the LaTeX style. However such a custom tool requires time to develop and therefore lies beyond the scope of this project.

- *Unicode [uni]* - Using a third party application which provides facilities to utilize mathematical symbols and can output these to a Unicode file. A search on the internet shows that many Unicode editors often use rather slow input systems as virtual keyboards for an extended character set.

Moreover these applications rely on installed character sets to display the characters. Therefore this alternative is far from optimal.

- *ASCII [Wika] variant of the language* A variant of the mathematical notation designed to make it easy to input using a default keyboard and editor of choice, easy to read while keeping it as close as possible to the original notation. This option does not rely on any third party software and keeps the specifications more readable than the LaTeX alternative.

With usability and available time in mind it is decided to proceed with the ASCII variant option. The next paragraph will describe the syntax and semantics of this language which will be referred to as ASCII BHPC.

3.2 ASCII BHPC

The ASCII BHPC language is designed to match the original, mathematical, BHPC language as much as possible while making sure the models can be easily created by means of a text editor and characters present on commonly used keyboards. This chapter will describe the syntax and semantics for this language. For a complete definition of the language we refer to appendix A. This definition is conform the ANTLR syntax which is a variant of the Extended Backus Naur Form (EBNF) [iso].

Please note that this chapter contains the BHPC operator superposition. This operator is currently not included in the BHPC language, however it is supported by the parser for future use.

3.2.1 Syntax

The ASCII BHPC definition allows one to define actions, qualifiers, constants, processes and/or trajectories within a model. First we will describe the global elements and expressions used throughout the entire specification. Next we will describe how actions, qualifiers and constants can be defined followed by descriptions for the processes and trajectories.

Global elements

ASCII BHPC defines a number of global elements which are used throughout the specification.

- *IDENTIFIER* - A sequence of Latin characters and/or numeric characters, however the first characters must be a Latin one.
- *NUMBER* - A sequence of numeric characters optionally followed by a dot and one or more numeric characters.
- *qualifier* - A synonym for IDENTIFIER.
- *action* - A synonym for IDENTIFIER.
- *constant* - A synonym for IDENTIFIER.

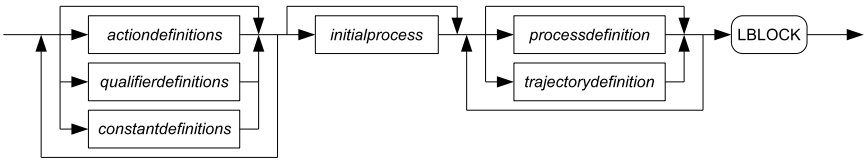
- *parameterdefs* - A sequence of IDENTIFIER separated by commas. Used for defining parameters for a process or trajectory.
- *parameters* - A sequence of expression separated by commas. Used by symbolic trajectory prefixes, parameterized action prefixes and recursions.
- *qualifiers* - A sequence of qualifier separated by commas.
- *actions* - A sequence of action separated by commas.
- *mixedlist* - A sequence of actions and/or qualifiers separated by commas.

Expressions

Expressions are defined according to the usual mathematical sense. The supported operators and their semantics can be found in table 3.2.

The model

A model consists of any number of *action-*, *qualifier-*, or *constant* definitions. These definitions can be placed in order. After these definitions a single initial process has to be defined. Finally the last part of the model consists of one or more *process-* and/or *trajectory* definitions. The makeup of the model is visualized in the following railroad diagram.



Actions, qualifiers and constants

Actions, qualifiers and constants have to be defined separately in the model. This allows the parser to easily check for syntactic and semantic errors in the model. Actions and qualifiers can be defined by respectively the keywords *actions* and *qualifiers* followed by a colon and a list of names.

Constants can be defined by the keyword *constants* followed by a colon and a list of tuples. Each of these tuples consists of a name for the constant followed by an assignment sign (`:=`) and a value for the constant. The list of tuples is separated by commas.

Processes

A process can be defined by the keyword *process* followed by a name for the process, optional parameter definitions, a define symbol (`^=`) and the definition for the process itself. The optional parameters are defined as the *parameterdefs* structure encapsulated by parenthesis.

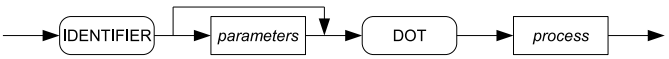
The process definition itself can consist of the following operators with the corresponding syntax. The following descriptions show railroad diagrams to

visualize the syntax of the operators[1]. In the railroad diagrams, besides the global elements, certain keywords are used to denote characters or strings of characters. Tables 3.1 shows these keywords and the character(s) they represent. The rounded rectangles represent a character or string of characters. The normal rectangles are used to represent another part of the hole railroad diagram.

Keyword	Character(s)
LBLOCK	[
RBLOCK	]
DOT	.
PIPE	|
PLUS	+
PLUSPLUS	++
LPAREN	(
RPAREN	)
BACKSLASH	\
KW_DEADLOCK	stop
KW_NEW	new
KW_IN	in
LCURLY	{
RCURLY	}

Table 3.1: ASCII BHPC keywords

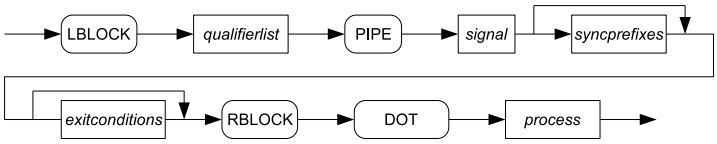
- **(Parameterized) action prefix** An *IDENTIFIER* followed by optional parameters as defined by *parameters*, a dot (`.`) and a process. The *IDENTIFIER* represents the name for the action. This action name should be defined in the model. The syntax for this operator is visualized in the following railroad diagram.




```
Example non parameterized: a.B
Example parameterized: a(5).B
```

- **Symbolic trajectory prefix** A left block character (`[`) followed by a list of qualifiers a pipe character (`|`) a trajectory name, optionally synchronizing prefixes and/or exit conditions. The operator is completed a closing right block character (`]`), a dot character (`.`) and a process. The synchronizing prefixes and exit conditions are a list of expressions preceded by respectively the keywords *conds* and the keyword *exits*. *Signal* is the name of the trajectory and should be defined within the model. The syntax for this operator is visualized in the following railroad diagram.

[1]More information on railroad diagrams can be found on the website http://www.epcedit.com/UserManual/x1176.html



`Example: [h,v | phi exits h > 0 conds v > 0].B`

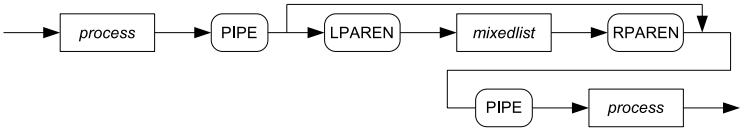
- **Binary choice operator** A process followed by the choice operator (`+`) and another process. The syntax for this operator is visualized in the following railroad diagram.

`Example: a.A + b.B`

- **Binary superposition operator** A process followed by the superposition operator (`++`) and another process. The syntax for this operator is visualized in the following railroad diagram.

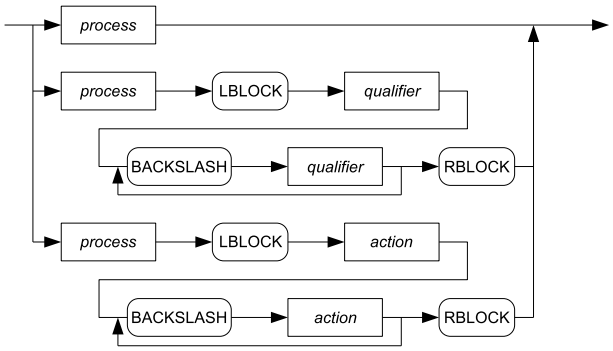
`Example: [h | phi1 exits h > 0].A ++ [h | phi2 exits h > 5].B`

- **Binary parallel operator** A process followed by the parallel operator and another process. The parallel operator consists of a pipe character (`|`), optional syncronizing parameters and a second pipe character. The sychronizing parameters are a list consisting of qualifiers and/or actions as defined by *mixedlist* which should be defined within the model.



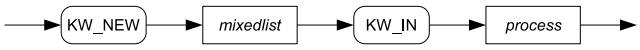
`Example: a.b.A |(c)| c.b.B`

- **Renaming operator** The renaming operator can optionally be placed behind a process. The renaming operator consists of a left block character (`[`) followed by the qualifier or action to be renamed, a backslash character (`\`) and a qualifier or action which holds the new name. The last backslash and qualifier/action can be repeated any number of times according to denote a renaming stack. Finally there is a right block character (`]`) to close the structure. The first qualifier or action representing the original qualifier or action should be defined within the model. The syntax for this operator is visualized in the following railroad diagram.



`Example: a.B[a\c]`

- **Hiding operator** The hiding operator can optionally placed before a process. The hiding operator consists of the keyword *new* followed by a list of actions and/or qualifiers as defined by *mixedlist*, the keyword *in* and a process. The qualifiers are the qualifiers or actions to be hidden in the process, these qualifiers and/or actions should be defined within the model. The syntax for this operator is visualized in the following railroad diagram.

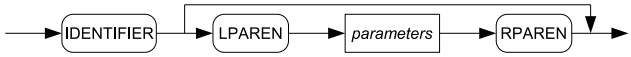


`Example: new a in a.B`

- **Deadlock** The deadlock is simply denoted by the keyword *stop*. The syntax for this operator is visualized in the following railroad diagram.

`Example: a.stop`

- **Recursion** The recursion is an *IDENTIFIER* optionally followed by *parameters* encapsulated by parenthesis. The identifier should be the name of a process defined in the same model. The syntax for this operator is visualized in the following railroad diagram.



`Example: a.ThOn`

- **Guard operator** The guard operator is an expression encapsulated by curly brackets followed by a dot character (`.`) and a process. The syntax for this operator is visualized in the following railroad diagram. In contrast to the mathematical version of BHPC, which uses chevrons for the guard operator, ASCII BHPC uses braces to encapsulate the predicate expression. Chevrons increased the complexity of the design and decreased readability due to the fact that the predicate expression also contains chevrons for equations.


```
Example: {h > 50}.B
```

These operators can be combined in the manor defined in definition 2.4.1. However the syntax for a number of operators has changed. The exact syntax for the operators is as follows.

Initial process

The initial process denotes the starting point for execution/simulation of the model. An initial process is defined by the keywords *initial* and *process* followed by an *IDENTIFIER* and optional parameters. The identifier resembles the name of the process where the execution/simulation should begin, this process should be defined within the model.

Trajectories

A trajectory can be defined by the keyword *signal* followed by a name for the trajectory, optional parameter definitions, a define symbol (`^=`) and the definition for the trajectory itself. The optional parameters are defined as the *parameterdefs* structure encapsulated by parenthesis.

The trajectory definition itself consists of a number of elements. It starts with a list of qualifiers followed by a colon, a time interval. The definition is closed by assignments, differential equations and/or expressions which model the evolution of the trajectory. When assignments are used they must be placed before the differential equations which in turn must be placed before the expressions.

A time interval is denoted as follows

```
(0, expression] => numbersys
```

where

- *Expression* denotes the duration of the trajectory.
- *Numbersys* is a mathematical number system $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$ denoted as respectively $\backslash N$, $\backslash Z$ and $\backslash R$.

Assignments are used to give initial values to the qualifiers of the trajectory. Assignments are denoted as `qualifier(0) := expression` where *qualifier* is the qualifier which needs to be initialized and *expression* is the value to assign to the qualifier.

Differential equations are used to model the evolution of the trajectory. Differential equations are denoted as `der(q) = expression` where q is a qualifier. This structure is equivalent to the mathematical form $\dot{q} = expression$.

3.2.2 Precedence and parsing specifics

Tables 3.2 and 3.3 show the precedences and associativity for respectively the expression operators and the BHPC operators. The precedences are represented by means of numbers. The lower the number the higher the precedence.

Operator	Precedence	Associativity
`-expr`	1	Left to right associative
`expr * expr`	2	Left to right associative
`expr / expr`	3	Left to right associative
`expr + expr`	4	Left to right associative
`expr - expr`	5	Left to right associative
`expr = expr`	6	Left to right associative
`expr != expr`	7	Left to right associative
`expr > expr`	8	Left to right associative
`expr >= expr`	9	Left to right associative
`expr < expr`	10	Left to right associative
`expr <= expr`	11	Left to right associative

Table 3.2: Expression operators

Operator	Precedence	Associativity
$a.B$ $[\varphi].B$ P $\langle Pred \rangle$	1	Left to right associative
$B[\sigma]$	2	Left to right associative
$B \|\|_A^H B$	3	Left to right associative
$\bigoplus_{i \in I} B_i$	4	Left to right associative
$\sum_{i \in I} B_i$	5	Left to right associative

Table 3.3: BHPC operators

Example 3.2.1 (Mixed BHPC operator precedence) The process

```
process T ^= a.stop ++ b.stop + c.stop
```

combines the superposition and choice operator without the use op parenthesis to explicitly define the meaning of the process. Due to the precedences defined for Basic BHPC we can conclude that the example process is equal to the process `(a.stop ++ b.stop) + c.stop` meaning that `a.stop` and `b.stop` will be superposed to each other. The entire construct of the superposed `a.stop` and `b.stop` will be the first choice of the choice operator and `c.stop` will be the second choice of the operator. When this process would be simulated the system would first make a choice between `(a.stop ++ b.stop)` and `c.stop`. Only when the first case is chosen the superposition would be executed. □

Example 3.2.2 (BHPC operator precedence) The process

```
process T ^= a.stop + b.stop + c.stop
```

describes a process with two choice operators. Because the choice operator is implemented in left to right binary form this process is equivalent to the process `(a.stop + b.stop) + c.stop`. During simulation the system will first make

a choice between `(a.stop + b.stop)` and `c.stop`. Only when the first case is chosen it will make the choice between `a.stop` and `b.stop`. □

3.3 Internal representation

3.3.1 Introduction

The internal representation is used to provide the BHPC model in a structured way to other developers who wish to use a BHPC model. Representing the BHPC model in an easy to use, structured format might reduce any ambiguity and decrease development time.

The internal representation is structured conforming the W3 Consortium Extended Markup Language (XML) 1.0 standard [W3X04]. XML is a widely used standard for representing data in a platform independent way. It is well documented and easy to use.

XML schema's are a way to define the structure of an XML document. With an XML schema documents can be checked for compliance to the structure. An XML schema for the BHPC internal representation can be found in appendix C. The remainder of this paragraph describes the format in a textual way complemented with several illustrations. It is assumed that the reader has basic knowledge about the XML standard.

3.3.2 Structure

Each document contains exactly one BHPC model. The model always starts with a `<BHPC>` tag. This tag has several children. It's first child is the `<version>` tag. The version of the internal representation used for the document is placed between an opening and closing version tag. The other children of the BHPC tag are definitions for qualifiers, actions, constants, processes and trajectories. Figure 3.1 shows a schematic overview of the upper level structure of a BHPC model.

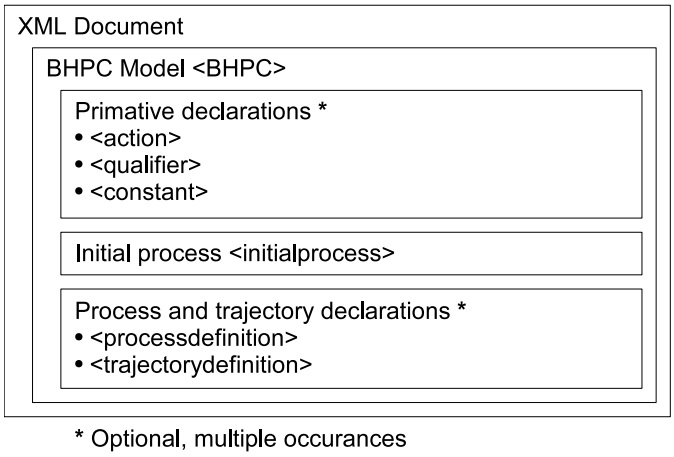


Figure 3.1: Internal representation overview

Primitive declarations

The global qualifiers and actions used in the model are listed by children of the top node. Qualifiers and actions are represented by respectively the `<qualifier>` and `<action>` nodes. The data between the opening and closing tag is the name for the qualifier or action.

Constants can be declared with the Constant tag `<constant>` which has two child nodes. The `<name>` and `<value>` nodes are respectively for the name and value of the constant.

Example 3.3.1 (Internal representation: Primative declarations) The following fragment is taken from the Thermostat example and shows some primitive declarations from the internal representation.

```
<action>on</action>
<action>off</action>
<qualifier>l</qualifier>
<constant>
  <name>K</name>
  <value>0.8</value>
</constant>
<constant>
  <name>h</name>
  <value>50</value>
</constant>
```

Initial process

The initial process is represented by the `<initialprocess>` node. This node has one or more child nodes. The first child is the `name` node that holds the name of the initial process. The other child nodes are optional `<parameter>` nodes that hold parameter expressions to be passed to the initial process. The number of initial processes is restricted to one.

Example 3.3.2 (Internal representation: Initial process) The following fragment is taken from the Thermostat example and shows the initial process node from the internal representation.

```
<initialprocess>
  <name>ThOff</name>
  <parameter>10</parameter>
</initialprocess>
```

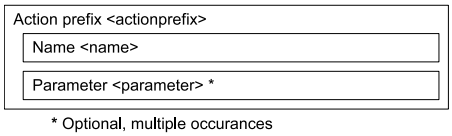
Process definition

A BHPC process is represented by the `<processdefinition>` node. The first child node of the process definition is the `<name>` node that holds the name of the process. This node is followed by optional `<parameter>` nodes that hold any parameters the process might have. The rest of the child nodes depend on the construction of the process and can be one or more of the nodes explained in the section Process nodes.

Process nodes

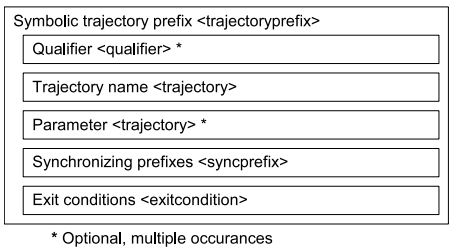
Each of the process nodes represent a BHPC operator. Since several of these operators allow multiple processes to continue these nodes can have on or more child nodes that again contain one or more process nodes. The nodes that are member of this group are the following.

- *Deadlock* This node consists of a single tag `<deadlock/>` and represents the deadlock operator.
- *(Parameterized) action prefix* This node has several child nodes that are represented in the following diagram.



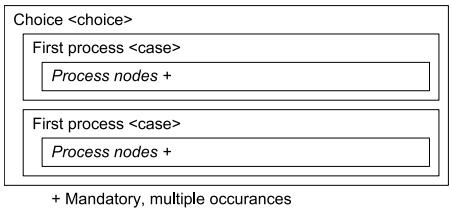
  The process that follows the action prefix is represented by the sister node of the action prefix node.

- *Symbolic trajectory prefix* The child nodes for this node are represented in the following diagram.

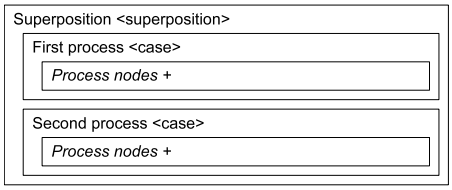


  The process that follows the action prefix is represented by the sister node of the action prefix node.

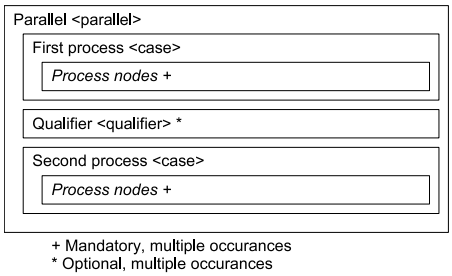
- *Choice* The binary choice operator provides a choice between two processes. Each of these processes are grouped in a `<case>` node. Both case nodes are children of the parent node `<choice>`. The make up of the choice node is displayed in the following diagram.



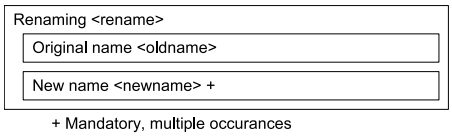
- *Superposition* Like the binary choice operator, the binary superposition has two `<case>` nodes that represent the two processes that the operator operates on. Both case nodes have one or more child nodes from the group of process nodes. The make up of the `<superposition>` node is displayed in the following diagram.



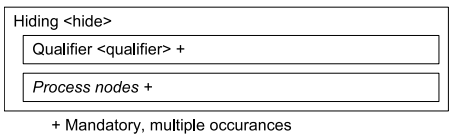
- *Parallel* The binary parallel operator provides a parallel composition between two processes. Like the choice and superposition operator the `<parallel>` node has two `<case>` child nodes that each hold one of these processes. Between these two case nodes there can be optional `<qualifier>` nodes that represent the qualifiers involved in the synchronizing of the two processes. The make up the the parallel node is displayed in the following diagram.



- *Renaming* The `<rename>` node has a child node that holds the original name and one or more child nodes that hold the new names. The make up of this node is displayed in the following diagram.

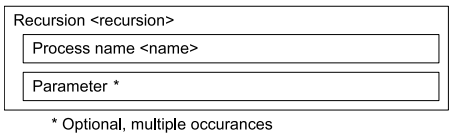


- *Hiding* The hiding operator has one or more `<qualifier>` nodes that hold the names of the qualifiers and/or actions that are to be hidden. These qualifier nodes are followed by one or more process nodes that form the process in which the qualifiers are hidden. All these nodes are children of the `<hide>` node that represents the entire operator. The make up of the hiding operator is displayed in the following diagram.



- *Recursion* The recursion operator is represented by the `<recursion>` node. This node has a `<name>` child node that holds the name of the process that is called. This child node can be followed by the optional `<parameter>`

nodes that hold any parameters that are passed to the recurring process. The make up of the recursion node is displayed in the following diagram.

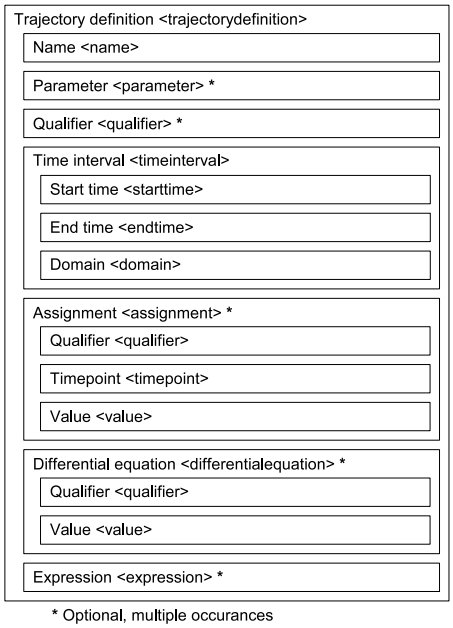


- *Predicate prefix* The predicate prefix holds its expression between its starting `<guard>` node and accompanying closing tag.

Note that restrictions on the possible combinations of the BHPC operators are enforced by the parser and/or checker.

Trajectory definition

The trajectory definition node `<trajectorydefinition>` represents continuous behaviour of the model. This node has several child nodes that represent the various properties of the trajectory definition. Several child nodes of the trajectory definition have child nodes of their own. The complete structure of the trajectory definition node is displayed in the following diagram. These nodes have a simple structure and are therefore considered trivial and need no further explanation.



3.4 Parser generation

The parser is created using the tool ANTLR [ant]. This tool can generate Java or C++ code for lexographical analysers, parsers and tree walkers. For each

one of these objects the tool needs an EBNF specification defining the keywords and syntactic structure of the target language.

These objects can be used sequentially to create an application which can parse a given language and perform certain actions such as checking for semantic errors and generating output.

3.4.1 Encountered problems

This paragraph will describe which problems where encountered during the development of the parser and how these problems where solved.

Parenthesis

The use of parenthesis in the language introduces different meanings of content between the parenthesis. Take for example the following two processes:

```
(a + b).stop
(A + B)
```

The content in the first process is clearly meant to be used as action prefixes and unfolding the process would lead to `a.stop + b.stop`. The other process however is meant to be used as a choice between two recursions and would lead to `A + B`. The difference between these two meanings can be deducted from the dot directly after the closing parenthesis. If this dot is present the content should end with a prefix. And of course in the case of branching between the parenthesis all branches should end with a prefix. When the dot is not present all branches between the parenthesis should be complete processes.

This difference in meaning caused problems with the parsing since this behaviour would lead to branching inside the parser which would lead to quite some redundancy in the parser specification. The current implementation already has troubles with distinction between recursion and an action prefix since both start with an IDENTIFIER. To determine the difference the parser checks for a leading dot after the IDENTIFIER. When this dot is present it is an action prefix and otherwise recursion.

The parenthesis problem is solved in the checker and generator classes. These classes are fit with a boolean variable which tells whether a complete process is expected or not. From the initial iteration from the process definition is variable is initialized to the value true. When, during the parsing of the AST, parenthesis are found it is checked whether these parenthesis are followed by a dot. In the case of a dot the variable for the further recursion is changed to false. In the other case, when the dot is present, the variable for the further recursion is changed to true. The only place where the variable is used is in the recursion rules. When the variable is true the recursion is handled as recursion and when the variable is set to false the recursion is handled as an action prefix.

3.5 Examples

This section will show ASCII BHPC models for the thermostat and bouncing ball examples from examples 2.5.1 and 2.5.2.

Example 3.5.1 (Thermostat) When the thermostat BHPC model shown in example 2.5.1 is rewritten to ASCII BHPC the output will be the following:

```
actions     : on, off
qualifiers  : l
constants   : K := 0.8, h := 50, tempMin := 10, tempMax := 20,
              tempOff := 18, tempOn := 12

initial process ThOn(10)
process ThOff(l0) ^=
        [l | phiOff(l0) exits l >= tempMin, l < tempOn].on.ThOn(l)
process ThOn(l0)  ^=
        [l | phiOn(l0)  exits l >= tempOff, l < tempMax].off.ThOff(l)
signal phiOff(l0) ^=
        {l: (0, t] -> R | l(0) := l0,der(l) = -K * l}
signal phiOn(l0)  ^=
        {l: (0, t] -> R | l(0) := l0,der(l) = K * (h - l)}
```

Example 3.5.2 (Bouncing ball) When the bouncing ball BHPC model shown in example 2.5.2 is rewritten to ASCII BHPC the output will be the following:

```
qualifiers : h, v
constants  : c := 0.9, g := 9.8
actions    : bounce

initial process BB(10, 0)
process BB(h0, v0) ^=
        [h, v | phi(h0, v0) exits h <= 0, v < 0].bounce.BB(0, -c * v)
signal phi(h0, v0) ^=
        {h, v : (0, t] -> R | h(0) := h0, v(0) := v0,
        der(h) = v, der(v) = -g, h >= 0}
```

CHAPTER 4

Modelica and hybrid systems

4.1 Introduction

Modelica is an object-oriented modelling language for complex physical systems developed by the Modelica Association [modb]. The Modelica Association is a non-profit organisation founded for this purpose. Its aim is to facilitate simulations for complex multi-domain applications.

This chapter will describe the basics of Modelica. After the basics we will continue with a description of the StateGraph library followed by a description of how the StateGraph library, together with some additional Modelica code can be used to simulate hybrid automata.

4.2 Language

This paragraph will describe the language details of the Modelica language. Since the language is extensive in size we will focus on the basics of the language and the parts used for the BHPC translator.

4.2.1 Basic language elements

Physical system can be described by a system of equations; differential equations, algebraic or discrete ones[1]. Modelica supports these equation systems directly taking away the need to convert these systems to an algorithm. When this functionality is used these equations will be placed in the *equation* section of the model. Typically a model starts with the keyword `model` followed by the name for the model. After the model name variables can be declared followed by an equation and/or algorithm section.

[1] A differential equation is an equation in which the derivatives of a function appear as variables [Wikb].

In the declaration section components can be declared by an optional specifier, the component type, the name for the component, an optional modification and finally the delimiter `;`. Modelicas basic component types include the Boolean, Integer, Real and String. These basic types can be used to create models of more complex types such as a electrical resistor which can then be used by other models. A component declaration can be preceded by a specifier like `constant` or `parameter`. The constant specifier indicates that the component is constant and cannot be changes. The parameter specifier indicates that the value of the quantity is constant during simulation runs. It can be modified when a component is reused and between simulation runs. The component can be followed by a modification to change the value of the component or its attributes.

A few basic declarations:

```
Real u, y(start=1);
parameter Real T=1;
```

The first line shows the declaration of two components `u` and `y` with the type real. The component `y` is followed by a modification which sets the initial value to 1; The second line shows the declaration of a parameter `T` with the type Real and another form of modification setting the default value of the component to 1.

Example 4.2.1 (Basic Modelica Model) The following model describes the behaviour of an electrical resistor.

```
model Resistor
  Pin p, n;
  parameter Real R "Resistance";
equation
  R*p.i = p.v - n.v;
  n.i = p.i;
end Resistor;
```

Equations are composed of expressions on both the left and right hand side of an equality statement. The simulation will, in most cases, manipulate the equations symbolically to determine their order of execution and which components in the equation are inputs and which are outputs. Time derivative is denoted by the operator `der()`.

In order to facilitate reuse Modelica offers a class concept similar to that of programming languages. It is used for many purposes such as model components, connection mechanisms, parameter sets, input-output blocks and functions. In order to make Modelica classes easier to read and to maintain special keywords have been introduced for such special uses. The keywords `model`, `connector`, `record`, `block`, `function`, `type` and `package` apply certain restrictions to the classes like records are not allowed to contain equations.

The Pin types in example 4.2.1 are of the special type `connector`. This type can be used with the `connect` operator which specifies the interaction between two components. The connect operator generates equations taking into account what kind of quantities that are involved. The connector class can be used to define these quantities.

Example 4.2.2 (Electrical pin) An electrical component often has two pins to connect to other components. Such a pin is described by the following connector class.

```
connector Pin
  Voltage      v;
  flow Current i;
end Pin;
```

A connection `connect (Pin1, Pin2)` connect the two pins such that they form one node. This implies two equations, namely `Pin1.v = Pin2.v` and `Pin1.i + Pin2.i = 0`. The first equation indicates that the voltages on both branches connected together are the same. The second equation corresponds to Kirchhoff's current law stating that the currents sum to zero at a node. The sum-to-zero equations can be generated with the prefix `flow`. □

4.2.2 Inheritance

Just like in any other object oriented programming language Modelica offers a way to inherit the properties from another class and extends its behaviour. This can be done by writing the keyword `extends` followed by the name of the parent class and the delimeter `;` as the first line in the declaration block. The new class will then inherit all properties of the parent class. Multiple inheritance is also supported and can be achieved by placing several extends statements.

Example 4.2.3 (Modelica inheritance) The following example shows an Inductor model which has the OnePort class as parent.

```
model Inductor
  extends OnePort;
  parameter Real L(unit="H")
equation
  L*der(i) = v;
end Inductor;
```

It is also possible to denote a class for reuse only. Such a class can not be instantiated and can only be used in an extends statement. This notion is similar to abstract classes in many programming languages. By placing the keyword `partial` in front of the class keyword (or the model, connector, etc keyword) a class is declared for reuse only.

Example 4.2.4 (Modelica partial model) The following example shows the model OnePort which is defined as a partial model and can only be used in an inheritance structure.

```
partial model OnePort
  Pin p, n;
  Voltage v;
equation
  v = p.v - n.v;
  p.i + n.i = 0;
end OnePort;
```

4.2.3 Repetitions, algorithms and functions

Modelica also offers support for loop constructions using the `for` and `while` constructs commonly present in ordinary programming languages. We assume the reader is familiar with the concepts of these constructs and will not go into detail on these operators. Examples 4.2.5 and 4.2.6 show the syntax for these operators and how they can be used.

Example 4.2.5 (Modelica for loop) This example shows a Modelica code fragment running a for loop from 1 to n.

```
for i in 1:n loop
  xpowers[i+1] = xpowers[i]*x;
end for;
```

Example 4.2.6 (Modelica while loop) This example shows a Modelica code fragment running a while loop until the condition `i < 10` is not longer satisfied.

```
while i < 10 loop
  i = i + 1;
end while;
```

The for operator as well as the while operator can be used in the equation section as the algorithm section. This last section is used when it is more convenient to use assignment statements in stead of the equation mechanism. Especially for digital controllers it is more natural to model with ordered assignment statements since the actual controller will be implemented in such a way.

A Modelica algorithm is a function in the mathematical sense. That is, whenever such an algorithm is used with the same inputs, the result will be exactly the same. If a function is called during continuous integration this is an absolute prerequisite. Otherwise the mathematical assumptions on which the integration algorithms are based on, would be violated.

One of the class specializations is a function. A function has input values and output values. The components for the input and output can be marked with the keywords `input` and `output`. A function has the restriction that no internal states are allowed.

Example 4.2.7 (Function with algorithm section) This example shows a Modelica function with multiple inputs and outputs. The calculations are done in an algorithm section.

```
function Circle
  input  Real angle;
  input  Real radius;
  output Real x;
  output Real y;
algorithm
  x := radius*Modelica.Math.cos(phi);
  y := radius*Modelica.Math.sin(phi);
end Circle;
```

Modelica also allows external functions which are defined outside the Modelica language. The function itself can be defined in any language supported by the simulator like e.g. C code. The body of an external function is marked with the keyword `external` as shown in example 4.2.8

Example 4.2.8 (External function) This example shows an external function.

```
function log
  input  Real x;
  output Real y;
external
end log;
```

4.2.4 Conditional models

Modelica has a number of language constructs to create conditional models. The `if-then-else` statement can be used to change the behaviour of a modal according to certain conditions. The if-then-else statement can be used with two different syntax styles. One is the syntax style used by almost all imperative programming languages and written as

```
"if" condition "then"
  statement(s)
"else"
  statement(s)
"end if"
```

The other syntax is much like the conditional expression operator in the language C/C++. This operator is typically used on the right hand side of an assignment statement to

Example 4.2.9 (Modelica if-then-else statement) This example shows the use of the if-then-else statement.

```
block Controller
  input  Boolean simple=true;
  input  Real e;
  output Real y;
protected
  Controller1 c1(u=e, enabled=simple);
  Controller2 c2(u=e, enalbed=not simple);
equation
  y = if simple then c1.y else c2.y;
end Controller;
```

The output of the model is determined by the input parameter *simple*. □

The actions to be performed at events are specified by a `when` statement. The statements in the body of the when statement are activated once every time instantaneously when the condition is satisfied. These statements will not be executed again until, after a period during which the condition is no longer satisfied, the condition is again satisfied. A special operator can be used in combination with the when statement. In certain conditions it is necessary to change the value of component during integration. However when the value is changed in the ordinary fashion integration will fail. The `reinit` operator effectively removes the previous equation defining statevariable and adds a new equation statevariable with the value provided to the operator. Neither the number of

variables nor the number of equations are changed during this process. Modelica's *single assignment rule* is therefore not violated. The single assignment rule dictates that the number of equations is the same as the number of assignments. Anything else would create ambiguity and the solver would not be able to determine what value a variable should have at a certain point in time.

Example 4.2.10 (Modelica when statement) This example shows the use of the when statement.

```
equation
  when x < 5 then
    reinit(x, 0);
  end when;
```

When the condition `x < 5` is satisfied the statement `y = 7 - 2 * x` is exectuted. □

When statements in equations sections are allowed to have only one branch. However, in algorithm sections, `elsewhen` branches can be added. This is usefull to order priorities between discrete actions. When multiple conditions become true as the same event instant only the condition with the highest priority will be executed.

Example 4.2.11 (Modelica elsewhen statement) This example shows the use of the elsewhen statement.

```
algorithm
  when h1 < hmax then
    open := true;
  elsewhen pushbutton then
    open := false;
  end when;
```

The condition `h1 < hmax` has a higher priority as the condition `pushbutton`. When both conditions become true at the same event instant only the statement `open := true` will be executed. □

4.2.5 Annotations

In addition to the mathematical model with variables and equations Modelica offers language constructs to represent the model graphically. The language constructs are called annotations and are typically used in three separate contexts:

- Annotations associated with a component typically specify the position and size of the component.
- Annotations of a class specify the graphical representation of its icon, diagram and common properties such as the local coordinate system.
- Annotations associated with connections typically specify route and colour of the connection line.

Annotations are usually used by various modelling tools to represent the model in a graphical manor. Typically components can be easily dragged and dropped in the model and connections can be easily made by visually connecting the pins of one component to that of another. This approach makes modelling much easier and faster.

Example 4.2.12 (Modelica annotations) This example shows a simple electrical circuit consisting of a power source and a resistor. The components and connections have annotations specifying the place of the components and the route of the connections within the model.

```
model Unnamed
  annotation (uses(Modelica(version="1.6")), Diagram);
  Modelica.Electrical.Analog.Sources.SineVoltage
    SineVoltage1(V=220, freqHz=50)
     annotation (extent=[-74,8; -54,28]);
   Modelica.Electrical.Analog.Basic.Resistor
     Resistor1(R=1000)
    annotation (extent=[-76,48; -56,68]);
equation
  connect(SineVoltage1.p, Resistor1.p) annotation
    (points=[-74,18; -90,18;
             -90,58; -76,58],
     style(color=3, rgbcolor={0,0,255}));
  connect(Resistor1.n, SineVoltage1.n) annotation
    (points=[-56,58; -40,58;
             -40,18; -54,18],
     style(color=3, rgbcolor={0,0,255}));
end Unnamed;
```

[moda]

4.3 StateGraph library

Since Modelica version 2.1 the StateGraph library is part of the standard Modelica package and can be used to model discrete and reactive systems in a convenient way. A StateGraph is defined as an enhanced finite state machine. Together with some additional Modelica code this library can be used to simulate hybrid automata. However since neither the StateGraph library or Modelica has been formally defined it is not clear to what extent this combination implements hybrid automata. This paragraph will describe the StateGraph library and its components. The next paragraph will describe how the library can be used to simulate hybrid automaton with additional Modelica code.

Steps

A step represents a state within the finite state machine. When a step is active the Boolean variable *active* is true, and false otherwise. InitialStep objects are used to denote the starting point for the system. At the initial time all regular steps are deactivated. The InitialStep objects are activated at the initial time. InitialStep objects are represented by a double box while regular steps are represented by a single box.

Transition

A step can have multiple incoming and outgoing transitions. Transitions are used to change the state of the finite state machine. When the transition is active the previous state will be deactivated and the next state will be activated. Transitions have a Boolean expression *condition* which acts as a guard on the transition. Whenever this condition is true the transition can fire. Transitions also have a timer which is disabled by default, when this timer is activated the firing of the transition is delayed by the amount of time indicated by the *waitTime* variable. This property can be used to counteract problems arising under certain conditions. This is explained later in chapter 4.5.

In the case when multiple transitions can be fired the transitions with the highest priority is chosen from the set of active transitions. The priority is determined by its place in the array of outgoing transitions.

Parallel

The Parallel components can fire multiple transitions in parallel. Although it would seem that this component can be easily used to simulate the parallel BHPC operator it comes with quite some restrictions. Since Modelica does not support dynamic allocation of objects recursion can not be supported. A process with two parallel recursive calls to the same process will result in an untranslatable situation. The parallel component also requires the multiple paths that it starts to come together at some point. The last transitions of these paths can only be fired simultaneously. Due to these restrictions the parallel BHPC operator is not implemented during this project, although it might be subject for future work.

4.4 Simulating hybrid automata

This paragraph will describe how the StateGraph library together with some additional Modelica code can be used to simulate a hybrid automaton. This is done by relating the elements of the hybrid automaton to components of the StateGraph and/or additional Modelica code.

A hybrid automaton [Hen96] is a formal model for a mixed discrete-continuous system. Hybrid automata are popular ways to model and analyse hybrid systems. A hybrid automaton combines discrete and continuous behaviour. Discrete changes are described by transitions which are decorated with action names, guards and assignments. The guards define when a transition is active and can be taken, assignments define changes of the continuous state made by the transition. The action names are used as references in synchronization. Continuous behaviour is described by locations, flow conditions define continuous changes of the continuous state and are usually provided in the form of differential equations and invariants. The invariants restrict evolution in the location.

Definition 4.4.1 (Hybrid Automaton) A hybrid automaton is a collection $H = (X, L, Init, Inv, f, E, Guard, Assign, \Sigma)$ where:

- $X \subseteq \mathbb{R}^n$ is the continuous state space and $x = (x_1, x_2, ..., x_n)$, where $x_i \in \mathbb{R}, i = 1, n$, represents the continuous dynamics.

- L is a finite set of locations.
- $Init \subseteq L \times \mathbb{R}^n$ is a set of initial location state pairs.
- $Inv : L \to 2^X$ assigns to each location l an invariant to be satisfied by the state x while in the location l.
- $f : L \to (X \to \mathbb{R}^n)$ assigns to each location l a continuous vector field f_1 such that the state $x \in X$ should satisfy $\frac{d}{dt}x = f_1(x)$.
- $E \in L \times L$ is the set of transitions, also called switches.
- $Guard : E \to 2^X$ assigns to each transition a guard that has to be satisfied by the state x if the transition is taken.
- $Assign : E \to (X \to X)$ assigns to each transition an assignment that may alter the state x when the transition is taken.
- Σ is a set of transition labels. We assume labelling function $lab : E \to \Sigma$ and refer to transitions by their labels (assuming uniqueness). □

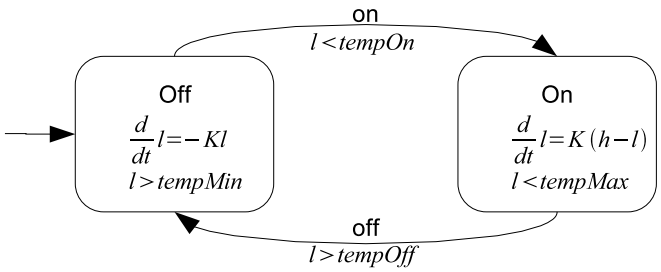


Figure 4.1: Thermostat hybrid automaton

Example 4.4.1 (Thermostat) A hybrid automaton of the thermostat from example 2.5.1 is shown in figure 4.1. The off location of the automaton resembles the states in which the heating is turned off. In this location the temperature evolves according to the equation $\frac{d}{dt}l = K(h-l)$. When the temperature is below *tempOn* the transition leading to the On location will become active. From this point the system non-deterministically chooses to take the transition or remain in the Off location. However when the system repeatedly does not take the transition ultimately the invariant $l > tempMin$ will become true forcing the system to take the transition.

When the transition is taken the action on will occur and the automaton will jump to the On location where the temperature evolves according to the equation $\frac{d}{dt}l = -Kl$. The transition to the Off state is essentially the same as the transition to the On location and is not explained further. □

4.4.1 Basics

The basic structure of the hybrid automaton is a directional graph (V, E) where the node $v \in V$ represents a location and the edge $e \in E$ represents a transition.

This structure can be easily represented by the StateGraphs step en transition components where every $v \in V$ is represented by a step and every $e \in E$ is represented by a transition. The set of initial locations $Init \subseteq V$ can be represented by using the StateGraphs initial step component instead of the regular step.

4.4.2 Decorations

The hybrid automaton decorations are, with the exception of the guard, not present in the StateGraph library and therefore need to be simulated by additional Modelica code. The following subsections describe for each decoration how this can be achieved.

Guards

The guard statements $g \in Guard$ can be represented by the *condition* property of the StateGraphs transition component. When a guard statement is absent the default value *true* is used which will have the same behaviour namely that the transition is always enabled.

Transition labels

Transition labels are used for synchronization between parallel automata. As described in chapter 5.3 simulation of parallel composition is not feasible in the Modelica language.

Since the discrete actions, other then synchronization, have no real execution during simulation it is sufficient to show the firing of the actions. Since the output of Modelica is in the form of a plot we have chosen to assign an Integer variable for each action which will be incremented on each firing of the action. Then the plot will show when and how many times the action was fired. The StateGraph library provides no support for discrete actions. However the library has the property that the automaton will always leave a location as soon as an active transition is available. By checking if a transition is enabled and if the location preceding the transition is currently active we can detect whether a transition is taken or not. When such a transition is taken and an action is present on the transition the variable assigned to the action can be increased. This functionality is represented by the following Modelica code.

```
when( previousLocation.active and transition.Condition )
  actionVariable = actionVariable + 1;
end when;
```

In the case when multiple transitions are enabled the StateGraph library uses a priority system to determine which transition to take. In contrast to the StateGraph library the above code will fire the transitions on all active transitions. However this will not be a problem since the hybrid automaton will be constructed such that only a single transition will be active at all times. This concept is explained in the section *Non-determinism*.

Invariants

Invariants in combination with guards provide a time window in which a transition can be taken in a non-deterministic manor. Since Modelica does not

support non-determinism an alternative has been devised. This alternative is described in the section *Non-determinism*.

Example 4.4.2 (Thermostat) Take the off location from the hybrid automaton shown in figure 4.1. The invariant $l > tempMin$ will force taking the transition at the point when this condition is met. However the guard $l < tempOn$ on the transition to the On state will become active when the guard condition is met. When $tempMin \neq tempOn$ these conditions are met at different points in time creating a window in which the transition can be taken non-deterministically. The simulation of the invariant, when entering the location, will determine a condition $l = k$ such that $tempMin < k < tempOn$ and replace the guard on the transition with the guard $l = k$. □

Continuous vector fields

Continuous vector fields can be simulated by creating an *if-then-else* branching structure for each qualifiers. The conditions in the branch are made up from the locations in the automaton. When a qualifier is influenced by some state this location is placed as a condition in the branching. In all other cases, where the current state does not influence the qualifier the value of the qualifier will remain the same. Such a branching structure looks like the following pseudo code:

```
der(qualifierX) =
  if state1.active then
    expression for state1;
  else if state3.active then
    expression for state3;
  else
    0;
```

The branches are placed in the equations section of the model.

Assigns

Assignments can be placed as a decoration on a transition. When the transition fires the assignment is to be executed. The firing of this transition can be easily detected by checking if the previous location is active and the transition is enabled. Due to the lack of non-determinism in Modelica and the semantics of the StateGraph library it is guaranteed the transition will be taken when those conditions are satisfied. In order to avoid any consistencies in the integration process the assignments are to be executed using Modelica's *reinit* operator.

Example 4.4.3 (Assignment simulation) The following Modelica code detects when the transition `ThOn_Transition1` fires and reinitializes the qualifier `l` to the value stored in the variable `phiOn__l0`.

```
when ThOn__Step1.localActive and ThOn__Transition1.condition then
  reinit(l, phiOn__l0);
end when;
```

These detections are placed in the algorithm section of the model.

Non-determinism

Modelica's single assignments rule does not allow the construction of any non-determinism in the code. The non-deterministic behaviour of BHPC is therefore quite hard to simulate. One way to achieve a crude simulation of non-determinism is to use a random function to create variation on the placed where non-determinism is required.

The random function can be obtained by creating an external function which uses the C/C++ random function. This external function looks as follows:

```
function random
  output Real r;
  external "C";
  annotation(
    Include="#include <stdlib.h>                  \n
    double random(void)                           \n
    {                                             \n
      srand(time(NULL));                          \n
      return (double)rand() / (double)RAND_MAX;   \n
    }"
  );
end random;
```

The function above defines a variable `r` of the type Real which receives the output of the function. The function itself is defined directly in an annotation rather than an external file. The code initializes the random generator and returns a value between 0 and 1.

The above code is not enough to obtain a random function in Modelica. The modelica simulator, Dymola, that is used for this project optimizes the code and makes the result of the function constant over time. In order to counter this optimization a variable `rnd` is introduced which receives a value influenced by the output of the random function.

The function to determine the value for this variable has been set to the following

$$rnd = (sin(time/(random()/100) + random()) + 1)/2)$$

Due to time constraints this simple function is used to simulate non-determinism. However to obtain more reliable results this function should be replaced by another with better random results.

There are two BHPC operators that exhibit non-deterministic behaviour, namely the choice and trajectory prefix operators. The method of simulation for these two operators are described in the following chapters.

Multiple outgoing transitions

Non determinism for multiple outgoing transitions is obtained by making the conditions for the transitions dependent on the `rnd` variable. Since the value of this `rnd` variable changes over time the transitions are continuously disabled and enabled during the simulation. The situation when more than one transitions is enabled is avoided by making the conditions mutually exclusive. For two transitions the exact conditions will be "$rnd >= 0$ *and* $rnd < 0.5$" and "$rnd >= 0.5$ *and* $rnd < 1$" for respectively transitions 1 and 2. Since the range

of possible values is evenly distributed among the transitions, every transition will have an equal change of being enabled.

Invariants with guards
Invariants in combination with guards provide a time window in which a transition can be taken in a non-deterministic manner. This behaviour will have to be simulated with custom Modelica code since the StateGraph library will take a transition as soon as it becomes enabled.

The only way to let the transition fire some time after it is enabled is to delay the point where the transition will be enabled. This can be achieved by changing the guard to a random value that lies within the regions of values determined by the expression of the guard and the expression of the invariant.

Example 4.4.4 (Non-determinism in a thermostat) The thermostat from example 2.5.1 has a non-deterministic exit from its two trajectory prefixes. When the temperature in the room is between the minimum and on temperatures the heating can be turned on. When the temperature is between the maximum and off temperatures the heating can be turned off. This example will show the code that handles the first situation where the thermostat is turned on.

The following code shows the declaration of the transition that follows the location where the thermostat is in the off state.

```
Modelica.StateGraph.Transition ThOff__Transition3(
  condition=l > NonDeterministic__0 - 0.0001 and
  l < NonDeterministic__0 + 0.0001);
```

The guard for the transition has been set to the condition that the temperature is more or less equal to the temperature defined by the variable `NonDeterministic__0`. Modelica does not allow the equal operators on variable of the type Real, therefore a small range for the variables value is used.

The value of the variable `NonDeterministic__0` is set by the following code that is placed in the equation section of the model.

```
NonDeterministic__0 = getvaluebetween(tempMin, tempOn, rnd);
```

The value is set to a value between the minimum temperature and the on temperature. The exact value is determined by the function `getvaluebetween`. The contents of this function is the following.

```
function getvaluebetween
  input Real min;
  input Real max;
  input Real rnd;
  output Real result;
algorithm
  result := min + ((max - min)*rnd);
end getvaluebetween;
```

The function chooses a random value by using the `rnd` variable that holds a random value. □

4.5 Restrictions

At some point in every model with a recursion loop time must advance. The only operator during which time expires is the symbolic trajectory prefix. When all the symbolic trajectory prefixes in the model contain exit conditions which allow the system to exit the trajectory no time will advance in the entire model. When no time expires during simulation, the simulation cannot end. When such a situation occurs the Dymola simualtor will output the following message:

```
ERROR: Finding consistent restart conditions failed at time:  0
```

This problem can probably be corrected by changing the exit conditions. This restriction is actually quite useful since BHPC does not allow trajectories in which time does not advance. However when the model contains multiple symbolic trajectory prefixes and time does advance in one of them the system is able to simulate it and might produce unwanted results.

An alternative for fixing this problem is to active the timer of one of the transitions and set the delay at a value greater then 0.

4.6 Conclusions

Modelica is a rich language for modelling complex physical systems. There are a large amount of libraries available for Modelica of which one contains the StateGraph library. This library is designed for modelling and simulation of discrete and reactive systems. With the addition of some auxiliary Modelica code this StateGraph library can be used to simulate hybrid automata. However there are some restrictions to the hybrid automata that can be simulated using this construction.

CHAPTER 5

The translator

5.1 Introduction

This chapter will describe the translator in detail. First we define a feasible subset of BHPC which on which the algorithm will operate. After this definition we will go into detail about the algorithm. The algorithm takes a BHPC specification and iterates through its operators. Each operator is translated into hybrid automaton fragments by means of mappings. When combined these fragments form a hybrid automaton. This hybrid automaton is then modelled in Modelica by using the StateGraph library and some additional Modelica code. Finally we will show some examples followed by conclusions.

5.2 Restrictions in simulation

Unlike other object oriented programming languages Modelica does not support dynamic instantiation of objects. This makes it hard or even infeasible to simulate a parallel composition. Take for example the following BHPC process

$$P \triangleq P \,||\, P$$

During simulation the process P will be instantiated. Next the parallel operator will be executed resulting in a total of two P processes. When the parallel operator of these two processes are executed the system will have a total of 4 P processes. Every iteration the number of processes will double. In order to simulate this behaviour dynamic instantiation of objects is required.

Apart from the restrictions mentioned before the StateGraphs parallel component has the property that every branch of a parallel execution will have to be finished before it can continue making it hard to even simulate simple parallel composition.

5.3 Subset of the BHPC language

Due to the restrictions mentioned in section 5.2 a feasible subset of the BHPC language is chosen. Although the parallel operator is not implemented the chosen subset suffices for an interesting set of operators. This subset is defined as follows and will be referred to as Basic BHPC.

Definition 5.3.1 (Basic BHPC) The subset of the BHPC operators subjected to translation to Dymola specifications are defined in the following Backus Naur form notation.

$$B ::= 0 \mid a.B \mid [t_1, ..., t_m \mid \Phi \downarrow Pred \Downarrow Pred_{exit}].B \mid$$

$$\langle Pred \rangle.B \mid B_1 + B_2 \mid P$$

- 0 is a deadlock.
- $a.B$ is an action prefix where $a \in \mathcal{A}$ is a discrete action name and B is a process. It denotes discrete transitions in the hybrid automaton.
- $a(v : V).B(v)$ is a parameterized action prefix where $a \in \mathcal{A}$ is a discrete action name, $v \in V$ is a parameter and B is a process. It denotes discrete transitions in the hybrid automaton.
- $[t_1, ..., t_m \mid \Phi \downarrow Pred \Downarrow Pred_{exit}].B$ is a symbolic trajectory prefix where $t_1, ..., t_m$ are trajectory qualifiers, Φ is the set of trajectories, $Pred$ and $Pred_{exit}$ are sets of predicates and B is a process.
- $\langle Pred \rangle.B$ is a guard statement where $Pred$ is the predicate to be evaluated and B is a process.
- $B_1 + B_2$ is a choice where B_1 and B_2 are processes.
- $B \triangleq P$ is a recursive equation. □

5.4 Algorithm

The translation algorithm converts a BHPC model into Modelica code which makes use of the StateGraph library as described in chapter 4.3. First we will show how the algorithm constructs a hybrid automaton from a BHPC specification. After that we will show how the hybrid automaton can be translated to Modelica by using the StateGraph library and additional Modelica.

5.4.1 Translating BHPC to a hybrid automaton

The algorithm converts a BHPC model with mappings of a hybrid automaton fragment on the BHPC operators. The used notion of a mapping is a fragment of a hybrid automaton which exhibits the same behaviour as the BHPC operator it corresponds with.

The algorithm is recursive and follows the order of the internal representation it is provided as input. In turn the internal representation has the same structure as the abstract syntax tree constructed from the BHPC specification.

Algorithm 5.4.1 (Abstract translation) The following abstract algorithm creates a hybrid automaton from a BHPC specification. The algorithm makes use of a number of dummy cases. Under some conditions these dummy steps can be removed, however for readability these optimisations are not included in the abstract algorithm.

```
translateBHPC(Process B)
{
   Add Location L to Hybrid Automaton;
   Add Transition T to Hybrid Automaton;
   doProcess(B, T);
}

doProcess(Process B, Transitions T)
{
   case Action prefix AP :
      Add action to trajectory T;
      doProcess(B\AP, T);
   case Trajectory prefix TP :
      Add Location L to Hybrid Automaton;
      Connect all transitions in T to L;
      Set trajectory evolutions of TP to L;
      Add outgoing transition T1 to L;
      Set exitconditions of TP as guard to T;
      doProcess(B\TP, T1);
   case Guard prefix GP :
      Add dummy Location L to Hybrid Automaton;
      Connect all transitions in T to L;
      Add outgoing Transition T1 to L;
      Set predicate of GP as guard to T1;
      Add outgoing Transition T2 to L;
      Set negated predicate of GP as guard to T2;
      Connect T2 to Deadlock location;
      doProcess(B\GP, T1);
   case Choice C :
      Add dummy Location L to Hybrid Automaton;
      Connect all transitions in T to L;
      Add outgoing Transition T1 to L;
      doProcess(C.Alternative1, T1);
      Add outgoing Transition T2 to L;
      doProcess(C.Alternative2, T2);
      if(B\C not empty) doProcess(B\C, {T1, T2});
   case Recursion R :
      Connect all transitions in T to First Location of
              target process;
   case Deadlock D :
      Connect all transitions in T to Deadlock location;
}
```

The algorithm processes a BHPC specification operator by operator and expands the Hybrid Automaton when necessary. The operators which are followed by

another operator always leave a dangling transition. A dangling transition is a transition for which the destination is not defined yet. The destination for the dangling transition will be determined during the next iteration. The operators which are not followed by another operator will connect the last dangling transition to a location. □

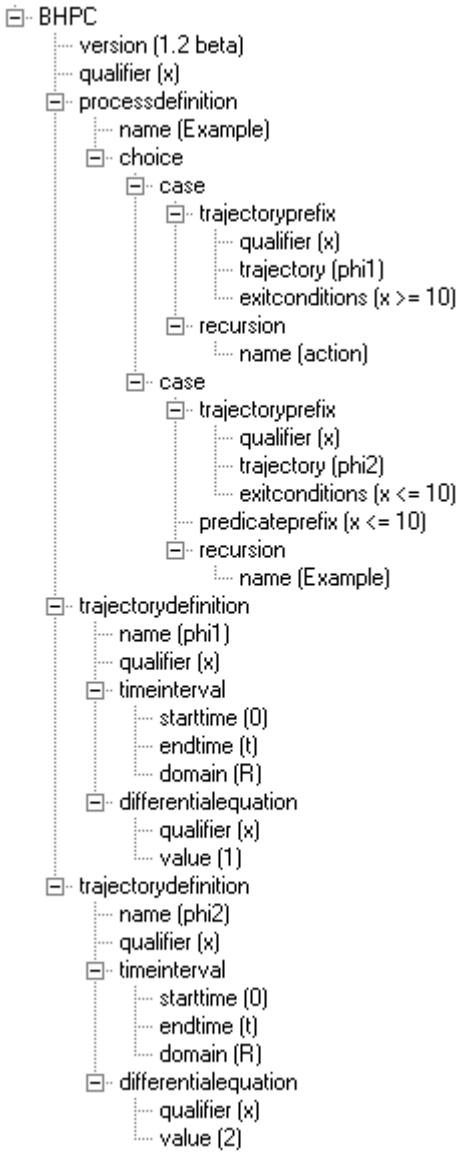


Figure 5.1: Internal representation for example 5.4.1

Example 5.4.1 Let B be the following BHPC specification

$$Example \triangleq [x \mid \Phi_1 \Downarrow l \geqslant 10].action.stop + [x \mid \Phi_2 \Downarrow l \leqslant 10].\langle l \leqslant 10\rangle.Example$$

During the translation step the Internal Representation will be built as shown in figure 5.1.

- Step 1: First the algorithm will create a dummy state and dangling transition which the operators can use to expand the hybrid automaton.
- Step 2: The first iterator of the *doProcess* function will process the choice operator. It will add another dummy location to the automaton and will execute two recursive calls, one for each alternative.
- Step 3: The first alternative from the choice operator in step 2 is the trajectory prefix $[x \mid \Phi_1 \Downarrow l \geqslant 10]$. The algorithm will create a location for the trajectory prefix and will set the evolutions to those defined in Phi_1. It will also add an outgoing transition to this location with the exit conditions $l \geqslant 10$ as guard on the transition. It will then do a recursive call for the operator which follows this trajectory prefix.
- Step 4: The next operator to be processes is the action prefix *action*. The algorithm will put the action on the dangling transition it received as input. Next it will do a recursive call for the operator which follows the action prefix.
- Step 5: The last operator of the first alternative of the choice operator is the deadlock operator. It will connect the dangling transition to the deadlock location of the hybrid automaton.
- Step 6: The first alternative of the choice operator is completed. The algorithm will continue by adding a second outgoing transition to the dummy location which was added for the choice operator. This transition will be used to connect the second alternative of the choice operator.
- Step 7: The first operator of the second alternative is the trajectory prefix $[x \mid \Phi_2 \Downarrow l \leqslant 10]$. The algorithm will create a location for the trajectory prefix and will set the evolutions to those defined in Φ_2. It will also add an outgoing transition to this location with the exit conditions $l \leqslant 10$ as guard on the transition. It will then do a recursive call for the operator which follows this trajectory prefix.
- Step 8: The next operator to be processed is the guard operator $\langle 1 \leqslant 10 \rangle$. The algorithm will add a dummy location to the automaton and will connect the last dangling transition to this location. It will then add two outgoing transitions to this dummy location. One will have the guard $\langle 1 \leqslant 10 \rangle$ and the other will have the negation of this guard. The last location with the negated guard will be connected to the deadlock location. The other will be provided to the next operator through the recursive call that follows.
- Step 9: The last operator to be processed is the recursive call *Example*. This will connect the last dangling transition to the first location of the process *Example*.

- Step 10: All alternatives of the choice operator are processed. The choice operator is not followed by other operators and therefore the algorithm will terminate with a hybrid automaton as result. □

In the algorithm there is specific code for each operator, these code fragments can be considered mappings of hybrid automaton fragments on the BHPC operators. When these fragments are combined they will form the hybrid automaton for the BHPC specification.

5.4.2 Translation mappings

The operators of Basic BHPC can be separated into two groups, namely those operators that are followed by another operator and the operators which are not followed by another operator. Both groups are places in sets by the following definitions.

Definition 5.4.1 (Intermediate operators) The set of operators which are recursive of nature and thus are always followed by another operator consist of the operators

$$BHPC_{intermediate} = \{a.B, [\varphi].B, B_1 + B_2, \langle Pred \rangle.B\}$$

Definition 5.4.2 (Ending operators) The set of operators which are not recursive of nature and thus cannot be followed by another operator and as a result mark the end of a specification consists of the following consists of the operators

$$BHPC_{ending} = \{0, P\}$$

Definition 5.4.3 (Dummy location) A dummy location is a location in which time does not pass. This condition is obtained by providing this location with a single outgoing transition which is always enabled. □

Process mapping

The process mapping is related to the very first part of the BHPC specification of a model.

Definition 5.4.4 (Process mapping) The mapping creates a dummy location. The outgoing transition of the dummy location is used by the BHPC operators as input to expand the model.
Preconditions BHPC process definition: $processname \triangleq B$, no dangling transitions.
Postconditions BHPC process: B, one dangling transition. □

Action prefix mapping

An action prefix takes a discrete transition in the system. Discrete changes, in hybrid automata, are described by transitions from one location to another. The action will be a decoration on the transition. Since the pre condition of this mapping requires a dangling transition, a transition is already taken. This transition however needs to be decorated with the action under evaluation.

Mapping 5.4.1 (Action prefix mapping) The action prefix operator will take a discrete transition in the system by applying the following SOS rule.

$$a.B \xrightarrow{a} B$$

The dangling transition already present in the hybrid automaton will be decorated with the action a. The algorithm will continue to evaluate process B.
Preconditions The hybrid automaton under construction has a single dangling transition, the BHPC process under evaluation is in the form $a.B$.
Postconditions The hybrid automaton under construction has a single dangling transition, the next process for evaluation is B. □

Symbolic trajectory prefix mapping

The trajectory prefix operator will take a set of trajectory Φ until $Pred_{exit}$ is satisfied and the trajectory can be completed. As long as $Pred_{exit}$ and $Pred$ are both satisfied the trajectory can be ended non-deterministically. However when $Pred$ is violated the trajectory is forced to end. When the trajectory has ended it will behave as process B. Continuous behaviour in a hybrid automaton is described by a location. The algorithm will add a location to the hybrid automaton under construction, the continuous behaviour defined by the trajectory under evaluation will be assigned to this new location. This location will also receive the invariant $Pred$ which determines whether the system can choose to remain in the location or is forced to leave. The exit conditions $Pred_{exit}$ determine whether the trajectory can be ended or not. An outgoing transition will be added to the new location. This transition will be decorated with a guard equal to the exit conditions. When the trajectory is complete the process can continue through this transition.

Mapping 5.4.2 (Trajectory prefix mapping) Let transition t_{prev} be the dangling transition and let location l_{prev} be the location t_{prev} originates from. Let the BHPC input be in the form $[f \mid \Phi \downarrow Pred \Downarrow Pred_{exit}].B$.

In the two conditions when

- $Pred_{exit}$ is not satisfied
- $Pred_{exit}$ and $Pred$ are both satisfied but the system chooses not to end the trajectory

the system must continue the trajectories in Φ by applying the SOS rule

$$[\varphi].B \xrightarrow{\varphi} B \quad \varphi \in \Phi$$

When the other cases then those appointed by the conditions above the system will end the trajectory and continue as process B by taking an SOS rule which is determined by process B.

The algorithm will add a location $l \in L$ to the hybrid automaton and connect the dangling transition to this location. Location l will be assigned the continuous dynamics $x1, ..., x_n$ where $x_i \in \mathbb{R}, i = 1, n$ and where $x_i \equiv t \in \varphi_{\mathbb{W}}$. A single outgoing transition $e \in E$ will be added to location l. Transition e will be decorated with the guard $g \in E \rightarrow 2^X$, where $g \equiv exitconditons$ After completion of the trajectory the algorithm will continue to evaluate process B.

Preconditions The hybrid automaton under construction has a single dangling transition, the BHPC process under evaluation is in the form $[\varphi].B$.
Postconditions The hybrid automaton under construction has a single dangling transition, the next BHPC process for evaluation is B. □

Choice operator mapping

The choice operator corresponds with a choice of outgoing transitions in a certain location. Due to the syntax of the Basic BHPC language this operator is restricted to its binary form. This leads to the choice of two outgoing transitions. This behaviour is simulated by introducing a dummy location with two outgoing transitions.

Mapping 5.4.3 Let transition t_{prev} be the dangling transition and let location l_{prev} be the location t_{prev} originates from. Let the BHPC input be in the form $B_1 + B_2$. By applying the following SOS rule the system will make a choice between B_1 and B_2.

$$\frac{B_1 \xrightarrow{a} B_1'}{\begin{array}{c} B_1 + B_2 \xrightarrow{a} B_1' \\ B_2 + B_1 \xrightarrow{a} B_2' \end{array}}$$

Let transitions t_{prev} be the dangling transition and let location l_{prev} be the location t_{prev} originates from. An equivalent behaviour to that as described by the choice operator can be obtained by adding the following elements to the hybrid automaton under construction.

- A location $l \in L$ such that $l_{prev} \xrightarrow{t_{prev}} l$.
- A transition t_1 such that $l \xrightarrow{t_1} B_1$
- A transition t_2 such that $l \xrightarrow{t_2} B_1$

Preconditions The hybrid automaton under construction has a single dangling transition t_{prev}, the BHPC process under evaluation is in the form $B_1 + B_2$.
Postconditions The hybrid automaton under construction has two dangling transitions t_1 and t_2, the processes B_1 and B_2 will be processed in parallel where B_1 will connect to t_1 and B_2 will connect to t_2. □

Guard prefix mapping

The guard operator checks whether the system satisfies a predicate. When this predicate is satisfied the system will continue. When the predicate is not satisfied no SOS rules can be applied resulting in a deadlock state. For simulation purposes it is decided to model a pre-determined deadlock as a separate state. Therefore, in this case, the system will take a transition to a global deadlock state.

Mapping 5.4.4 (Guard prefix mapping) Let transition t_{prev} be the dangling transition and let location l_{prev} be the location t_{prev} originates from. Let the BHPC input be in the form $\langle Pred \rangle.B$. When the predicate $Pred$ is satisfied the following SOS rule can be applied, the system then will continue to evaluate

B. When the predicate is not satisfied no SOS rules can be applied resulting in a deadlock.

$$\frac{B \xrightarrow{a} B'}{\langle Pred \rangle.B \xrightarrow{a} B'} \models Pred$$

The following elements will be added to the hybrid automaton under construction.

- A location $l \in L$ such that $l_{prev} \xrightarrow{t_{prev}} l$.
- A transition t_1 such that $l \xrightarrow{t_1} B$ and guard $Pred$
- A transition t_2 such that $l \xrightarrow{t_2} l_{deadlock}$ and guard $\neg Pred$

Preconditions The hybrid automaton under construction has a single dangling transition t_{prev}, the BHPC process under evaluation is in the form $\langle Pred \rangle.B$
Postconditions The hybrid automaton under construction has a single dangling transition, the next BHPC process for evaluation is B. □

Deadlock

A deadlock state is defined as a state in which no further actions can be taken. Typically such a state is modeled in automata by a state with no outgoing transitions.

Mapping 5.4.5 (Deadlock mapping) The deadlock operator will place the process in a state in which no further actions are possible. It will do this by connecting the dangling transition of the hybrid automaton to a global deadlock location.
Preconditions The hybrid automaton under construction has a single dangling transition, the BHPC process under evaluation is in the form 0.
Postconditions The hybrid automaton under construction has no dangling transitions, there is no more process to evaluate. □

Recursion mapping

The recursion operator will let the system continue with the behaviour as defined for the process that is assigned to the recursion operator. This operator can be modeled in an automaton by creating a transition to the first state of the target process.

Mapping 5.4.6 (Recursion mapping) The recursion mapping will connect the dangling transition left by the previous operator to the first location of the target process.
Preconditions The process that has led to this operator has left a dangling transition in the automaton under construction.
Postconditions The dangling transition has been connected to the first location of the target process. There is no preceding process to translate. □

All the mappings described in this section are visualized in Figure 5.2.

Example 5.4.2 (Thermostat) When the mappings described in this chapter are applied to the thermostat example from example 3.5.1 the following hybrid automaton is constructed.

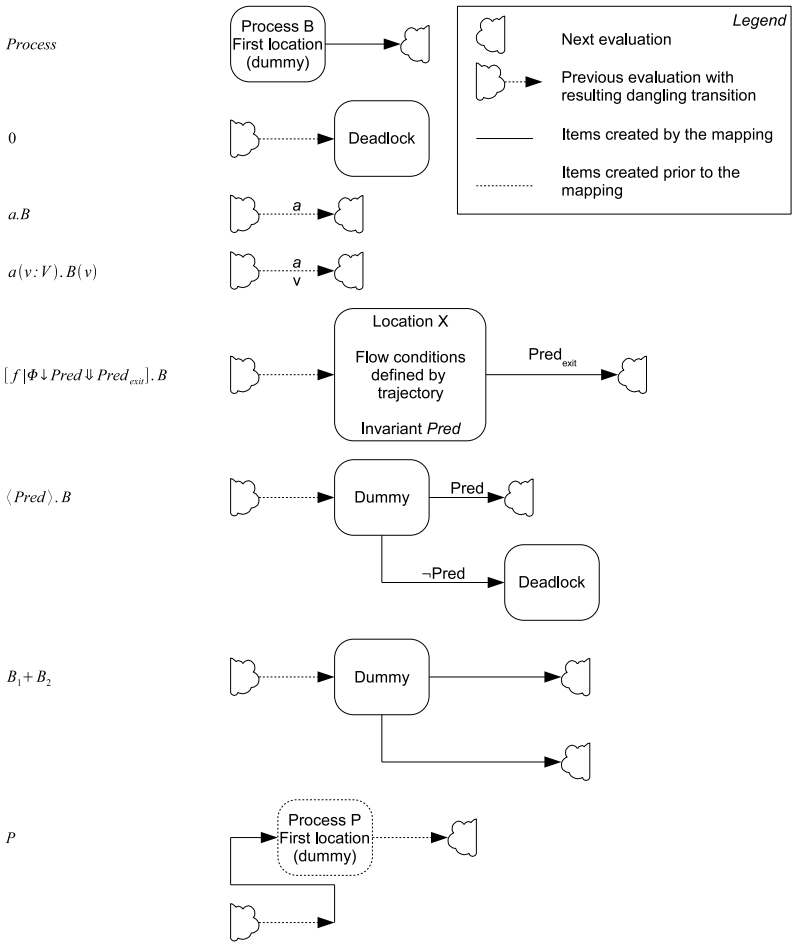
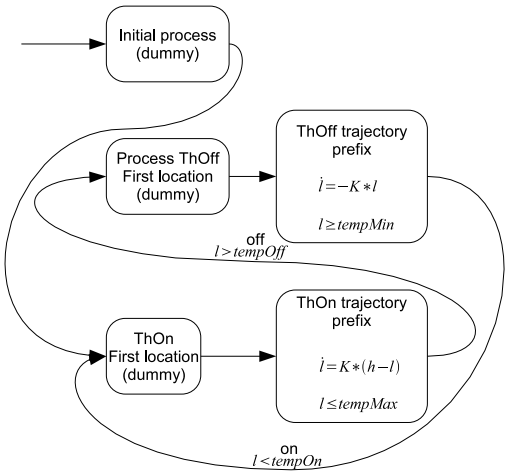


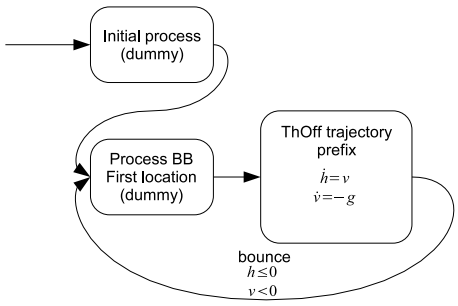
Figure 5.2: Translation mappings



The model clearly shows how the initial process is constructed into a dummy

node with a single outgoing transition. This transition leads to the first location of the process defined by the initial process. When processing the ThOff process a dummy start location is constructed with a single outgoing transition. This transition leads to the next location that is constructed by the mapping for the trajectory prefix. The location for the trajectory prefix shows the flow conditions and invariant derived from the trajectory and the exit conditions. Next the outgoing transition left by the trajectory prefix mapping receives the label `on` from the action prefix mapping. Finally before the ThOn process is translated the recursion mapping connect the transition left by the action prefix mapping to the first location of the process defined by the recursion operator. Next the ThOn process is processed in the same manor as the ThOff process.□

Example 5.4.3 (Bouncing ball) When the mappings described in this chapter are applied to the thermostat example from example 3.5.2 the following hybrid automaton is constructed.



The processing is done in analogously to the thermostat example and needs no further explanation. □

5.5 Translator tool

The above algorithm has been implemented into a translator application. This application can be used to construct and translate BHPC models. The application has two modules. The first module translates the model to its original mathematical notation. The output is in the LaTeX format. The second module can translate the model to Modelica code for simulation.

The translations are performed in several steps. First the model is parsed by the BHPC parser that translates the BHPC model to the XML based internal format described in section 3.3. This internal format is then passed to the actual translator module that will translate the model to the requested output. The Modelica translator applies the mappings, described in this chapter, to construct a hybrid automaton in Modelica code using components from the StateGraph library.

5.6 Simulation

This section will show simulation results for the thermostat and bouncing ball examples.

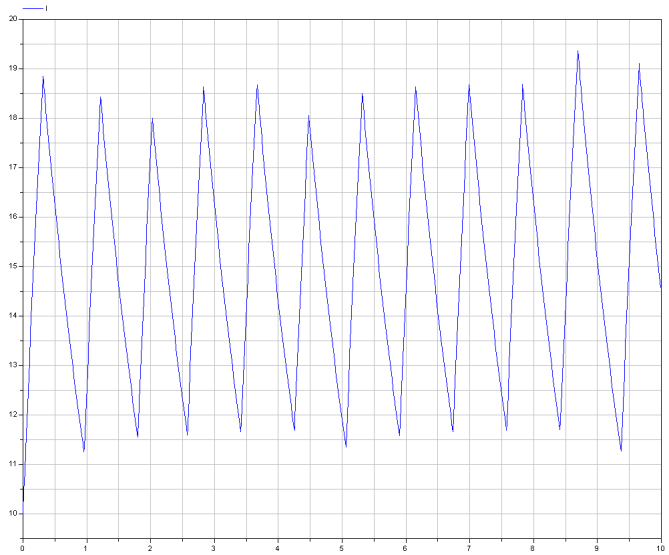
5.6.1 Thermostat

The original thermostat example from example 2.5.1 can be rewritten to the ASCII version of BHPC as shown in example 3.5.1. When this code is translated using the Translation tool constructed for this project the following Modelica code is generated.

```
model thermostat
  annotation (uses(Modelica(version="2.1")), Diagram);
  function random
    output Real r;
    external "C";
    annotation(Include="#include <stdlib.h>\n double random(void)
      {srand(time(NULL)); return (double)rand() / (double)RAND_MAX;}");
  end random;

  Real rnd(start = random());

  function getvaluebetween
    input Real min;
    input Real max;
    input Real rnd;
    output Real result;
  algorithm
    result := min + ((max - min)*rnd);
  end getvaluebetween;

  parameter Real K = 0.8;
  parameter Real h = 50;
  parameter Real tempMin = 10;
  parameter Real tempMax = 20;
  parameter Real tempOff = 18;
  parameter Real tempOn = 12;
  Real l;
  Real NonDeterministic__0;
  Real NonDeterministic__1;
  Integer on(start = 0);
  Integer off(start = 0);
  Real phiOff__l0;
  Real phiOn__l0;
  Real ThOff__l0;
  Modelica.StateGraph.Step ThOff__Step0(nIn=1, nOut=1);
  Modelica.StateGraph.Step ThOff__Step3(nIn=1, nOut=1);
  Modelica.StateGraph.Transition ThOff__Transition0(condition=true);
  Modelica.StateGraph.Transition ThOff__Transition3(
        condition=l > NonDeterministic__0 - 0.0001 and
        l < NonDeterministic__0 + 0.0001);
  Real ThOn__l0;
  Modelica.StateGraph.Step ThOn__Step1(nIn=2, nOut=1);
  Modelica.StateGraph.Step ThOn__Step4(nIn=1, nOut=1);
  Modelica.StateGraph.Transition ThOn__Transition1(condition=true);
  Modelica.StateGraph.Transition ThOn__Transition4(
        condition=l > NonDeterministic__1 - 0.0001 and
        l < NonDeterministic__1 + 0.0001);
  Modelica.StateGraph.InitialStep Initial__Step2(nIn=1, nOut=1);
  Modelica.StateGraph.Transition Initial__Transition2(condition=true);
algorithm
  when ThOff__Transition3.condition and ThOff__Step3.localActive then
    on :=on + 1;
  end when;
  when ThOn__Transition4.condition and ThOn__Step4.localActive then
    off :=off + 1;
  end when;
  when ThOff__Step0.localActive and ThOff__Transition0.condition then
    phiOff__l0 := ThOff__l0;
  end when;
  when ThOff__Step0.localActive and ThOff__Transition0.condition then
    reinit(l, phiOff__l0);
  end when;
  when ThOn__Step1.localActive and ThOn__Transition1.condition then
    phiOn__l0 := ThOn__l0;
  end when;
  when ThOn__Step1.localActive and ThOn__Transition1.condition then
```

```
    reinit(l, phiOn__l0);
  end when;
  when ThOn__Step4.localActive and ThOn__Transition4.condition then
    ThOff__l0 := l;
  end when;
  when Initial__Step2.localActive and Initial__Transition2.condition then
    ThOn__l0 := 10;
  end when;
  when ThOff__Step3.localActive and ThOff__Transition3.condition then
    ThOn__l0 := l;
  end when;
equation
  connect(ThOff__Step0.outPort[1], ThOff__Transition0.inPort);
  connect(ThOff__Step3.outPort[1], ThOff__Transition3.inPort);
  connect(ThOff__Step3.inPort[1], ThOff__Transition0.outPort);
  connect(ThOn__Step1.inPort[2], ThOff__Transition3.outPort);
  connect(ThOn__Step1.outPort[1], ThOn__Transition1.inPort);
  connect(ThOn__Step4.outPort[1], ThOn__Transition4.inPort);
  connect(ThOn__Step4.inPort[1], ThOn__Transition1.outPort);
  connect(ThOff__Step0.inPort[1], ThOn__Transition4.outPort);
  connect(Initial__Step2.outPort[1], Initial__Transition2.inPort);
  connect(ThOn__Step1.inPort[1], Initial__Transition2.outPort);

  if ThOff__Step3.localActive then
    der(l) = -K * l;
  elseif ThOn__Step4.localActive then
    der(l) = K * (h - l);
  else
    der(l) = 0;
  end if;

  rnd = (sin(time / (random()/100) + random()) + 1)/2;
  NonDeterministic__0 = getvaluebetween(tempMin, tempOn, rnd);
  NonDeterministic__1 = getvaluebetween(tempOff, tempMax, rnd);

end thermostat;
```

The following plot shows the simulations results from a run of this model. The plot shows that the model is able to switch the thermostat on and off in a non-deterministically.



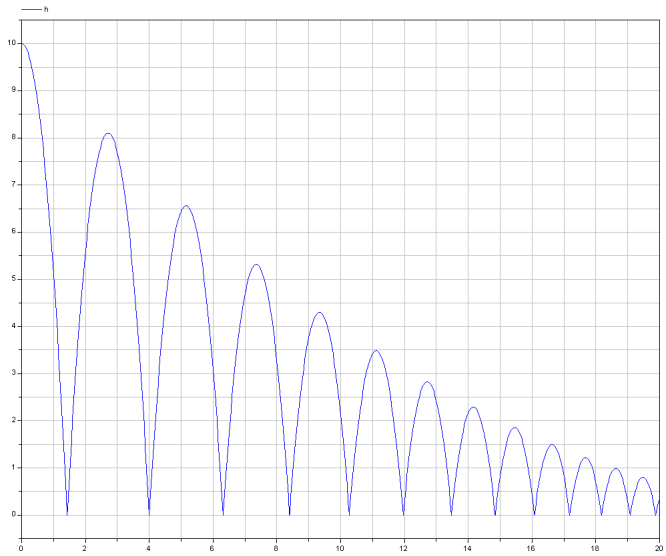
5.6.2 Bouncing ball

The original bouncing ball example from example 2.5.2 can be rewritten to the ASCII version of BHPC as shown in exampleřefex:ASCII BHPC Bouncing ball model. When this code is translated using the Translation tool constructed for this project the following Modelica code is generated.

```
model bouncingball
  annotation (uses(Modelica(version="2.1")), Diagram);
  function random
    "Do not use this function!! Use the rnd variable to get a random number"
    output Real r;
    external "C";
    annotation(Include="#include <stdlib.h>\n double random(void)
      {srand(time(NULL)); return (double)rand() / (double)RAND_MAX;}");
  end random;

  Real rnd(start = random());

  function getvaluebetween
    input Real min;
    input Real max;
    input Real rnd;
    output Real result;
  algorithm
    result := min + ((max - min)*rnd);
  end getvaluebetween;

  parameter Real c = 0.9;
  parameter Real g = 9.8;
  Real h;
  Real v;
  Integer bounce(start = 0);
  Real phi_h0;
  Real phi_v0;
  Real BB_h0;
  Real BB_v0;
  Modelica.StateGraph.Step BB_Step0(nIn=2, nOut=1);
  Modelica.StateGraph.Step BB_Step2(nIn=1, nOut=1);
  Modelica.StateGraph.Transition BB_Transition0(condition=true);
  Modelica.StateGraph.Transition BB_Transition2(
    condition=h <= 0 and v < 0);
  Modelica.StateGraph.InitialStep Initial_Step1(nIn=1, nOut=1);
  Modelica.StateGraph.Transition Initial_Transition1(condition=true);
algorithm
  when BB_Transition2.condition and BB_Step2.localActive then
    bounce :=bounce + 1;
  end when;
  when BB_Step0.localActive and BB_Transition0.condition then
    phi_h0 := BB_h0;
  end when;
  when BB_Step0.localActive and BB_Transition0.condition then
    phi_v0 := BB_v0;
  end when;
  when BB_Step0.localActive and BB_Transition0.condition then
    reinit(h, phi_h0);
    reinit(v, phi_v0);
  end when;
  when Initial_Step1.localActive and Initial_Transition1.condition then
    BB_h0 := 10;
  end when;
  when BB_Step2.localActive and BB_Transition2.condition then
    BB_h0 := 0;
  end when;
  when Initial_Step1.localActive and Initial_Transition1.condition then
    BB_v0 := 0;
  end when;
  when BB_Step2.localActive and BB_Transition2.condition then
    BB_v0 := -c * v;
  end when;
equation
  connect(BB_Step0.outPort[1], BB_Transition0.inPort);
  connect(BB_Step2.outPort[1], BB_Transition2.inPort);
```

```
  connect(BB__Step2.inPort[1], BB__Transition0.outPort);
  connect(BB__Step0.inPort[2], BB__Transition2.outPort);
  connect(Initial__Step1.outPort[1], Initial__Transition1.inPort);
  connect(BB__Step0.inPort[1], Initial__Transition1.outPort);

  if BB__Step2.localActive then
    der(h) = v;
  else
    der(h) = 0;
  end if;

  if BB__Step2.localActive then
    der(v) = -g;
  else
    der(v) = 0;
  end if;

  rnd = (sin(time / (random()/100) + random()) + 1)/2;

end bouncingball;
```

The following plot shows the simulations results from a run of this model. The plot shows the height of the ball. It is dropped from a predefined height and looses energy on every bounce.



5.7 Conclusions

For the initial version we have decided to define a basic calculus that would be feasible with respect to the available time and restrictions from Modelica and the StateGraph library. This basic calculus is named Basic BHPC. A translation algorithm is defined that can translate Basic BHPC models to hybrid automata. The translation algorithm is based on a number of mappings that relate each BHPC operator to a fragment of a hybrid automaton. When put together the total hybrid automaton will show the same behaviour as the BHPC process that was translated. This hybrid automaton can then be simulated using Modelicas StateGraph library with some additional Modelica code as shown in chapter 4.4.

CHAPTER 6

Conclusions

With this thesis we have taken a step forward in the simulation of hybrid systems using Behavioural Hybrid Process Calculus (BHPC). In addition to the discrete simulator developed by M.H. Schonenberg we now have the ability to simulate both discrete as continuous transitions within the BHPC model. BHPC is an extension of classical process algebra, based on behavioural theory that is suitable for modelling and analysis of hybrid systems.

6.1 Parser and internal representation

With the development of the BHPC parser we have created an easy way for future research project to work with the BHPC language. Applications that make use of this parser can be developed more efficiently since the XML based internal representation is much easier to parse than the language itself.

BHPC models can be easily designed with the use of the ASCII based version of BHPC. This language does not contain any mathematical symbols that are hard to input using a common keyboard. Using the prototype of the parser and translator the ASCII version of the language can easily be translated to its original notation using LaTeX output.

6.2 Translation to Modelica

A subset of the BHPC language is taken for the translation to Modelica. The complete language would prove to be to difficult due to the parallel operator. The parallel operator of the StateGraph shows different behaviour than needed. In addition to this, Modelica does not support dynamic allocation of memory making implementation of the parallel operator in combination with the recursion operator impossible.

For the development of the translator a set of mappings is defined that translate a single BHPC operator into a fragment of a hybrid automaton. The

combination of the individual fragments form a hybrid automaton that exhibits the same behaviour as the original BHPC model.

This hybrid model is translated to the Modelica language such that it can be executed during a simulation run. The simulations for the thermostat and bouncing ball examples show good results. The plots obtained from the Dymola simulator show the same results as described in [Kri06].

6.3 Future work

In this document we have shown how a subset of the BHPC language can be translated into a hybrid automaton and simulated using the Modelica language. Some operators were not implemented because their implementation was infeasible in a technical sense or by restriction of time.

When the superposition operator becomes available in the BHPC language a translation scheme can be developed for it. This operator might make it possible to implement the parallel operator as well. There are ideas on how a parallel composition can be expressed as a superposition of processes using an expansion law.

Other improvements involve modifying the choice and superposition in the parser program such that they can work on sets in stead of the fixed binary form that they are in now.

Since the BHPC parser uses a XML based intermediate format other applications can easily use this output. Therefore other projects may want to use the parser to work on the BHPC language. The parser should be up-to-date and made available to other groups that want to use it.

Bibliography

[AHS96] R. Alur, T.A. Henzinger, and E.D. Sontag, editors. *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III: verification and control*, volume 1066, 1996.

[ant] Antlr parser generator. http://www.antlr.org.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *Computer Networks*, 14:25–59, 1987.

[BCL+05] C. Brooks, A. Cataldo, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng. Hyvisual: A hybrid system visual modeler. Technical report, University of California, 2005.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[BK05] E. Brinksma and T. Krilavičius. Behavioural hybrid process calculus. Technical Report TR-CTIT-05.45, CTIT, UT, 2005.

[BKU05] E. Brinksma, T. Krilavičius, and Y.S. Usenko. Process algebraic approach to hybrid systems. In *Proc. of 16th IFAC World Congress*, Prague, Czech Republic, juli 2005.

[Hen96] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292, 1996.

[HL02] Yerang Hur and Insup Lee. Distributed simulation of multi-agent hybrid systems. In *IEEE International Symposium on Object-Oriented Real-time distributed Computing (ISORC)*, 2002.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[iso] Iso/iec 14977, international organization for standardization, 1996. http://www.cl.cam.ac.uk/ mgk25/iso-14977.pdf.

[Kri06] T. Krilavičius. Behavioural hybrid process calculus. Draft, 2006.

[KS05] T. Krilavičius and H. Schonenberg. Discrete simulation of behavioural hybrid process calculus. In P.M.E. Bra and J.J. van Wijk, editors, *IFM2005 Doctoral Symposium on Integrated Formal Methods*, pages 33–38, Eindhoven, Netherlands, November 2005. DMCs, TUe.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989. ISBN 0-13-115007-3.

[moda] Modelica a unified object-oriented language for physical systems modeling. http://www.modelica.org/documents/ModelicaTutorial14.pdf.

[modb] Modelica association. http://www.modelica.org.

[MS06] Ka Lok Man and Ramon Robert Hubert Schiffelers. *Formal Specication and Analysis of Hybrid Systems*. PhD thesis, Technische Universiteit Eindhoven, 2006.

[PW98] J.W. Polderman and J. C. Willems. *Introduction to Mathematical Systems Theory: a behavioral approach*. SPRINGER, 1998.

[Sch05] R.A. Schouten. Simulation of hybrid processes. Master's thesis, Technische Universiteit Eindhoven, 2005.

[Sch06] M.H. Schonenberg. Discrete simulation of behavioural hybrid process algebra. Technical report, University of Twente, 2006. Masters thesis.

[uni] The unicode standard, version 4.0, unicode consortium, 2003. http://www.unicode.org/versions/Unicode4.0.0/ch14.pdf.

[W3X04] Extensible markup language (xml) 1.0 (third edition), 2004.

[Wika] Ascii - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Ascii.

[Wikb] Differential equation - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Differential_equations.

APPENDIX A

ASCII BHPC definition

This appendix shows the complete EBNF definition for the ASCII variant of the BHPC language.

A.1 Global elements

```
IDENTIFIER:
  ('a'..'z' | 'A'..'Z')
  ('a'..'z' | 'A'..'Z' | '0'..'9')*

NUMBER:
  ('0'..'9')+
  ('.' ('0'..'9')+)?

qualifier             : IDENTIFIER
action                : IDENTIFIER
constant              : IDENTIFIER
processname           : IDENTIFIER
trajectoryname        : IDENTIFIER

parameterdefs:
  IDENTIFIER (',' IDENTIFIER)*

parameters:
expression (',' expression)*

qualifiers:
  qualifier (',' qualifier)*

actions:
  action (',' action)*
```

A.2 Expressions

```
expression:
equalityexpr ("=" equalityexpr)*

equalityexpr:
  additiveexpr (
   ("="
```

```
   | "!="
   | '>'
   | ">="
   | '<'
   | "<="
   ) additiveexpr)*

additiveexpr:
multiplicexpr (
('+'
| '-'
) multipricexpr)*

multipricexpr:
unaryexpr (
('*'
| '/'
) unaryexpr)*

unaryexpr:
  ('-')? (qualifier | constant | NUMBER)
| '(' expression ')'

expressions:
  expression (',' expression)*
```

A.3 Model

```
bhpcmodel:
  (actiondefinitions | qualifierdefinitions   | constantdefinitions)*
  initialprocess
  (processdefinition | trajectorydefinition)+
```

A.4 Action, qualifier and constant definitions

```
actiondefinitions     : "actions" ':' action (',' action)*
qualifierdefinitions  : "qualifiers" ':' qualifier (',' qualifier)*
constantdefinitions   : "constants" ':' constantdefinition
                        (',' constantdefinition)*
constantdefintion     : IDENTIFIER ":=" expression
```

A.5 Process definitions

```
initialprocess: "initial" processname (parameters)?
processdefinition: process (parameters)? "^=" processops

processops:
  deadlock         | actionprefix     | trajectoryprefix | choice           |
  superposition    | parallel         | hiding           | renaming         |
  recursion

deadlock              : "stop"
actionprefix          : action '.' processops
trajectoryprefix      : '[' qualifiers
                        '|' ( trajectory (expressions)?
                            | trajectorydefinition )
                        (syncprefixes)? (exitconditions)?
                        ']' '.' processops
choice                : processops '+' processops
superposition         : processops "++" processops
parallel              : processops '|' (synchronizers)? '|' processops
hiding                : "new" ( actions (',' qualifiers)? | qualifiers )
                        "in" processops
renaming              : processops '[' renamingfunc ']'
```

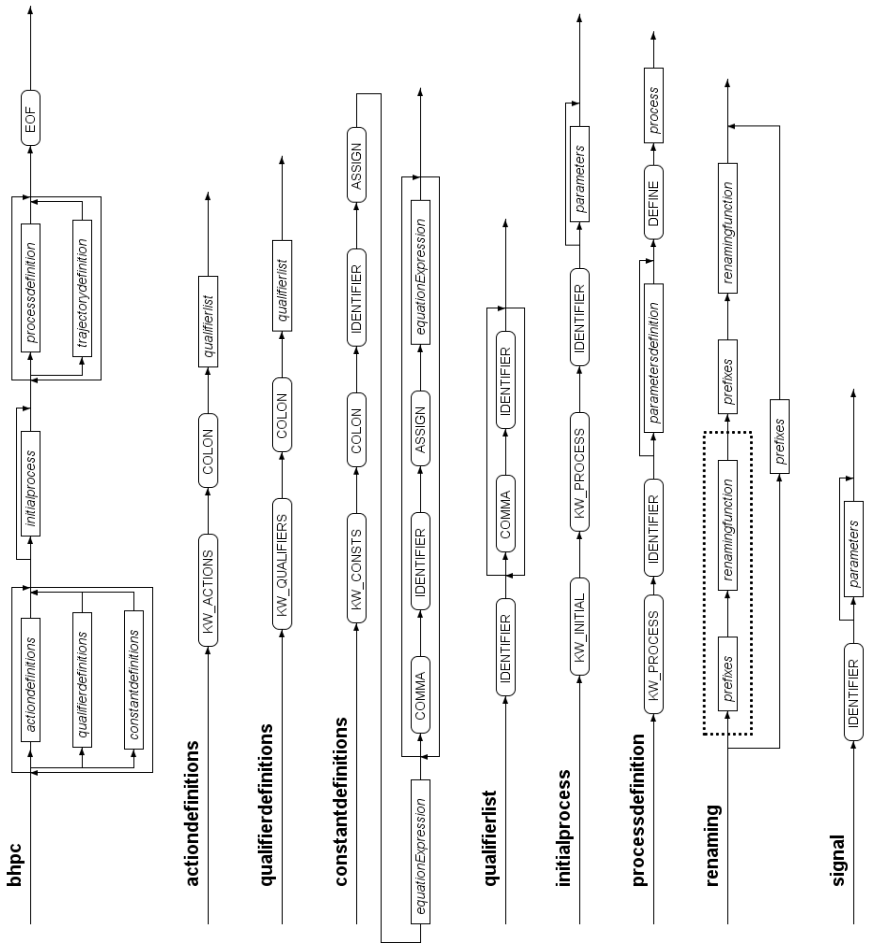
```
recursion             : process (expressions)?

syncprefixes          : "conds" expressions
exitconditions        : "exits" expressions
synchronizers         : (qualifiers (',' actions)?) | actions
renamingfunc          : (action | qualifier) ('\' IDENTIFIER)*
```

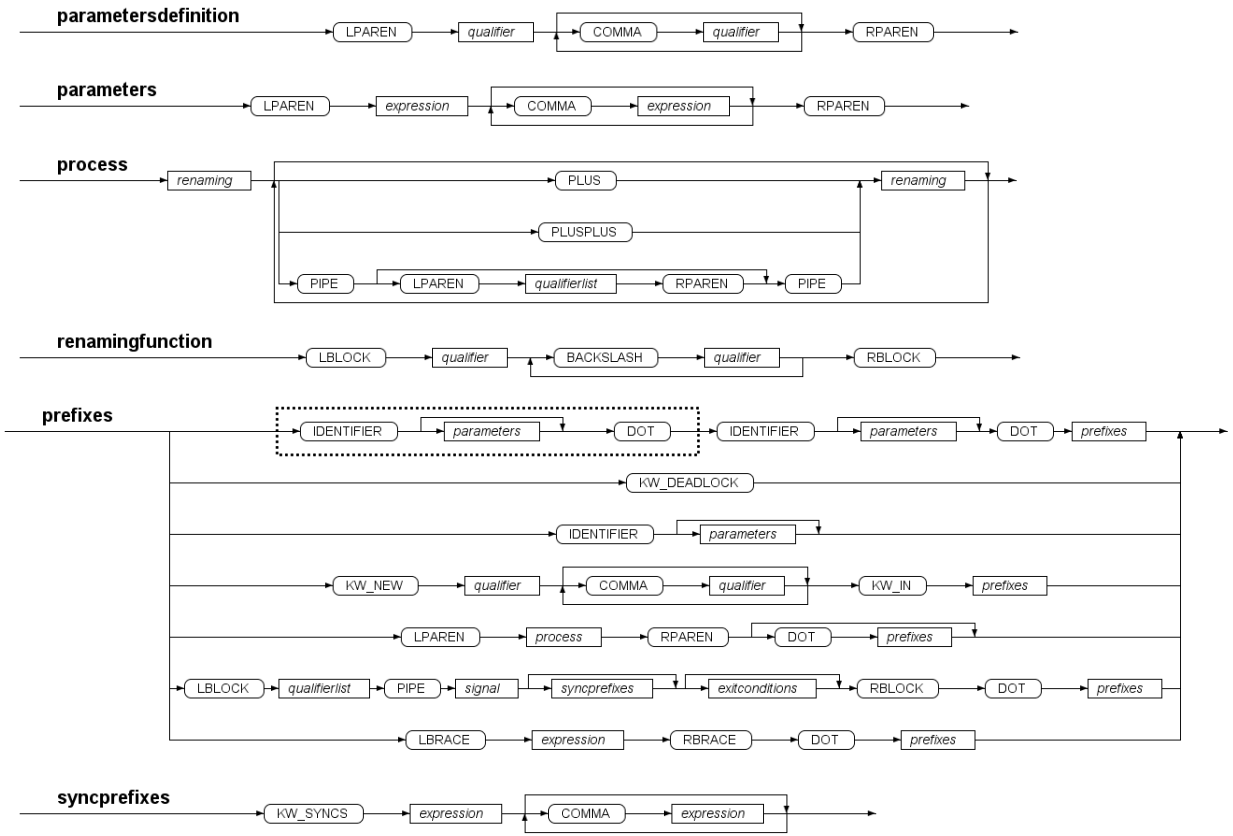
A.6 Trajectory definitions

```
trajectorydefinition:
  "signal" trajectoryname (parameters)? "^="
  qualifiers ":" timeinterval
  ( (assignments (',' differentialequations)? (',' expressions)?)
  | (differentialequations (',' expressions)?)
  | expressions
  )

timeinterval          : '(' NUMBER ',' (expression | 't') ']' "=>" domain
domain                : "\N" | "\Z" | "\R"
assignments           : assignment (',' assignment)
assignment            : qualifier ('(' expression ')')? ":=" expression
differentialequations : "der" '(' qualifier ')' '=' expression
```

APPENDIX B

BHPC ASCII variant railroad diagram




parametersdefinition
LPAREN
qualifier
COMMA
qualifier
RPAREN
parameters
LPAREN
expression
COMMA
expression
RPAREN
process
renaming
PLUS
renaming
PLUSPLUS
PIPE
LPAREN
qualifierlist
RPAREN
PIPE
renamingfunction
LBLOCK
qualifier
BACKSLASH
qualifier
RBLOCK
prefixes
IDENTIFIER
parameters
DOT
IDENTIFIER
parameters
DOT
prefixes
KW_DEADLOCK
IDENTIFIER
parameters
KW_NEW
qualifier
COMMA
qualifier
KW_IN
prefixes
LPAREN
process
RPAREN
DOT
prefixes
LBLOCK
qualifierlist
PIPE
signal
syncprefixes
exitconditions
RBLOCK
DOT
prefixes
LBRACE
expression
RBRACE
DOT
prefixes
syncprefixes
KW_SYNCS
expression
COMMA
expression


exitconditions
KW_EXITS
expression
COMMA
expression
qualifier
IDENTIFIER
parameter
qualifier
trajectorydefinition
KW_SIGNAL
IDENTIFIER
parameters
DEFINE
LBRACE
qualifierlist
COLON
timedomain
PIPE
trajectoryExpressions
COMMA
trajectoryExpressions
RBRACE
trajectoryExpressions
IDENTIFIER
LPAREN
NUMBER
RPAREN
ASSIGN
assignment
expression
differentialequation
timedomain
LPAREN
NUMBER
COMMA
expression
RBLOCK
RARROW
IDENTIFIER
differentialequation
KW_DER
LPAREN
qualifier
COMMA
NUMBER
RPAREN
EQUAL
expression
assignment
qualifier
LPAREN
NUMBER
RPAREN
ASSIGN
equationExpression
expression
equationExpression

equationExpression
additiveExpression
EQUAL
NOTEQUAL
GREATER
GREATEREQ
SMALLER
SMALLEREQ
additiveExpression
additiveExpression
multiplicativeExpression
PLUS
MINUS
multiplicativeExpression
multiplicativeExpression
unaryExpression
MULTIPLY
DIVIDE
unaryExpression
unaryExpression
MINUS
NUMBER
IDENTIFIER
LPAREN
expression
RPAREN

APPENDIX C

Internal representation XML schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.utwente.nl"
           elementFormDefault="qualified">

<xs:element name="bhpc">
  <xs:complexType>
    <xs:element name="action" type="xs:string">
      <xs:complexType>
        <xs:element ref="name" minOccurs="0" maxOccurs="unbounded"/>
      </xs:complexType>
    </xs:element> <!-- action tag -->

    <xs:element name="qualifier" type="xs:string">
      <xs:complexType>
        <xs:element ref="name" minOccurs="0" maxOccurs="unbounded"/>
      </xs:complexType>
    </xs:element> <!-- qualifier tag -->

    <xs:element name="constant" type="xs:string">
      <xs:complexType>
        <xs:element ref="name" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="value" minOccurs="0" maxOccurs="unbounded"/>
      </xs:complexType>
    </xs:element> <!-- constant tag -->

    <xs:element name="processdefinition" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:element ref="name" minOccurs="1" maxOccurs="1"/>
        <xs:element name="process" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:element ref="deadlock        " minOccurs="0" maxOccurs="1"/>
                <!-- deadlock can only occur once -->
            <xs:element ref="recursion       " minOccurs="0" maxOccurs="1"/>
                <!-- recursion can only occur once -->
            <xs:element ref="actionprefix    " minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="trajectoryprefix" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="renaming        " minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="alternative     " minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="superposition   " minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="parallel        " minOccurs="0" maxOccurs="unbounded"/>
          </xs:complexType>
        </xs:element> <!-- process tag -->
      </xs:complexType>
    </xs:element> <!-- processdefinition -->
```

```
    <xs:element name="trajectory" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:element ref="name"/>
        <xs:element ref="variables"/>
        <xs:element name="timeperiod" minOccurs="1" maxOccurs="1"/>
        <xs:element name="demain" minOccurs="1" maxOccurs="1"/>
        <xs:element name="definition" minOccurs="1" maxOccurs="1"/>
      </xs:complexType>
    </xs:element> <!-- signal tag -->

  </xs:complexType> <!-- bhpc tag -->
</xs:element>

<!-- base elements -->
<xs:element name="name" type="xs:string"/>

<xs:element name="value" type="xs:string"/>

<xs:element name="variables" minOccurs="1" maxOccurs="1">
  <xs:complexType>
    <xs:element name="variable" minOccurs="0" maxOccurs="unbounded" type="xs:string"/>
  </xs:complexType>
</xs:element>

<xs:element name="deadlock"></xs:element>
<!-- need to create all other operators here -->

</xs:schema>
```