"Discrete Simulation of
Behavioural Hybrid Process Calculus"

M.H. Schonenberg

Bhave!

"Discrete Simulation of Behavioural Hybrid Process Calculus"

Master's Thesis
Faculty of Electrical Engineering,
Mathematics & Computer Science
University of Twente

M.H. Schonenberg
m.h.schonenberg@student.utwente.nl

Graduation Committee

Dr. T. Krilavičius (first supervisor)
Dr. Ir. R. Langerak
Dr. J.W. Polderman

20th September 2006

Abstract

Nowadays complex systems, such as electrical and mechanical, are often controlled by software. Moreover, hybrid models are used to model and analyse complex biological and chemical systems. These systems are called hybrid systems because they display both discrete and continuous behaviour. For the development and analysis of these systems techniques are needed that capture both behaviours and their relation. Simulation is one of the tools to obtain insight in dynamical systems behaviour. Simulation results provide information on system performance and they are useful for detecting potential weaknesses and errors.

In [8, 9, 11] Behavioural Hybrid Process Calculus (BHPC) is proposed for modelling hybrid systems. BHPC is a process algebraic calculus that extends the standard repertoire of operators that combine discrete functional behaviour with features to also represent and compose continuous-time behaviour. At the higher abstraction level these two types of behaviour are treated uniformly, i.e., as normal elements of process algebra. Following [20], behaviour can be simply seen as the set of all allowed real-time evolutions, or trajectories, of the system variables. One of the most important properties of BHPC is that the strong bisimulation relation for BHPC is a congruence. Such a property allows to interchange bisimilar processes in any process algebraic expression. In other words, it allows to refine processes, change their internal representation, and interchange them without any losses as long as they manifest the same behaviour.

We present abstraction techniques that can be used to simulate hybrid system control structures. These techniques are applicable for all hybrid calculi where continuous-time behaviour is treated like an ordinary element of the algebra. We also present a general and modular discrete engine definition that is based on BHPC semantics. Due to its modularity, this definition allows hybrid extension and further extensions of BHPC. Applicability of the discrete simulation theory and definitions is tested by the implementation of these techniques in a prototype. The prototype is able to simulate control structures of hybrid systems, but moreover it is applicable as tool for testing developments of BHPC since the modular implementation supports extensions of BHPC. Finally, we propose a visualisation technique for hybrid simulation that allows analysis of hybrid behaviour.

The applicability as “sand-box” tool for BHPC can help with further developments of the BHPC framework. We believe that due to the modularity of our engine definitions, the engine could also be suitable for other other frameworks. Furthermore we believe that our visualisation technique for analysing hybrid behaviour has great potential and can help in gaining insight in the relation between discrete and continuous-time behaviour.

Acknowledgements

This thesis is the result of many months of work and doubt. During this period I had wonderful people around me, helping me on the topic, being good company, or both.

First of all I would like to thank Tomas Krilavičius for giving valuable feedback and for helping me get back to work during periods of doubt. For the finishing touches on the thesis and presentation, I would like to thank Bas van Schoonhoven and Laurens Satink. I would like to thank Albert Brand for converting my sketch of a bee into a logo for the Bhave tool set. Also, I would like to give my sincere thanks to my family, that supported me, even without fully understanding the contents of my study. Finally, thanks to Ramon de Jonge for all your good company, patience and belief in me.

Enschede, 20th September 2006
Helen Schonenberg

Contents

List of Tables

1 Introduction

Behaviour of systems with continuously changing behaviour, also called continuous-time systems, can be expressed by variables that evolve over time. Opposite to these are discrete event systems in which behaviour is influenced by asynchronous discrete events, rather than by time. Hybrid systems are systems that display both discrete and continuous-time behaviour. Hybrid behaviour can be found in physical systems that display sudden changes in continuous evolution, for instance caused by interaction with digital devices. For example, electrical, mechanical and biological systems can manifest hybrid behaviour. Hybrid systems have become more common now that physical systems are more often controlled digitally, for example by the increased usage of embedded software.

The increasing interest in hybrid systems has led to a growing interest in models and formalisms to specify and analyse these systems. Simulation is one of the tools for analysing dynamical systems behaviour. It can help in detecting potential weaknesses and errors in hybrid system models and in choosing adequate control strategies and parameters. In this thesis we describe the development of techniques and a prototype for discrete simulation of Behavioural Hybrid Process Calculus (BHPC).

BHPC is an extension of classical process algebra. The extension is based on the behavioural theory [25], which is suitable for modelling and analysis of hybrid dynamical systems [8, 9, 11]. In the remainder of this introduction we give a more detailed description of our goals and approach. First we provide some background information about modelling and analysis of hybrid systems in Section 1.1 and 1.2. Then we describe our goals in Section 1.3 and present our methodology in Section

1.4. Section 1.5 summarises our contribution to the field of analysing hybrid systems. Finally the outline for this thesis is given in Section 1.6.

1.1 Modelling hybrid systems

A prominent framework for hybrid systems is provided by the family of hybrid automata models (hybrid automata [2], hybrid behavioural automata [19], hybrid input/output automata [22]). For an overview of these modelling frameworks we refer to [11]. More recently process algebraic models have been put forward as a vehicle for the study of hybrid systems [6, 8, 13, 29].

Process algebra A process algebra [24, 16, 5, 7] is a theoretical framework for the modelling and analysis of the behaviour of concurrent discrete event systems that has been developed within computer science in the past quarter century. It has generated a deeper understanding of the nature of concepts such as observable behaviour in the presence of non-determinism, system composition by interconnection of concurrent system components, and notions of behavioural equivalence of such systems. It has contributed fundamental concepts such as bisimulation, and has been successfully used in a wide range of problems and practical applications in concurrent systems.

Behavioural Hybrid Process Calculus In [8, 9, 11] Behavioural Hybrid Process Calculus (BHPC) is proposed for modelling hybrid systems. BHPC is a process algebraic calculus that extends the standard repertoire of operators that combine discrete functional behaviour with features to also represent and compose continuous-time behaviour. At a higher abstraction level these two types of behaviour are treated uniformly, i.e., as normal elements of process algebra. Following [25], behaviour can be simply seen as the set of all allowed real-time evolutions, or *trajectories*, of the system variables.

In contrast to [13] and [6], in BHPC strong bisimulation is a congruence relation with respect to the parallel composition of subsystems[1], i.e., substituting a subsystem with a bisimilar subsystem does not affect the behaviour of the composition. We believe that BHPC has great potential for modelling and analysis of hybrid systems. In this work we contribute to the analysis of hybrid systems by developing techniques and a prototype for discrete simulation of BHPC.

1.2 Hybrid system analysis

Most real life systems are too complex to be conveniently analysed by their algebraic description only. Simulation can be used for computer guided analysis of the model.

[1]In [13], the robust and stateless bisimulations are however congruent.

Simulation tools are used to evaluate model behaviour by trying different model scenarios. For hybrid simulation there are several tools available, based on different modelling frameworks. In [10] an exhaustive overview for languages for modelling, simulation and analysis of hybrid systems is given. For a description of the most popular tools we refer to [11].

Bhave tool set We believe in the potential of a tool set for modelling and analysis of hybrid systems, based on the characteristics of BHPC. Bhave[2], acronym for *"BHPC's hybrid and visual engine"*, is a collection of tools, developed for this purpose. Currently the tool set is still under development. The Bhave prototype [11] is a prototype for limited hybrid simulation of BHPC. Despite its limits, it has been used successfully for simulation of systems with complex behaviour, e.g. Zeno phenomena.

Visualisation of simulation results Whereas simulation results for continuous-time behaviour are given by plots that visualise the evolution of system variables along a time line, simulation results for discrete behaviour are given by message sequence charts (MSC) [27, 17], that depict interaction patterns between processes. Hybrid systems manifest both types of behaviour that can influence one another. More precisely, the evolution of system variables may trigger events and events may trigger change of evolution. Hence a combined view is crucial to fully analyse hybrid system behaviour. Several approaches are proposed in [28, 21, 15], but none of them provides such a combined view.

1.3 Goals

We believe that the BHPC framework can contribute to the modelling and analysis of hybrid systems. Hybrid system analysis without tool support can be a cumbersome task. Hence, tool support is essential for the employment of BHPC in its field. Simulation is a common technique for tool guided analysis of hybrid systems.

We take a first step towards full hybrid simulation of BHPC by developing techniques and a prototype for discrete simulation of BHPC. Discrete simulation is the simulation of discrete behaviour (actions) and can be used to analyse hybrid system control structures. The prototype for discrete simulation has to be extendible to hybrid simulation to allow support for continuous-time behaviour in further developments. Moreover the prototype should be developed as a sand-box type of tool to facilitate further developments of BHPC.

[2]`http://fmt.cs.utwente.nl/tools/bhave/`

1.4 Methodology

To accomplish our goal, we investigate abstraction of continuous-time behaviour, the definition and implementation of an engine based on BHPC semantics, extendibility of the engine and visualisation of hybrid simulation results.

Abstraction techniques To perform discrete simulation of BHPC, continuous-time behaviour must be removed from the model. In BHPC continuous-time behaviour is represented by trajectories, which are treated like normal process algebraic elements. Therefore discrete simulation can be performed by abstracting from trajectories. This can be done by treating trajectories as special actions (representatives), regarding the operational semantics that apply for trajectories. For example, there should not be synchronisation between actions and representatives for trajectories in discrete simulation.

Engine based on BHPC semantics The engine performs correct simulation, if and only if for all BHPC specifications, the engine produces the same traces as the operational semantics of BHPC. Therefore there must be a clear relation between the engine procedures that are responsible for determining simulation steps and the operational semantics of BHPC. Following [32] we define procedures that determine simulation steps by decomposing BHPC expressions using the operational semantics of BHPC.

Extendibility of the engine The engine needs to be extendible for both hybrid simulation and changes to BHPC. Both extensions can affect the procedures that determine the set of the simulation steps, which are based on the semantics of BHPC. For hybrid extension, the procedure for executing simulation steps must also be able to cope with execution of continuous-time behaviour and related changes to the engine state. Therefore we present a modular engine definition in which procedures for determination and execution of simulation steps are hidden from the engine definition.

Visualisation of hybrid simulation results Hybrid systems display both discrete and continuous-time behaviour, which may influence one another. For full analysis of hybrid behaviour it is necessary to present both behaviours and their interaction. This can be done by combining existing visualisation techniques for both behaviours. Plots and message sequence charts (MSC) can be combined in one view by drawing plots above MSCs that are rotated 90 degrees. The combination of discrete and continuous-time behaviour shows their interaction, e.g. when an event changes the evolution of a system variable, then the plot for that variable will change after occurrence of that event.

1.5 Contribution

In this thesis we have made several contributions to the field of hybrid system analysis.

Abstraction techniques for discrete simulation We have presented abstraction techniques to abstract from continuous-time behaviour in BHPC. These techniques can be used to gain insight in control structures of hybrid systems without parallelism. They are also applicable for other hybrid algebraic formalisms where continuous-time behaviour is treated as a normal algebraic element.

General engine definition We have provided a modular engine definition for the discrete simulation of BHPC that is based on its semantics. Due to the modularity, the engine allows hybrid extension and changes to BHPC itself. Another result of the modularity is that BHPC related procedures can easily be replaced by procedures for other modelling frameworks, making this a general engine definition, suitable for other hybrid algebraic frameworks.

Tool for experimenting with BHPC We have implemented a prototype that is based on the discrete engine definition. The tool can be used to simulate control structures of hybrid models, specified in BHPC. Moreover, the tool is suitable as experimental environment for BHPC, supporting further BHPC developments.

Visualisation techniques Finally, we have presented MSP, a visualisation technique for the analysis of hybrid systems, that visualises hybrid behaviour. Other visualisation techniques do not represent a combined view of discrete and continuous-time behaviour and their relation.

1.6 Outline

We start with a description of BHPC in Chapter 2. Then we present the discrete engine in Chapter 3. The design and implementation of the discrete engine are given in Chapter 4. Simulation results are presented in Chapter 5. In Chapter 6 definitions for a hybrid engine are presented. Moreover in this chapter we propose a new format to present simulation results for hybrid simulation that captures both the discrete and continuous behaviour. Finally conclusions and future work are presented in Chapter 7.

2 Behavioural Hybrid Process Calculus

Behavioural Hybrid Process Calculus (BHPC) [8, 9, 11] is an extension of classical process algebra that is suitable for the modelling and analysis of continuous and hybrid dynamical systems and can be seen as a generalisation of the behavioural approach [25] in a hybrid setting. This chapter contains a selection of BHPC definitions and theory from [11] that are relevant for our work. For a full description of BHPC and proofs we refer to [11].

Behaviour in hybrid systems can be seen as the set of all allowed real-time evolutions, or trajectories, of the system variables. Section 2.1 introduces trajectories. [8, 9, 11] use a labelled transition system to describe dynamic system behaviour. Their labelled transition system, or hybrid transition system, consists of two types of transitions, namely discrete transitions and continuous transitions. Section 2.2 describes hybrid transition systems. The BHPC language and operational semantics are described in Section 2.3. Section 2.4 defines the hybrid expansion law that is used to solve parallel composition. Section 2.5 gives additional BHPC operators to increase usability of the language. Application of BHPC is exemplified in Section 2.6.

2.1 Trajectories

In [11] it is assumed that trajectories are defined over time intervals $(0, t]$ (where t can be ∞) 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, like temperature, pressure, etc. These components are associated with *trajectory qualifiers* that identify them.

Definition 2.1.1 (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

$$\mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n))$$

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

Example 2.1.2 (Signal space). A bouncing ball is a simple example of a hybrid system (Example 2.6.1). It is a simplified model of an elastic ball that is bouncing and losing a fraction of its energy with every bounce. The altitude of the ball is h, v is the vertical speed, and c is a coefficient for the energy loss. The following signal space can be used to define dynamical behaviour of ball:

$$\mathbb{W}_{\mathrm{BB}} = (\mathbb{R}_+ \times \mathbb{R}, (h, v))$$

where qualifiers h and v refer to the altitude of ball (in $\mathbb{R}_+$) and the vertical speed (in $\mathbb{R}$), respectively. □

Definition 2.1.3 (Trajectory). Let $\mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n))$ be a signal space. Then a *trajectory* in signal space $\mathbb{W}$ is a function

$$\varphi_{\mathbb{W}} : (0, t] \to W_1 \times \cdots \times W_n \tag{2.1}$$

where $t \in \mathbb{R}_+$ is the duration of the trajectory, also denoted as $\mathrm{t}(\varphi)$. Subscript $\mathbb{W}$ is omitted when the signal space is clear from the context. □

Often trajectories are defined over infinite time intervals. However, hybrid systems usually evolve according to some trajectory only for a certain period of time. The restriction to interval $(0, t]$ allows to define such evolutions. There are two reasons for the choice of such type of intervals. It is convenient technically. Moreover, it most accurately reflects the reality. An evolution starts at a certain time with a certain state, and the first change occurs in the left limit. Consequently, the evolution stops at a certain time moment with a certain state.

Furthermore, infinite trajectories are allowed, but with certain limitations. They have also defined an *empty trajectory*.

Definition 2.1.4 (Empty trajectory). ϵ denotes an *empty* (or a *completed* trajectory). □

Definition 2.1.5 (Set of trajectory qualifiers). A function $\mathrm{T} : \Phi \to \mathcal{T}$, where Φ is a set of any trajectories and $\mathcal{T}$ is a set of qualifiers, collects all trajectory qualifiers of the trajectory:

$$\begin{aligned}
&\mathrm{T}(\varphi_{\mathbb{W}}) = \\
&\quad \{q \mid \varphi_{\mathbb{W}} : (0,u] \to W_1 \times \cdots \times W_n \\
&\qquad \wedge \mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n)) \wedge q \in \{q_1, \ldots, q_n\}\}.
\end{aligned}$$

and ϵ is defined such that $\forall \varphi \neq \epsilon\ : \mathrm{T}(\varphi) = \mathrm{T}(\epsilon)$. □

Notation 2.1.6. Φ denotes a set of trajectories, possibly including infinite trajectories. Unless mentioned explicitly, all trajectories in the set (except empty trajectories) evolve in the same signal space (have the same trajectory qualifiers, i.e., formally $\forall \varphi, \psi \neq \epsilon \in \Phi\ \ \mathrm{T}(\varphi) = \mathrm{T}(\psi)$).

$\varphi \restriction t$ is shorthand for $\varphi \restriction (0,t]$, where φ is a trajectory and $\restriction t$ is a restriction of a function (sometimes denoted $|_t$). □

Definition 2.1.7 (Projection). Let $\varphi : (0,u] \to W_1 \times \cdots \times W_n$ be a trajectory, such that $\mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n))$. Then a *projection* of the trajectory w.r.t. a trajectory qualifier q_i ($i = 1, \ldots, n$) is the trajectory

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

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

Remark 2.1.8 (Extended projections). Let

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

be a trajectory in $\mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n))$ and let $\mathcal{T}' \subseteq \mathrm{T}(\varphi)$. Then projection for a set of trajectory qualifiers $\mathcal{T}'$ is the trajectory

$$\pi^{\mathcal{T}'}(\varphi) : (0,u] \to W'_1 \times \cdots \times W'_m$$

such that $\mathbb{W}_{\mathcal{T}'} = \left(W'_1 \times \cdots \times W'_m, \left(q'_1, \ldots, q'_m\right)\right)$, $\{q'_1, \ldots, q'_m\} = \mathcal{T}'$ and $\forall q_i \in \mathcal{T}'\ \pi^{q_i}(\varphi) = \pi^{q_i}(\pi^{\mathcal{T}'}(\varphi))$. □

Example 2.1.9 (Trajectories and projections). Let $\mathbb{W}_{\mathrm{BB}} = (\mathbb{R}_+ \times \mathbb{R}, (\mathit{Altitude}, \mathit{Velocity}))$ be a signal space for the bouncing ball (Example 2.1.2). Then the trajectory for the bouncing ball can be defined as a mapping

$$\varphi : (0,t] \to \mathbb{R}_+ \times \mathbb{R}$$

and, e.g., given as

$$\begin{aligned}
\frac{d}{dt}\pi^{\mathit{Altitude}}(\varphi) &= \pi^{\mathit{Velocity}}(\varphi) \\
\frac{d}{dt}\pi^{\mathit{Velocity}}(\varphi) &= -g
\end{aligned}$$

with initial values $\pi^{\mathit{Altitude}}(\varphi)(0) = h_0$ and $\pi^{\mathit{Velocity}}(\varphi)(0) = v_0$, respectively. □

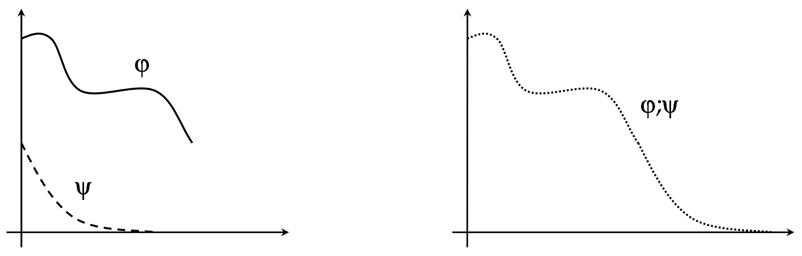


Figure 2.1: Concatenation

If the signal types of two trajectories coincide, they can be concatenated to one trajectory, which is not necessarily smooth.

Definition 2.1.10 (Concatenation of trajectories). Let $\varphi : (0, t] \to W_1 \times \cdots \times W_n$ (where $t \neq \infty$) and $\psi : (0, u] \to W_1 \times \cdots \times W_n$ be trajectories. The *concatenation* of φ and ψ is given by the trajectory

$$\phi; \psi : (0, t + u] \to W_1 \times \cdots \times W_n$$

defined by

$$\varphi\, ; \psi(t') = \begin{cases} \varphi(t'), & 0 < t' \leqslant t \\ \psi(t' - t), & t < t' \leqslant t + u \end{cases}$$

Moreover, ϵ is a neutral element in the concatenation, i.e., for all $\Phi, \varphi \in \Phi \quad \epsilon\, ; \varphi = \varphi = \varphi\, ; \epsilon$. □

Example 2.1.11 (Concatenation). Let φ and ψ be trajectories depicted by the solid and dashed lines on the left side of Figure 2.1, respectively. Then the concatenation $\varphi\, ; \psi$ is a trajectory depicted on the right side of Figure 2.1 by a dotted line. □

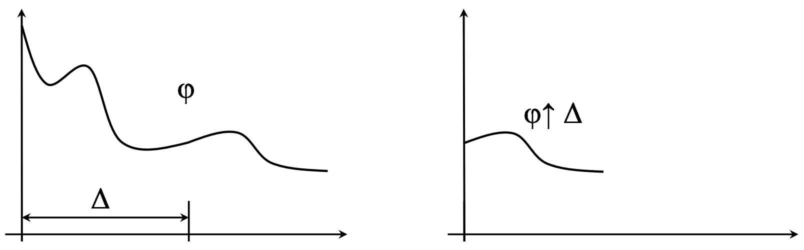


Figure 2.2: Time-shift

For the sake of convenience, a time-shift operation is defined. It displaces a trajectory to "the left" by some time. An example of the time shift by Δ time units is presented in Figure 2.2. The figure on the left side represents the original function, and the figure on the right side represents the function after the time-shift.

Definition 2.1.12 (Time-shift). Let Φ be a set of trajectories and $\varphi : (0,t] \to W_1 \times \cdots \times W_n$ be a trajectory. Then the *time-shift* operator

$$\uparrow : \Phi \times \mathbb{R}_{\geqslant 0} \to \Phi$$

is defined for $t' < t$ as follows:

$$\varphi \uparrow t' : (0, t-t'] \to W_1 \times \cdots \times W_n \text{ such that } \forall u \in (0, t-t'] \quad \varphi \uparrow t'\ (u) = \varphi(t'+u).$$

□

If one trajectory coincides on the signal space with the initial part of another trajectory, it is called a *prefix* of this trajectory.

Definition 2.1.13 (Prefix of trajectory). Let $\varphi : (0,t] \to W_1 \times \cdots \times W_n$ and $\psi : (0,u] \to W_1 \times \cdots \times W_n$ be trajectories, such that $t \leqslant u$. Then φ is a *prefix* of ψ (denoted $\varphi \preceq \psi$), if $\varphi = \psi \upharpoonright t$. Furthermore, if $\varphi \preceq \psi$ and $t < u$, then φ is called a *strict prefix* of ψ and denoted $\varphi \prec \psi$. □

They define a set of trajectories prefixes and a closure of such set.

Definition 2.1.14 (Set of trajectories prefixes). Let Φ be a set of trajectories such that $\forall \chi, \kappa \in \Phi\ \mathrm{T}(\chi) = \mathrm{T}(\kappa)$. Then a *set of trajectories prefixes* is defined as follows $\mathcal{P}ref^{\prec}(\Phi) = \{\varphi \mid \exists \psi \in \Phi, \varphi \prec \psi\} \setminus \Phi$ and the *set of trajectories prefixes minus empty trajectory* as $\mathcal{P}ref^{+} = \mathcal{P}ref^{\prec}(\Phi) \setminus \epsilon$. □

Definition 2.1.15 (Set of trajectories prefixes closure). Let Φ ($\forall \varphi, \psi \in \Phi\ \mathrm{T}(\varphi) = \mathrm{T}(\psi)$) be a set of trajectories. A set that includes all behaviours from Φ and all prefixes of the behaviours from the Φ can be defined as follows $\overline{\Phi} = \Phi \cup \mathcal{P}ref^{\prec}(\Phi)$. The *set of trajectories closure minus empty trajectory and minus infinite trajectories* is defined as follows

$$\overline{\Phi}^{+} = \{\varphi \mid \exists \psi \in \Phi, \varphi \prec \psi, \mathrm{t}(\varphi) \neq \infty\} \setminus \epsilon$$

□

The remainder of a taken trajectory is expressed by trajectory continuation, a supplement to the trajectory prefix.

Definition 2.1.16 (Trajectory continuation). Let $\varphi : (0,t] \to W_1 \times \cdots \times W_n$ and $\psi : (0,u] \to W_1 \times \cdots \times W_n$ be trajectories such that $\psi \prec \varphi$. Then *trajectory continuation* of φ after taking ψ is defined as

$$\varphi \backslash\backslash \psi : (0, t-u] \to W_1 \times \cdots \times W_n,$$

such that

$$\varphi \backslash\backslash \psi = \varphi \uparrow u.$$

□

Trajectory continuations define the remainder of the taken trajectory. A generalised version of it, a *set of trajectory continuations*, singles out a subset of trajectory continuations, i.e., all remainders from the set of trajectories, which have the same initial part.

Definition 2.1.17 (Set of trajectory continuations). Let Φ be a set of trajectories such that $\forall \chi, \kappa \in \Phi \ \mathrm{T}(\chi) = \mathrm{T}(\kappa)$ and ψ be a trajectory or trajectory prefix of some trajectory belonging to the set. Then a *set of trajectory continuations* for ψ is defined as follows

$$\Phi \backslash\!\backslash \psi = \{\varphi \mid \psi \,;\, \varphi \in \Phi\}$$

□

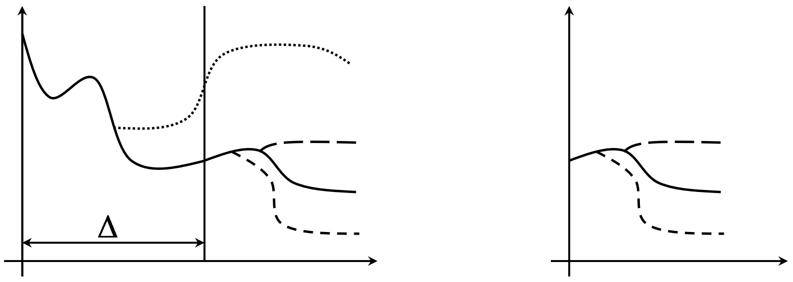


Figure 2.3: Set of continuations

Example 2.1.18 (Set of trajectory continuations). Consider the set of trajectories depicted on the left side of Figure 2.3, consisting of 5 partly coinciding trajectories. Suppose the the trajectory depicted by the solid line is taken for a duration of Δ. Then the set of trajectory continuations for this prefix will include all continuations depicted on the right side of Figure 2.3. Only remainders of trajectories coinciding with the trajectory depicted by the solid line during Δ are part of the continuation set. □

Definition 2.1.19 (Partial prefix). Let H be a set of trajectory qualifiers, and let $\varphi : (0,t] \to W_1 \times \cdots \times W_n$ in $\mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n))$ and $\psi : (0,u] \to W'_1 \times \cdots \times W'_m$ in $\mathbb{W}' = \left(W'_1 \times \cdots \times W'_m, \left(q'_1, \ldots, q'_m\right)\right)$ be trajectories, such that $t \leqslant u$. Let $\mathcal{T} = \mathrm{T}(\varphi) \cap \mathrm{T}(\psi) \subseteq H$.

- Trajectory φ is a *partial prefix* of ψ (denoted $\varphi \preceq^H \psi$), if $\pi^{\mathcal{T}}(\varphi) = \pi^{\mathcal{T}}(\psi \upharpoonright t)$.
- If $\varphi \preceq^H \psi$ and $t < u$, then φ is called a *strict partial prefix* of ψ and denoted $\varphi \prec^H \psi$.
- In case of $t = u$ the trajectories are equal on the coinciding trajectory qualifiers and are called *partially equal* (denoted $\varphi =^H \psi$).

□

The partial prefix relaxes requirements put by the prefix (Definition 2.1.13), i.e., only the projections over coinciding trajectory qualifiers are compared.

Example 2.1.20 (Partial prefix). Consider two trajectories that define the (altitude, velocity) and (altitude, temperature) pairs, respectively. Then one of these trajectories is a partial prefix of another, if the altitude evolves in the same way. It allows to define different aspects of the same object separately and then compose definitions to get a complete specification of the object. $\square$

Based on synchronising trajectory qualifiers, two trajectories can be composed creating a new, "wider", trajectory, such that evolutions of coinciding trajectory qualifiers are merged and non-coinciding parts extend the state space.

Definition 2.1.21 (Composition of trajectories). Let H be a set of synchronising trajectory qualifiers, and let $\varphi : (0,t] \to W_1'' \times \cdots \times W_k''$ in $\mathbb{W} = \left(W_1'' \times \cdots \times W_k'', \left(q_1'', \ldots, q_k''\right)\right)$ and $\psi : (0,u] \to W_1' \times \cdots \times W_m'$ in $\mathbb{W}' = \left(W_1' \times \cdots \times W_m', \left(q_1', \ldots, q_m'\right)\right)$ be trajectories such that $\mathrm{T}(\varphi) \cap \mathrm{T}(\psi) \subseteq H$, $\pi^{\mathrm{T}(\varphi)\cap\mathrm{T}(\psi)}(\varphi) = \pi^{\mathrm{T}(\varphi)\cap\mathrm{T}(\psi)}(\psi)$ and $u \leqslant t$. Then a *composition of trajectories* is a trajectory

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

in $\mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n))$ such that

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

$\square$

They extend composition of trajectories to composition of sets of trajectories.

Definition 2.1.22 (Composition of sets of trajectories). Let H be a set of synchronising trajectory qualifiers, and let Φ and Ψ be sets of trajectories such that, $\forall \varphi, \psi \in \Phi\ \mathrm{T}(\varphi) = \mathrm{T}(\psi)$, $\forall \varphi, \psi \in \Psi\ \mathrm{T}(\varphi) = \mathrm{T}(\psi)$ and $\forall \varphi \in \Phi, \forall \psi \in \Psi$ holds $\mathrm{T}(\varphi) \cap \mathrm{T}(\psi) \subseteq H$. Then a *composition of sets of trajectories* is defined as follows

$$\Phi \times_H \Psi = \{\varphi \times_H \psi \mid \varphi \in \overline{\Phi} \land \psi \in \overline{\Psi} \land \varphi =^H \psi \land \pi^{\mathrm{T}(\varphi)\cap\mathrm{T}(\psi)}(\varphi) = \pi^{\mathrm{T}(\varphi)\cap\mathrm{T}(\psi)}(\psi)\}$$

Moreover, if $\epsilon \in \Phi \lor \epsilon \in \Psi$, then $\epsilon \in \Phi \times_H \Psi$. $\square$

Several different ways will be used to define sets of trajectories. All trajectories are required to have the same qualifiers, i.e., $\forall \varphi, \psi \in \Phi\ \mathrm{T}(\varphi) = \mathrm{T}(\psi)$.

- By listing all trajectories belonging to the set: $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ such that $\forall i, j = 1, \ldots, n\ \mathrm{T}(\varphi_i) = \mathrm{T}(\varphi_j)$.

- By putting restrictions on the already existing set of trajectories: $\Phi \downarrow \mathcal{P}red = \{\varphi \in \Phi \mid \mathcal{P}red(\varphi)\}$, where $\mathcal{P}red$ is a predicate.
- Sometimes it is useful to define conditions on the *end-points* of trajectories or the *exit conditions*. $\Downarrow$ denotes such conditions, as restrictions on the set of trajectories: $\Phi \Downarrow \mathcal{P}red_{\text{exit}} = \{\varphi : (0,u] \to W_1 \times \cdots \times W_n \in \Phi \mid \mathcal{P}red_{\text{exit}}(\varphi(u))\}$. Application of exit conditions is exemplified in Example 2.6.1, where, e.g. $\Downarrow h = 0$ requires that the trajectory finishes at 0 altitude.

  Moreover, an empty trajectory ϵ will denote an instantaneous exit availability.

Trajectory qualifiers can be used to access corresponding parts of trajectories when it is clear from the context. For example , q_i will mean the same as $\pi^{q_i}(\varphi)$ $(i = 1 \ldots n)$ for

$$\varphi : (0,t] \to W_1 \times \cdots \times W_n \text{ with } \mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n)).$$

Furthermore, q_i can be used instead of $\pi^{q_i}(\varphi)(u)$ with $u \in (0,t]$ as a time, when it is clear from the context, e.g., as it is used in definition of $\mathcal{P}red_{\text{exit}}$.

Moreover, the combination of trajectory restrictions and exit conditions is allowed

$$\Phi \downarrow \mathcal{P}red \Downarrow \mathcal{P}red_{\text{exit}} = \{\varphi : (0,u] \to \mathbb{W} \in \Phi \mid \mathcal{P}red(\varphi) \wedge \mathcal{P}red_{\text{exit}}(\varphi(u))\}$$

2.2 Hybrid transition systems

Automata, state-transition diagrams and other similar models are often used to describe the dynamic behaviour of the systems. They consist of states $s \in S$ (with S as a set of states) and some construct, defining changes of the states. Most of the time changes of the states are defined by *transitions*, which are given as a relation (function) over a subset of the Cartesian product of the states $(S \times S)$. Usually transitions are denoted by an arrow, e.g., $(s, s') \in\to$ or $s \to s'$.

Labelled transition systems are a class of transition systems, where transitions are labelled with some *actions* $\mathsf{a} \in \mathcal{A}$ (where $\mathcal{A}$ is a set of actions). The transition relation is defined over subset of $S \times \mathcal{A} \times S$. For $(s, \mathsf{a}, s') \in\to$ they write $s \xrightarrow{\mathsf{a}} s'$.

A hybrid transition system is a labelled transition system with two types of transitions.

Definition 2.2.1 (HTS). A *hybrid transition system* is a tuple $HTS = \langle S, \mathcal{A}, \to, \mathbb{W}, \Phi, \to_c \rangle$, 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* $\varphi : (0, t] \rightarrow W_1 \times \cdots \times W_n$ for $t \in \mathbb{R}_+$, and $t \neq \infty$, $\epsilon \notin \Phi$;
- $\rightarrow_c \subseteq S \times \Phi \times S$ is a *(continuous-time) transition relation.*

They define the following convenient equivalences for transition relation membership

$$s \xrightarrow{\mathsf{a}} s' \iff (s, \mathsf{a}, s') \in \rightarrow$$
$$s \xrightarrow{\varphi} s' \iff (s, \varphi, s') \in \rightarrow_c .$$

The set of discrete action names includes a *silent action*, denoted τ. It does not represent a potential communication and is not directly observable. Silent action may be used to specify a non-deterministic behaviour (as *internal actions* in [24]). □

Remark 2.2.2. Constraints $\epsilon \notin \Phi$ and $\forall \varphi \in \Phi, \mathrm{t}(\varphi) \neq \infty$ can be used to exclude empty and infinite transitions. However, infinite evolutions can be constructed by concatenating finite trajectories. ϵ can used to denote completed trajectories. □

Notation 2.2.3. They adhere to the following notation.

- Greek alphabet symbols (like φ, ψ) will be used to denote trajectories, which are taken on a continuous-time transition.
- Latin alphabet (like a, b) will be used to denote actions.

□

Definition 2.2.4 (Density). *Density* is required for all trajectories

$$s \xrightarrow{\varphi} s' \iff \exists s'', \varphi_1, \varphi_2 : \ \varphi = \varphi_1 \,;\, \varphi_2 \ \wedge \ s \xrightarrow{\varphi_1} s'' \ \wedge \ s'' \xrightarrow{\varphi_2} s'.$$

□

Density allows to split every trajectory into arbitrarily many parts.

Remark 2.2.5 (Labels of continuous-time transitions). Label φ in $s \xrightarrow{\varphi} s'$ is a *semantic object*, viz. the set theoretic graph of the function φ. □

Remark 2.2.6 (Sufficiency of density). The above property of density does not suffice in general, because it allows pathological transition systems, see [18]. However, BHPC cannot describe such pathological cases, so that the density definition for BHPC suffices. □

2.2.1 Bisimulation

One of the main tools to compare systems is *strong bisimulation*. The bisimulation for continuous dynamical systems is presented in [30]. The process algebraic version is nicely explained in [24]. Strong bisimulation requires both subsystems to be able to imitate each other at every step. A strong bisimulation for hybrid transition systems requires both systems to be able to execute the same trajectories and actions and to have the same branching structure.

Definition 2.2.7 (Hybrid strong bisimulation). A binary relation $\mathcal{R} \subseteq S \times S$ on the states is a *hybrid strong bisimulation*, if for all $p, q \in S$, such that $p\,\mathcal{R}\,q$, holds

$$\begin{aligned} p \xrightarrow{\mathsf{a}} p' &\Longrightarrow \exists q' \text{ such that } q \xrightarrow{\mathsf{a}} q' \text{ and } p'\,\mathcal{R}\,q' \\ q \xrightarrow{\mathsf{a}} q' &\Longrightarrow \exists p' \text{ such that } p \xrightarrow{\mathsf{a}} p' \text{ and } p'\,\mathcal{R}\,q' \\ p \xrightarrow{\varphi} p' &\Longrightarrow \exists q' \text{ such that } q \xrightarrow{\varphi} q' \text{ and } p'\,\mathcal{R}\,q' \\ q \xrightarrow{\varphi} q' &\Longrightarrow \exists p' \text{ such that } p \xrightarrow{\varphi} p' \text{ and } p'\,\mathcal{R}\,q'. \end{aligned}$$

□

The first two statements define bisimulation requirements for the discrete actions, and the last two for the continuous-time transitions.

Definition 2.2.8 (Bisimilarity). States p and q are *bisimilar* (denoted $p \sim q$), if there exists a hybrid strong bisimulation $\mathcal{R}$, containing the pair (p, q). □

2.3 Language and operational semantics

2.3.1 Language

To define evolution and interaction of systems, a language and it's semantics, based on hybrid transition systems are introduced. The syntax of language is presented in BNF notation (Backus-Naur form).

$$B ::= \mathbf{0} \;\Big|\; \mathsf{a}.B \;\Big|\; [f \mid \Phi]\,.B \;\Big|\; \sum_{i \in I} B_i \;\Big|\; B \parallel_A^H B \;\Big|\; \mathsf{new}\; w.B \;\Big|\; B\,[\sigma] \;\Big|\; P$$

- **0** is a *deadlock*, the process that does not show any behaviour.
- $\mathsf{a}.B$ is an *action prefix*, where $\mathsf{a} \in \mathcal{A}$ is a discrete action name and B is a process. It first performs a and then engages in B. An action prefix denotes a discrete transition in the underlying hybrid transition system.
- $[f \mid \Phi]\,.B\,(f)$ 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.

- $\sum_{i \in I} B_i$ 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. Binary version of choice is denoted $B_1 + B_2$.
- $B \parallel_A^H B$ 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. Parallel composition defines a new process that executes both processes in parallel forcing trajectory prefixes and actions in A to synchronise. If actions are not in A, they are executed in the interleaving manner, i.e., sequentially in an arbitrary order.
- $\mathsf{new}\ w.B$ is a *hiding* operator, where w is a set of discrete action names and trajectory qualifiers to hide.
- $B[\sigma]$ is a *renaming* operator, where σ is a renaming function. Function σ takes an action name or a trajectory qualifier and changes it. Renaming function for the actions $\sigma : \mathcal{A} \to \mathcal{A}$. For the trajectory qualifiers renaming is defined as $\sigma : \mathcal{T} \to \mathcal{T}$ and it should be injective for the trajectory qualifiers. $B[\sigma]$ behaves as B but with the actions and trajectory-qualifiers renamed according to σ.
- P is a *recursive equation*, where P is a process identifier.

Syntactic functions $\mathfrak{L}(B)$ and $\mathcal{N}(B)$ are used for collecting action and trajectory qualifiers occurring in B, respectively.

Consistent signal flow is required, i.e., only the parallel composition is allowed to change the set of trajectory qualifiers in the process. Renaming operation only renames trajectory qualifiers, but does not change their types. We refer to [11] for the definition of restrictions that ensure consistent signal flow.

2.3.2 Operational semantics of BHPC

This section defined the semantics of the BHPC operators.

Action prefix $\mathsf{a}.B$

Process $\mathsf{a}.B$ defines a process which executes the action a and then behaves as B.

A special *silent action*, denoted τ, is introduced. It does not represent a potential communication and is not directly observable. Silent actions may be used to specify a non-deterministic behaviour (as *internal actions* in [24].

$$\mathsf{a}.B \xrightarrow{\mathsf{a}} B \tag{2.2}$$

In Section 2.5.1 a parametrised version of the action prefix is given. The use of both ordinary and parametrised action prefixes is illustrated in Section 2.6.

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.

$$[f \mid \Phi] \,.B\,(f) \xrightarrow{\varphi} [f' \mid \Phi \backslash\!\backslash \varphi] \,.B\,(\varphi \,;\, f') \qquad \text{for all } \varphi \in \overline{\Phi}^{+} \tag{2.3}$$

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 $\mathrm{t}(\varphi) \neq \infty$ and $\varphi \neq \epsilon$. 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 (Definition 2.1.17). Hotheyver, if a whole trajectory was taken, then the system may continue with the consecutive process with the substituted trajectories (see (2.4)). $(\varphi \,;\, f')$ 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 behaviour for empty trajectories ϵ will be defined later, after the definition for concatenation (2.4) is given.

Notation 2.3.1. The following notation for trajectory prefix can be convenient to express restrictions and exit conditions.

$$\left[q_1, \ldots, q_m \;\middle|\; \Phi \downarrow \mathcal{P}red \Downarrow \mathcal{P}red_{\mathrm{exit}}\right]$$

where

- $q_1, \ldots, q_m$ are trajectory qualifiers, which can be used to access corresponding parts of trajectories.
- As explained in the end of Section 2.1, the set of trajectories can be defined in several different ways.

Definition of trajectory sets directly in the definition of trajectory prefix is allowed. Commas will be used to separate conditions. If required, $\Downarrow$ can be used to separate exit conditions. □

Concatenation

Concatenation extends definition of 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 derivation rules.

$$\frac{B(\varphi) \xrightarrow{\psi} B'}{[f \mid \Phi]\,.B(\varphi) \xrightarrow{\varphi;\psi} B'} \qquad \varphi \in \Phi \tag{2.4a}$$

$$\frac{B(\epsilon) \xrightarrow{\mathsf{a}} B'}{[f \mid \Phi]\,.B(f) \xrightarrow{\mathsf{a}} B'} \qquad \epsilon \in \Phi \tag{2.4b}$$

In (2.4a) it is shown, how to concatenate two trajectories. While (2.4b) defines a situation, where after taking a whole trajectory process continues with an action prefix.

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

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

Choice $\sum\{B(v) \mid v \in I\}$

Choice is a generalised operator on sets of behaviour expressions. To generate the set they allow arbitrary index sets I. It can be thought of as a generalisation of the ordinary process algebraic choice.

$$\frac{B(w) \xrightarrow{\mathsf{a}} B'}{\sum\limits_{v\in I} B(v) \xrightarrow{\mathsf{a}} B'} \qquad w \in I \tag{2.5a}$$

$$\frac{B(w) \xrightarrow{\varphi} B'}{\sum\limits_{v\in I} B(v) \xrightarrow{\varphi} B'} \qquad w \in I \tag{2.5b}$$

In (2.5a) the choice for action prefixes is defined, which is the same as in usual process algebras. Rule (2.5b) tells that choice for trajectories is made before taking a trajectory. Examples from Section 2.6 illustrate the application of choice.

Parallel composition $B_1 \parallel_A^H B_2$

Parallel composition models concurrent evolution of several processes. During the evolution they may interact with each other via synchronisation on discrete and continuous-time transitions. In BHPC synchronisation on identical names is assumed as the basic synchronisation concept. In order to avoid context-dependent interpretations of operators, the set of action names A and the set of trajectory qualifiers H that are subject to synchronisation, are made explicit in the parallel operator $\|_A^H$.

This form of synchronisation implies that parallel components jointly execute identical actions or trajectories with common signal evolutions that occur in their transitions and are subject to synchronisation.

The basic idea of synchronising trajectories is not much different than that of synchronising actions. Let B_1 and B_2 be the processes which can take trajectories

$$\varphi : (0,t] \to W_1' \times \cdots \times W_m' \quad \text{and} \quad \psi : (0,t] \to W_1'' \times \cdots \times W_k''$$

in $\mathbb{W}' = \big(W_1' \times \cdots \times W_m', \big(q_1', \ldots, q_m'\big)\big)$ and $\mathbb{W}'' = \big(W_1'' \times \cdots \times W_k'', \big(q_1'', \ldots, q_k''\big)\big)$, respectively. The static constraint is imposed that $B_1 \|_A^H B_2$ is only theyll-formed iff $\mathfrak{L}(B_1) \cap \mathfrak{L}(B_2) \subseteq A$ and $\mathcal{N}(B_1) \cap \mathcal{N}(B_2) \subseteq H$ (where $\mathfrak{L}$ is a syntactical function, that collects actions names from the process). Let $\mathcal{W}$ be a set of signal domains and let

$$\mathcal{T}' = \mathrm{T}(\varphi) \cap \mathrm{T}(\psi). \tag{2.6}$$

If a set of coinciding trajectory quantifiers is a subset of the synchronisation set

$$\mathcal{T}' \subseteq H \tag{2.7a}$$

and trajectories are the same on the coinciding quantifiers

$$\pi^{\mathcal{T}'}(\varphi) = \pi^{\mathcal{T}'}(\psi), \tag{2.7b}$$

then the resulting trajectory is a synchronised trajectory of $B_1 \|_A^H B_2$ that simultaneously changes the states of B_1 and B_2, defined as

$$\chi : (0,t] \to W_1 \times \cdots \times W_n$$

in $\mathbb{W} = (W_1 \times \cdots \times W_n, (q_1, \ldots, q_n))$ such that

$$\begin{aligned} \mathrm{T}(\chi) &= \mathrm{T}(\varphi) \cup \mathrm{T}(\psi), \\ \pi^{\mathcal{T}'}(\varphi) &= \pi^{\mathcal{T}'}(\psi) = \pi^{\mathcal{T}'}(\chi), \\ \varphi &= \pi^{\mathrm{T}(\varphi)}(\chi), \\ \psi &= \pi^{\mathrm{T}(\psi)}(\chi). \end{aligned}$$

It can also be defined via the composition of trajectories (Definition 2.1.21), i.e., $\chi = \varphi \times_H \psi$. For parallel composition the following SOS rules are defined.

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1', B_2 \xrightarrow{\mathsf{a}} B_2'}{B_1 \|_A^H B_2 \xrightarrow{\mathsf{a}} B_1' \|_A^H B_2'} \quad \mathsf{a} \in A \tag{2.8a}$$

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1'}{\begin{array}{c} B_1 \|_A^H B_2 \xrightarrow{\mathsf{a}} B_1' \|_A^H B_2 \\ B_2 \|_A^H B_1 \xrightarrow{\mathsf{a}} B_2 \|_A^H B_1' \end{array}} \quad \mathsf{a} \notin A \tag{2.8b}$$

$$\frac{B_1 \xrightarrow{\varphi} B_1', B_2 \xrightarrow{\psi} B_2'}{B_1 \parallel_A^H B_2 \xrightarrow{\varphi \times_H \psi} B_1' \parallel_A^H B_2'} \quad \text{(2.6) and (2.7) hold} \tag{2.8c}$$

Rule (2.8b) explains interleaving semantics for the discrete behaviour, when discrete actions names do not coincide. Synchronisation on actions is defined in (2.8a). Rule (2.8c) defines the parallel composition of similar trajectories. Notice that because of density (Definition 2.2.4) no rules for the trajectories with different durations are needed. Parallel composition is illustrated in Examples 2.6.1and 2.6.2.

Hiding new $w.B$

Following the conventions of the process calculus a *hiding* is introduced as a scope restriction operator. new $w.B$ restricts the use of the names w to B. Hiding for discrete actions is just an ordinary hiding. It is worth emphasising that hiding (especially in continuous-time case) should be used carefully, because two different trajectories can easily become observably equivalent, if only equivalent parts of the behaviour are visible. Hiding may easily influence the outcome of parallel composition and choice.

$$\frac{B \xrightarrow{\mathsf{a}} B'}{\mathsf{new}\ w.B \xrightarrow{\tau} \mathsf{new}\ w.B'} \quad \mathsf{a} \in w \tag{2.9a}$$

$$\frac{B \xrightarrow{\mathsf{a}} B'}{\mathsf{new}\ w.B \xrightarrow{\mathsf{a}} \mathsf{new}\ w.B'} \quad \mathsf{a} \notin w \tag{2.9b}$$

$$\frac{B \xrightarrow{\varphi} B'}{\mathsf{new}\ w.B \xrightarrow{\pi^{\mathrm{T}(\varphi)\setminus w}(\varphi)} \mathsf{new}\ w.B'} \tag{2.9c}$$

The first rule states that if an action should be hidden, it is renamed to τ (silent) action. Otherwise (the second rule) nothing changes. The third rules defines hiding for the continuous-time behaviour, i.e., some qualifiers are not visible any more.

Renaming $B[\sigma]$

Renaming operator $B[\sigma]$, where σ is a defined *renaming function*. Renaming of both action and signal names is allowed. The renaming function σ changes trajectory qualifiers, but not their type.

$$\frac{B \xrightarrow{\mathsf{a}} B'}{B[\sigma] \xrightarrow{\sigma(\mathsf{a})} B'[\sigma]} \qquad \frac{B \xrightarrow{\varphi} B'}{B[\sigma] \xrightarrow{\sigma(\varphi)} B'[\sigma]} \tag{2.10}$$

Recursion

The ordinary process algebraic recursion is extended to work with trajectory prefix. It allows to define processes in terms of each other, like in equation $P \triangleq B$, where P is a process identifier and actions and signal types of B are only allowed actions and signal types in P.

$$\frac{B \xrightarrow{\mathsf{a}} B'}{P \xrightarrow{\mathsf{a}} B'} \quad P \triangleq B \qquad\qquad \frac{B \xrightarrow{\varphi} B'}{P \xrightarrow{\varphi} B'} \quad P \triangleq B \tag{2.11}$$

2.3.3 Congruence property

Process algebras usually employ a *congruence* as a basis for systems analysis. A *congruence* for a process algebra is an equivalence relation (i.e, reflexive, symmetric and transitive) that has the substitution property, i.e, equivalent systems can replace each other inside any larger system, without changing the behaviour of that system.

Theorem 2.3.2. *Hybrid strong bisimulation equivalence on HTSs is a congruence w.r.t. the operations of BHPC defined by the in Section 2.3.2.*

For the proof of this theorem we refer to [11].

2.4 Expansion law

The expansion law (Theorem 2.4.2) expresses the parallel composition as a choice of processes (where parallel composition of discrete actions is resolved in the *interleaving* manner).

It is possible to reduce any process in BHPC to a basic form. For processes that do not involve parallel composition it is trivial. Below they show how the parallel composition can be eliminated.

Remark 2.4.1 (Notation). make apparent Substitution is explicitly made apparent in the expansion law and its proofs in [11]. $(\varphi\,;f'/f)$ denotes substitution of f by $\varphi\,;f'$. □

Theorem 2.4.2 (Expansion law). *Let*

$$B = \sum_{i\in I} \mathsf{b}_i.B_i + \sum_{j\in J} \left[f_j \mid \Phi_j\right].B_j, \quad C = \sum_{k\in K} \mathsf{c}_k.C_k + \sum_{l\in L} \left[g_l \mid \Psi_l\right].C_l$$

for some terms B_i, B_j, C_k and C_l, actions b_i and c_k, trajectories $\left[f_j \mid \Phi_j\right]$ and $\left[g_l \mid \Psi_l\right]$ and the corresponding sets of qualifiers names $\mathcal{T}_{\Phi_j}$ and $\mathcal{T}_{\Psi_l}$, finite index sets $I \cap J = K \cap L = \emptyset$. Let

$h_j = \pi^{\mathcal{T}_{\Phi_j}}(h)$ and $h_l = \pi^{\mathcal{T}_{\Psi_l}}(h)$. Then

$$B \parallel^H_A C = \tag{2.12}$$

$$\sum_{\substack{i \in I \\ \mathsf{b}_i \notin A}} \mathsf{b}_i.(B_i \parallel^H_A C) + \sum_{\substack{k \in K \\ \mathsf{c}_k \notin A}} \mathsf{c}_k.(B \parallel^H_A C_k) + \sum_{\substack{i \in I, k \in K \\ \mathsf{b}_i \in A, \\ \mathsf{b}_i = \mathsf{c}_k}} \mathsf{b}_i.(B_i \parallel^H_A C_k) + \tag{2.13}$$

$$\sum_{\substack{j \in J \\ l \in L}} \left[h \mid \Phi_j \times_H \Psi_l \right]. \tag{2.14}$$

$$\left(\left[f'_j \mid \Phi_j \backslash\!\!\backslash h_j \right].B_j\left(h_j \,;\, f'_j / f_j\right) \;\Big\|^H_A\; \left[g'_l \mid \Psi_l \backslash\!\!\backslash h_l) \right].C_l\left(h_l \,;\, g'_l / g_l\right) \right) \tag{2.15}$$

Here Φ_j and Ψ_j are assumed not to contain ϵ. When Φ_j and Ψ_j contain ϵ, then according to (2.3.2)*, they can be rewritten to $[f \mid \Phi \setminus \epsilon].B(f) + B(\epsilon)$.*

For the proof of Theorem 2.4.2 we refer to [11].

2.5 Derived BHPC operators

BHPC is an assembly language for modelling of hybrid systems. [8, 9, 11] present several derived operators to increase usability of the language. Parametrisation of action prefix is given in Section 2.5.1. Section 2.5.2 and 2.5.3 describe idle and $\Delta(\textit{delay})$, defining a trajectory prefix without any observable behaviour and delay, respectively. Finally a guard operator is introduced in Section 2.5.4.

2.5.1 Parametrisation of action prefix

Parametrisation of action prefix is defined similar to [24]

$$\mathsf{a}(v : V).B(v) \triangleq \sum_{v \in V} \mathsf{a}(v).B(v) \tag{2.16}$$

Parametrisation is frequently used for *value passing*, as it is demonstrated in the bouncing ball example (Example 2.6.1).

2.5.2 Idling

Idling in BHPC is treated as a continuous-time signal without any observable behaviour.

$$\mathsf{idle}.B = \left[t \mid \overline{0} \right].B$$

where t is a reserved variable denoting time and the set of trajectories $\overline{0}$ is defined as $\overline{0} = \{(0, t] \to \mathbb{R}_+ \; \forall t\}$. Variable t is not used in the formal definition of the calculus. In its implementation however, it is part of any trajectory prefix, see [11]. It does not manifest any observable behaviour, but reacts as soon as it is invoked by another process, which communicates with the process that follows the idling period.

2.5.3 Delays

In BHPC time and time related constructs, e.g., delays, are treated as a continuous-time signals with rate 1.

$$\Delta(delay).B = \left[t \;\middle|\; t(0) = 0, \dot{t} = 1 \Downarrow t = delay\right].B, \tag{2.17}$$

where t is a reserved variable denoting time. Variable t is not used in the formal definition of the calculus. In its implementation however, it is part of any trajectory prefix, see [11]. In contrast to idling (Section 2.5.2) it contains only one trajectory, and can exit only after completing it. Of course, it can take this trajectory in parts. A delay process does not manifest any observable behaviour for *delay* time units, thereafter the system progresses with the process following delay.

2.5.4 Guard

Sometimes it is useful to check some conditions explicitly, and if they are not satisfied, to stop the progress of process. *Guard* is one of such constructs.

$$\langle \mathcal{P}red(\overline{x})\rangle .B(\overline{x}) = \sum_{\overline{w} \models \mathcal{P}red(\overline{w})} B(\overline{w}) \tag{2.18}$$

Here $\overline{x}$ are process parameters variables. Behaviour is very simple, i.e., if a transition can be taken, then it is taken, if and only if the guard is satisfied.

2.6 Application of BHPC

The next examples illustrate the application of BHPC.

2.6.1 Bouncing ball

Example 2.6.1 (Bouncing ball). A well known example of a hybrid system is a bouncing ball that dissipates its energy with every bounce. In [11] the hybrid automaton for the bouncing ball is depicted by Figure 2.4.

In the figure and the remainder of this example h denotes the altitude of the ball and v denotes its velocity. c and g are constants that denote the loss of energy and gravitational constant, respectively. The ball moves according to the flow conditions (described in the bulb) and variables are reassigned (along the edges) at the bounce ($h = 0$). Evolution of the simple bouncing ball is depicted in Figure 2.5.

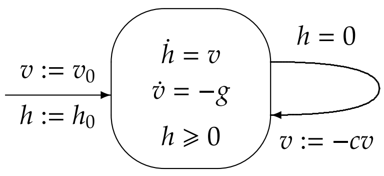


Figure 2.4: A bouncing ball.

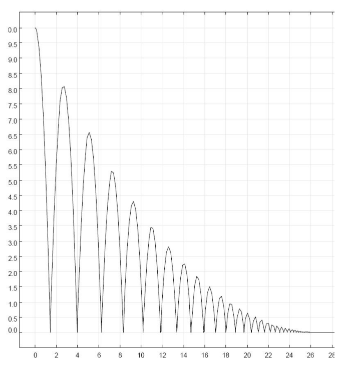


Figure 2.5: Graph for h.

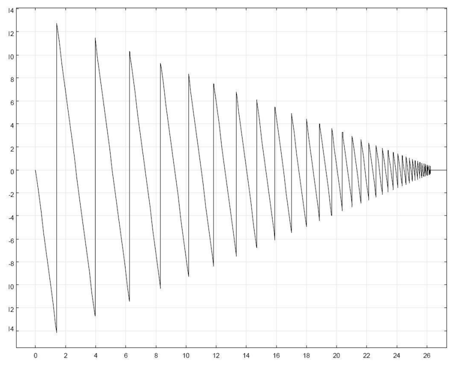


Figure 2.6: Graph for v.

In BHPC the bouncing ball can be defined in the following way:

$$
\begin{aligned}
\mathrm{BB}(h_0, v_0) &\triangleq [h, v \mid \Phi(h_0, v_0) \Downarrow h = 0]\,.\mathrm{BB}(0, -c * v) \\
\Phi(h_0, v_0) &= \{h, v : (0, t] \rightarrow \mathbb{R} \mid \\
&\qquad h(0) = h_0, v(0) = v_0, \dot{h} = v, \dot{v} = -g, h \geqslant 0\}
\end{aligned}
$$

The BHPC description contains definitions for a bouncing ball process and a trajectory set. The bouncing ball process is defined as a symbolic trajectory prefix $[h, v \mid \Phi(h_0, v_0) \Downarrow h = 0]$, followed by a recursive call to the bouncing ball process. The trajectory prefix contains a reference to trajectory set Φ in which the dynamics of the bouncing ball are described by flow conditions. Furthermore, there is only one exit condition defined in the prefix. After execution of the trajectory prefix the system continues with a recursive call to the bouncing ball process. During recursion ($\mathrm{BB}(0, -c * v)$), signals can be updated.

Consider the following extension for the bouncing ball in which the loss of kinetic energy is compensated by a controller.

$$\begin{aligned}
\mathrm{BB}(h_0, v_0)) &\triangleq [h, v \mid \Phi(h_0, v_0)) \Downarrow h = 0]\,.\mathsf{bounce}(c : [0,1]). \\
&\quad [h, v \mid \Phi(0, -cv) \Downarrow v = 0]\,.\mathsf{push}(v : \mathbb{R}).\mathrm{BB}(h, v) \\
\mathrm{Control}(v_0) &\triangleq \mathsf{idle}.\mathsf{bounce}(c : [0,1]). \\
&\quad \mathsf{idle}.\mathsf{push}\,((1-c)\,v)\,.\mathrm{Control}\,((1-c)\,v) \\
\mathrm{System}(h, v) &\triangleq \mathrm{BB}(h_0, v_0) \parallel^{v_0}_{\mathsf{push,bounce}} \mathrm{Control}(v_0)
\end{aligned}$$

The controller pushes the ball and increases its kinetic energy to compensate for the dissipation of energy. Information about the loss of energy is available by synchronisation on discrete actions between the bouncing ball and the controller.

□

2.6.2 Thermostat

Example 2.6.2 (Thermostat). Another well known introductory example of hybrid systems is a thermostat [1]. The room temperature is controlled by a thermostat, which continuously senses the temperature and switches a heater on and off. The temperature changes are defined by the ordinary differential equations. The hybrid automaton is depicted in Figure 2.7.

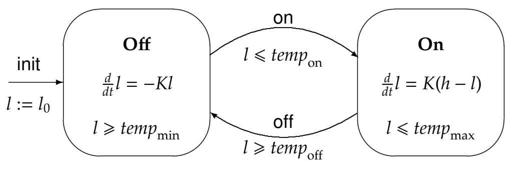

init
$l := l_0$
Off
$\frac{d}{dt}l = -Kl$
$l \geqslant temp_{\mathrm{min}}$
on
$l \leqslant temp_{\mathrm{on}}$
off
$l \geqslant temp_{\mathrm{off}}$
On
$\frac{d}{dt}l = K(h - l)$
$l \leqslant temp_{\mathrm{max}}$

Figure 2.7: A thermostat

When the heater is off, the temperature decreases according to the exponential function $l(t) = \theta e^{Kt}$, where t is time, l is the temperature in the room, θ is the initial temperature, and K is a constant determined by the room. When the heater is on, the temperature increases according to the function $l(t) = \theta e^{-Kt} + h(1 - e^{-Kt})$, where h is a constant that depends on the power of the heater. The temperature should be maintained between $temp_{\mathrm{min}}$ and $temp_{\mathrm{max}}$. Temperatures $temp_{\mathrm{on}}$ and $temp_{\mathrm{off}}$ are the minimal and maximal thresholds, when the heater can be turned on and off, respectively. Evolution of the temperature is depicted in Figure 2.8.

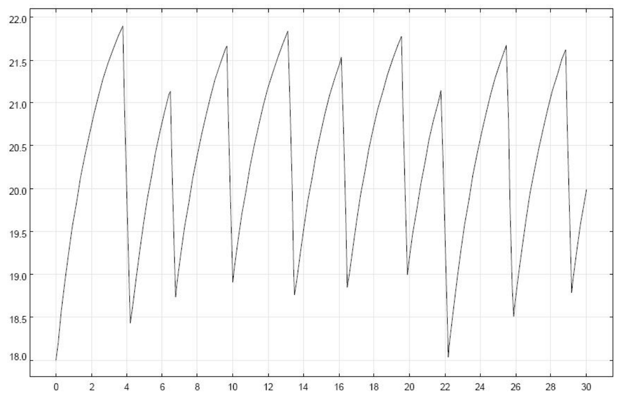


Figure 2.8: Graph for the thermostat.

In BHPC the thermostat can be defined as:

$$\begin{aligned}
\mathrm{Thermostat}(l_0) &\triangleq \mathrm{ThOff}(l_0)\\
\mathrm{ThOff}(l_0) &\triangleq \Big[l \;\Big|\; \Phi_{\mathrm{Off}}(l_0) \Downarrow \mathit{tempOn} \geqslant l \geqslant \mathit{tempMin}\Big].\mathsf{on}.\mathrm{ThOn}(l)\\
\mathrm{ThOn}(l_0) &\triangleq \Big[l \;\Big|\; \Phi_{\mathrm{On}}(l_0) \Downarrow \mathit{tempOff} \leqslant l \leqslant \mathit{tempMax}\Big].\mathsf{off}.\mathrm{ThOff}(l)\\
\Phi_{\mathrm{Off}}(l_0) &= \{l : (0,t] \to \mathbb{R} \mid l(0) = l_0, \dot{l} = -Kl\}\\
\Phi_{\mathrm{On}}(l_0) &= \{l : (0,t] \to \mathbb{R} \mid l(0) = l_0, \dot{l} = K(h-l)\}
\end{aligned}$$

The BHPC description contains two processes:

- In process ThOff the heater is off and the trajectory prefix defines the temperature fall. When the temperature reaches the interval $[\mathit{tempOn}, \mathit{tempMin}]$, the process can perform action on and switch to the process ThOn.
- Process ThOn analogously defines the period of heating.

By adding a controller it is possible to upgrade the thermostat without changing the specification itself. The controller observes temperature and forces the thermostat to switch on and off at exactly *tmpOn* and *tmpOff*, correspondingly:

$$\begin{aligned}
\mathrm{Control}(l_0) \triangleq &\Big[l \;\Big|\; \mathrm{any}(l_0) \Downarrow l = \mathit{tmpOn}\Big].\mathsf{on}.\\
&\Big[l \;\Big|\; \mathrm{any}(l_0) \Downarrow l = \mathit{tmpOff}\Big].\mathsf{off}.\mathrm{Control}(l)\\
\mathrm{UpgradedThermostat}(l_0) \triangleq &\mathrm{Thermostat}(l_0) \parallel^{l}_{\mathsf{on},\mathsf{off}} \mathrm{Control}(l_0)
\end{aligned}$$

where any(l) is a special function that models an observer, i.e., it accepts any behaviour for l. It works only in parallel composition. Technically it just adds exit conditions to the parallel composition of trajectory prefixes.

□

We refer to [11] for more examples on application of BHPC.

2.7 Conclusions

BHPC is a framework for modelling and analysis of hybrid systems that combines principles from control theory and computer science, hereby acknowledging the complexity and importance of both aspects of hybrid system behaviour. The most notable feature of BHPC is that hybrid strong bisimulation property is a congruence, which allows substitution of bisimilar processes. These and other features make the framework a suitable candidate for modelling and analysis of hybrid systems.

BHPC theory is applicable in simulation tools, as shown with a prototype in [11]. We use discrete simulation of BHPC as starting point for hybrid simulation. For discrete simulation we will abstract from the continuous behaviour in BHPC, which is represented by trajectories. The syntactic separation of discrete behaviour and continuous behaviour in BHPC enables such an approach. In the next chapter we investigate abstraction techniques for discrete simulation of BHPC.

3 Discrete Simulation

Most real life systems are too complex to be analysed conveniently without employing simulation tools. In this chapter we will give an introduction to simulation and discrete simulation in Section 3.1. We also present the language that can be used to model hybrid systems for simulation in Section 3.2. Discrete simulation can be performed by abstraction from continuous-time behaviour. Abstraction techniques for discrete simulation are presented in Section 3.3. The remainder of this chapter is used for the definition of the engine. The state of the engine is defined in Section 3.4. The engine is defined by means of a simulation algorithm in Section 3.6. The algorithm consists of two main procedures; one for determining simulation steps and one for executing simulation steps. Both procedures and related definitions are defined in Section 3.5.

3.1 Simulation

In computer science, systems are usually analysed by abstract models in which a mathematical notation is used to represent system behaviour. Most real life systems are too complex to facilitate convenient analysis of their mathematical representation without employing (simulation) tools. Simulation tools can be used to evaluate model behaviour by automatic generation of different model scenarios. For correct simulation, the simulation procedures should be based on the semantics of the modelling framework. Simulation can be used for analysis of existing systems and development of future systems. Hybrid simulation has two precursors: discrete and continuous simulation.

Simulation of discrete systems Discrete simulation in computer science can be based on various types of automata (e.g. [4],[20], [3]), or process algebras (e.g. [7], [32], [14]). In control theory it is based on discrete event systems [34]. Discrete simulation of BHPC only simulates the discrete behaviour of a hybrid model specified in BHPC. It can be used to simulate hybrid system control structures and is useful during early stages of modelling or prototyping. In our work, discrete simulation serves as starting point for development of techniques and tools for hybrid simulation.

Currently verification techniques, like model checking [12], have replaced simulation techniques in computer science. Simulation can only detect presence of errors in a simulated scenario, while many other scenarios may remain unexplored. Model checking can be used to verify the properties of a system for all scenarios and therefore can detect absence of errors. In general, verification is undecidable for hybrid systems and can only be applied to some classes [26].

3.2 Language

For the discrete simulation of BHPC, we simulate the language, defined in Section 2.3.1. In this definition, the set of BHPC process expressions is generated by B. Each word produced by B is a BHPC process expression $p \in B$, i.e. $(\mathsf{one}.P + \mathsf{two}.Q) \in B$.

$$B ::= \mathbf{0} \;\Big|\; \mathsf{a}(v).B \;\Big|\; [f \mid \Phi]\,.B \;\Big|\; \langle \mathcal{P}red \rangle\,.B \;\Big|\; \sum_{i \in I} B_i \;\Big|\; B \parallel_A^H B \;\Big|\; \mathsf{new}\; w.B \;\Big|\; B\,[\sigma] \;\Big|\; P$$

Normal action prefix has been replaced by the parametrised action prefix. Trajectory prefix may be part of the hybrid model for discrete simulation, but we use abstraction to eliminate continuous-time behaviour. Abstraction replaces continuous-time behaviour with a representative that follows the semantics for normal actions.

3.3 Abstraction

In this section we discuss several abstraction techniques. Abstraction converts a hybrid model into a model that only contains discrete behaviour. We distinguish two types of abstraction: one in which continuous time behaviour is represented by internal actions (internal abstraction) and the other in which continuous time behaviour is represented by observable actions (observable abstraction). Both techniques yield a model containing only discrete behaviour. Rules and semantics for discrete actions from discrete algebra can be used to simulate the abstracted model.

3.3.1 Internal abstraction

Internal abstraction is an abstraction technique that converts continuous behaviour into internal actions. Usually τ represents internal actions [24]. Internal abstraction converts (symbolic) trajectory prefixes into action prefixes with τ as action. So $[\varphi]\,.B$ and $[f \mid \Phi]\,.B$ are converted into $\tau.B$. Example 3.3.1 shows internal abstraction for the bouncing ball example.

Example 3.3.1 (Internal abstraction for the bouncing ball). Internal abstraction converts

$$\begin{aligned}
\mathrm{BB}(h_0, v_0) &\triangleq [h, v \mid \Phi(h_0, v_0) \Downarrow h = 0]\,.\mathrm{BB}(0, -c * v)\\
\Phi(h_0, v_0) &= \{h, v : (0, t] \to \mathbb{R} \mid \\
&\qquad h(0) = h_0, v(0) = v_0, \dot{h} = v, \dot{v} = -g, h \geqslant 0\}
\end{aligned}$$

into

$$\begin{aligned}
\mathrm{BB}(h_0, v_0) &\triangleq \tau.\mathrm{BB}(0, -c * v)\\
\Phi(h_0, v_0) &= \{h, v : (0, t] \to \mathbb{R} \mid \\
&\qquad h(0) = h_0, v(0) = v_0, \dot{h} = v, \dot{v} = -g, h \geqslant 0\}
\end{aligned}$$

where τ represents the symbolic trajectory prefix. Although the trajectory set definition (Φ) is not removed from the model, the model does not contain any references to this definition, since all references have been removed during abstraction. □

We can not distinguish between normal internal actions and trajectory abstractions with τ as representative for continuous time behaviour. However, this distinction is required by the semantics of parallel composition, in which continuous components can only synchronise with other continuous components. Without the distinction, continuous components represented by τ can also evolve without continuous synchronisation. We shall illustrate this in Example 3.3.2.

Example 3.3.2 (Abstracting mixed parallel compositions). Consider the parallel composition $\mathsf{a}.B_1 \parallel_A^H [\varphi]\,.B_2$. According to the SOS rules for parallel composition this composition has the following transitions

$$\begin{aligned}
&\mathsf{a}.B_1 \parallel_A^H [\varphi]\,.B_2 \not\rightarrow && , \text{if } \mathsf{a} \in A\\
&\mathsf{a}.B_1 \parallel_A^H [\varphi]\,.B_2 \xrightarrow{\mathsf{a}} B_1 \parallel_A^H [\varphi]\,.B_2 && , \text{if } \mathsf{a} \notin A
\end{aligned}$$

Internal abstraction, using τ converts $\mathsf{a}.B_1 \parallel_A^H [\varphi]\,.B_2$ into $\mathsf{a}.B_1 \parallel_A^H \tau.B_2$, which has the following transitions

$$\mathsf{a}.B_1 \parallel_A^H \tau.B_2 \xrightarrow{\tau} \mathsf{a}.B_1 \parallel_A^H B_2 \quad , \text{if } \mathsf{a} \in A$$

$$\mathsf{a}.B_1 \parallel_A^H \tau.B_2 \xrightarrow{\tau} \mathsf{a}.B_1 \parallel_A^H B_2 \quad , \quad \mathsf{a}.B_1 \parallel_A^H \tau.B_2 \xrightarrow{\mathsf{a}} B_1 \parallel_A^H \tau.B_2 \quad , \text{if } \mathsf{a} \notin A$$

□

As illustrated in Example 3.3.2, simple replacement of continuous behaviour with τ changes system behaviour, which is not what we want. Therefore we must be able to distinguish between internal actions and abstractions and hence we do not use τ as representative for abstraction. The definition of another representative for internal abstraction that allows distinction between internal actions and abstracted continuous behaviour is a solution for this problem. Suppose we define $\tau_@$ to be this representative, then we extend the set of actions to $A \cup \{\tau_@\}$.

Action prefix SOS rule revisited After extending the action set with the representative for internal abstraction, we need to adapt the semantics for action prefix. For normal actions the SOS rule remains unchanged, but note that we removed the abstraction representative from the action set in 3.1.

The semantics for trajectory prefixes state that trajectories can either be executed completely, or partly. Therefore it seems natural to add these characteristics to the new SOS rules for action prefix, in which a trajectory is represented by $\tau_@$. In 3.2 we give the SOS rule for $\tau_@$ that allows unobservable execution of the abstraction representative.

Definition 3.3.3 (SOS Action Prefix for internal abstraction).

$$\mathsf{a}.B \xrightarrow{\mathsf{a}} B \qquad \mathsf{a} \in A \setminus \{\tau_@\} \tag{3.1}$$

$$\tau_@.B \xrightarrow{\tau_@} B + \tau_@.B \tag{3.2}$$

□

The abstracted system produces correct traces when it produces traces which are congruent (up to representation of qualifiers) to the traces of the hybrid system. Unfortunately this is not always the case. Consecutive behaviour for trajectory prefixes of the abstracted models can be executed too early or too late, leading different traces. This may lead to miss of potential deadlocks and simulating irrelevant behaviour, i.e. behaviour that is not in the original model. Example 3.3.4 illustrates this problem.

Example 3.3.4 (Abstraction problems). Consider the parallel composition

$$[\varphi]\,.\mathsf{x}.\mathsf{y}.B_1 \parallel_A^H [\psi]\,.\mathsf{x}.B_2$$

and suppose

$$\begin{array}{ccc} A & = & \{\mathsf{x}\} \\ \varphi & \prec^H & \psi \end{array}$$

Then the composition has the following trace

$$[\varphi]\,.\mathsf{x}.\mathsf{y}.B_1 \parallel_A^H [\psi]\,.\mathsf{x}.B_2 \quad \xrightarrow{\varphi \times_H \psi} \quad \mathsf{x}.\mathsf{y}.B_1 \parallel_A^H [\psi \backslash\backslash \varphi]\,.\mathsf{x}.B_2$$

After execution of $\varphi \times_H \psi$ there are no transitions for the parallel composition, since one component can only synchronise on an action, whereas the other can only synchronise on a trajectory.

While its abstraction

$$\tau_{@}.\mathsf{x}.\mathsf{y}.\tau_{@}.B_1 \parallel_A^H \tau_{@}.\mathsf{x}.B_2$$

according to 3.2 contains the following trace (amongst others)

$$\begin{aligned} \tau_{@}.\mathsf{x}.\mathsf{y}.B_1 \parallel_A^H \tau_{@}.\mathsf{x}.B_2 &\xrightarrow{\tau_{@}} \mathsf{x}.\mathsf{y}.B_1 \parallel_A^H \mathsf{x}.B_2 \\ &\xrightarrow{\mathsf{x}} \mathsf{y}.B_1 \parallel_A^H B_2 \\ &\vdots \end{aligned}$$

This trace is not part of the original system. □

Incorrect traces like the one from Example 3.3.4 can be caused by either executing trajectories too long, or too short. The semantics from 3.2, in which execution of a trajectory can be extended, do not prevent incorrect traces and can be replaced by a simpler version, presented in 3.3.

$$\tau_{@}.B \xrightarrow{\tau_{@}} B \tag{3.3}$$

Although discrete abstraction can disturb event order and can produce irrelevant traces, discrete simulation produces correct traces for all systems without parallel composition and for all parallel compositions with partial equivalent trajectories, i.e. parallel compositions for which the trajectories have the same duration. For these systems event order remains intact during simulation of the abstraction. So, if a system contains parallelism, correct discrete simulation can be performed under the restriction of partial equivalent trajectories. Examples of parallel compositions with partially equivalent components are

- parallel compositions in which a trajectory is in parallel with an idling process
- parallel compositions that have the same exit conditions for the synchronising qualifiers

We have redefined the SOS rule for action prefix and presented two alternatives, see the overview presented in Table 3.1. Both options can not guarantee correct traces in presence of parallelism. While the first option enables premature execution of consecutive behaviour, the second option *also* allows overdue execution of consecutive behaviour. So the second option offers no benefits while its implementation is more complicated, i.e. the engine must decide whether to execute a trajectory partly or completely. Hence we use the SOS rule from the first option for our tool.

Furthermore, we adapt the SOS rules such that a parallel composition synchronises on internal abstraction representatives.

Option 1				
	$\mathsf{a}.B$	$\xrightarrow{\mathsf{a}}$	B	$\mathsf{a} \in A \setminus \{\tau_@\}$
	$\tau_@.B$	$\xrightarrow{\tau_@}$	B	
Option 2				
	$\mathsf{a}.B$	$\xrightarrow{\mathsf{a}}$	B	$\mathsf{a} \in A \setminus \{\tau_@\}$
	$\tau_@.B$	$\xrightarrow{\tau_@}$	$B + \tau_@.B$	

Table 3.1: Overview of alternative SOS rules for action prefix.

Definition 3.3.5 (SOS Parallel Composition for internal abstraction).

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1', B_2 \xrightarrow{\mathsf{a}} B_2'}{B_1 \parallel_A^H B_2 \xrightarrow{\mathsf{a}} B_1' \parallel_A^H B_2'} \quad \mathsf{a} \in A \setminus \{\tau_@\} \tag{3.4a}$$

$$\frac{B_1 \xrightarrow{\tau_@} B_1', B_2 \xrightarrow{\tau_@} B_2'}{B_1 \parallel_A^H B_2 \xrightarrow{\tau_@} B_1' \parallel_A^H B_2'} \tag{3.4b}$$

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1'}{\begin{array}{c} B_1 \parallel_A^H B_2 \xrightarrow{\mathsf{a}} B_1' \parallel_A^H B_2 \\ B_2 \parallel_A^H B_1 \xrightarrow{\mathsf{a}} B_2 \parallel_A^H B_1' \end{array}} \quad \mathsf{a} \notin A \cup \{\tau_@\} \tag{3.4c}$$

□

3.3.2 Observable abstraction

Observable abstraction is an abstraction technique in which continuous-time behaviour is represented by observable actions. Since we want an abstraction that is observable and that enables the distinction between actions and abstractions, we need to define a representative for abstraction that follows the semantics for actions. We use @ as representative for observable abstraction and again we extend the SOS rule for action prefix to include this representative.

Definition 3.3.6 (SOS Action Prefix for observable abstraction).

$$\mathsf{a}.B \xrightarrow{\mathsf{a}} B \qquad \mathsf{a} \in A \cup \{@\}$$

□

Notice the difference with the semantics for internal abstraction; here the representative is observable. This representative will be visible in the set of simulation steps and in simulation traces. We also change the SOS rule for parallel composition such that abstractions synchronise in parallel composition and again representatives will be visible.

Definition 3.3.7 (SOS Parallel Composition for observable abstraction).

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1', B_2 \xrightarrow{\mathsf{a}} B_2'}{B_1 \parallel_A^H B_2 \xrightarrow{\mathsf{a}} B_1' \parallel_A^H B_2'} \quad \mathsf{a} \in A \cup \{@\} \tag{3.5a}$$

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1'}{\begin{array}{c} B_1 \parallel_A^H B_2 \xrightarrow{\mathsf{a}} B_1' \parallel_A^H B_2 \\ B_2 \parallel_A^H B_1 \xrightarrow{\mathsf{a}} B_2 \parallel_A^H B_1' \end{array}} \quad \mathsf{a} \notin A \cup \{@\} \tag{3.5b}$$

□

With this abstraction technique all trajectories are represented by one representative as shown for the extended bouncing ball in Example 3.3.8.

Example 3.3.8 (Observable abstraction for the extended bouncing ball). Observable abstraction converts

$$\begin{aligned}
\mathrm{BB}(h_0, v_0)) &\triangleq [h, v \mid \Phi(h_0, v_0)) \Downarrow h = 0]\,.\mathsf{bounce}(c : [0, 1]). \\
&\quad [h, v \mid \Phi(0, -cv) \Downarrow v = 0]\,.\mathsf{push}(v : \mathbb{R}).\mathrm{BB}(h, v) \\
\mathrm{Control}(v_0) &\triangleq \mathsf{idle}.\mathsf{bounce}(c : [0, 1]). \\
&\quad \mathsf{idle}.\mathsf{push}\,((1 - c)\, v)\,.\mathrm{Control}\,((1 - c)\, v) \\
\mathrm{System}(h, v) &\triangleq \mathrm{BB}(h_0, v_0) \parallel^{v_0}_{\mathsf{push,bounce}} \mathrm{Control}(v_0)
\end{aligned}$$

into

$$\begin{aligned}
\mathrm{BB}(h_0, v_0)) &\triangleq @.\mathsf{bounce}(c : [0, 1]).@.\mathsf{push}(v : \mathbb{R}).\mathrm{BB}(h, v) \\
\mathrm{Control}(v_0) &\triangleq @.\mathsf{bounce}(c : [0, 1]).@.\mathsf{push}\,((1 - c)\, v)\,.\mathrm{Control}\,((1 - c)\, v) \\
\mathrm{System}(h, v) &\triangleq \mathrm{BB}(h_0, v_0) \parallel^{v_0}_{\mathsf{push,bounce}} \mathrm{Control}(v_0)
\end{aligned}$$

where @ is the representative for a trajectory prefix. □

Simulation traces for observable abstraction not only contain actions, but also representatives of trajectories, approaching hybrid traces just a little bit more than internal abstraction techniques. However, it can be very difficult to relate an abstraction representative from the trace to a trajectory from the BHPC expression. Consider Example 3.3.9 where we give a BHPC model for which the representatives in the trace can not be related to the trajectories.

Example 3.3.9 (Relating representatives to trajectories).

$$\begin{aligned}
\mathrm{P} &\triangleq \mathsf{a}.(\mathrm{Q} + \mathrm{R}) \\
\mathrm{Q} &\triangleq [f \mid \Phi]\,.\mathrm{S} \\
\mathrm{R} &\triangleq [f \mid \Psi]\,.\mathrm{S}
\end{aligned}$$

$$\mathsf{S} \triangleq \ldots$$

The trace for this BHPC model would look like $\mathsf{a}, @, \ldots$. It is impossible to deduce whether a trajectory from Φ or Ψ has been executed. □

Relating trajectory expressions Having an unique abstraction representative for each unique trajectory expression yields insight in which trajectory of the model has been simulated. This can be done by adding an unique index to the abstraction representative for each trajectory expression that is not encountered before. There is a special index for delay and idle trajectories. We define $H_@$ to be the set of all abstraction representatives. Now we change the SOS rules for action prefix and parallel composition again to use abstraction representatives from the set $H_@$.

Definition 3.3.10 (SOS Action Prefix for unique observable abstraction).

$$\mathsf{a}.B \xrightarrow{\mathsf{a}} B \qquad \mathsf{a} \in A \cup H_@$$

Where $H_@$ is the set of all abstraction representatives. □

Definition 3.3.11 (SOS Parallel Composition for unique observable abstraction).

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1', B_2 \xrightarrow{\mathsf{a}} B_2'}{B_1 \parallel_A^H B_2 \xrightarrow{\mathsf{a}} B_1' \parallel_A^H B_2'} \quad \mathsf{a} \in A \tag{3.6a}$$

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1', B_2 \xrightarrow{\mathsf{b}} B_2'}{B_1 \parallel_A^H B_2 \xrightarrow{\mathsf{a}\times\mathsf{b}} B_1' \parallel_A^H B_2'} \quad \mathsf{a}, \mathsf{b} \in H_@ \tag{3.6b}$$

$$\frac{B_1 \xrightarrow{\mathsf{a}} B_1'}{\begin{array}{c} B_1 \parallel_A^H B_2 \xrightarrow{\mathsf{a}} B_1' \parallel_A^H B_2 \\ B_2 \parallel_A^H B_1 \xrightarrow{\mathsf{a}} B_2 \parallel_A^H B_1' \end{array}} \quad \mathsf{a} \notin A \cup H_@ \tag{3.6c}$$

□

We shall exemplify the use of unique abstraction for the extended bouncing ball in Example 3.3.12.

Example 3.3.12 (Unique observable abstraction for the extended bouncing ball).

$$\begin{aligned} \mathrm{BB}(h_0, v_0)) &\triangleq @_1.\mathsf{bounce}(c : [0,1]).@_2.\mathsf{push}(v : \mathbb{R}).\mathrm{BB}(h, v) \\ \mathrm{Control}(v_0) &\triangleq @_{\mathsf{idle}}.\mathsf{bounce}(c : [0,1]).@_{\mathsf{idle}}.\mathsf{push}\,((1-c)\,v)\,.\mathrm{Control}\,((1-c)\,v) \\ \mathrm{System}(h, v) &\triangleq \mathrm{BB}(h_0, v_0) \parallel^{v_0}_{\mathsf{push,bounce}} \mathrm{Control}(v_0) \end{aligned}$$

□

The original BHPC model contained four trajectories: two idle trajectories and two distinct trajectories for the bouncing ball process. With unique abstraction we distinguish three distinct trajectories, namely $@_{\mathsf{idle}}$ for the idle trajectories and for the distinct trajectories of the bouncing ball process $@_1$ and $@_2$. Implementation of this technique requires a mechanism to evaluate equality of trajectory expressions and a mechanism for uniquely naming new trajectory expressions.

Remark 3.3.13 (Equality of representatives). Note that in parallel composition we have a distinct rule (3.6b) for unique abstraction prefixes to enable synchronisation of different abstraction representatives. Whereas equality of actions is required for synchronisation of actions, this is not required for representatives of trajectories. For example it should be possible to execute $@_1$ and $@_{idle}$ in parallel in Example 3.3.12. □

All in all, although abstraction does not guarantee correct traces for all hybrid systems, it does help in getting insight in the control structure of a hybrid system. Furthermore observable abstraction gives more insight than internal abstraction. Since our tool set will be used as a "sand box", a place to experiment with BHPC and related developments, we use observable abstraction for the prototype for discrete simulation. In the future the tool will support hybrid simulation, so for further developments of the tool we recommend further investigation whether the gain of insight, by relating trajectory expressions, is worth the effort.

3.4 Engine state

The discrete engine simulates abstract hybrid behaviour by determining and executing transitions for states. Section 2.2 defined a hybrid transition system (HTS) that describes transitions by the SOS rules. Here we extend the HTS so that the states of engine relate to the HTS. An engine state is a triple that consists of a time stamp, a BHPC process expression and a snapshot of the signal space.

Definition 3.4.1. (Engine state).
Let the tuple HTS=$\langle S, \mathcal{A}, \rightarrow, \mathbb{W}, \Phi, \rightarrow_c\rangle$ be a hybrid transition system. Then we define a simulation state $e \in E$ as

$$e = \langle \theta, p, \varphi(\theta)\rangle$$

Where $\theta \in \mathbb{R}$ is the time stamp of e, $p \in S$ is a BHPC process expression of e and $\varphi(\theta)$ is the signal snapshot[1] for signal space $\mathbb{W}$ of s at time θ. □

Example 3.4.2 (Simulation state for the thermostat example). Let $\mathbb{W}_{thermostat} = (\mathbb{R}, (t))$. Suppose that after 300 time units the temperature t ($\in \mathbb{R}$) is 18.1 degrees and the system behaviour can be described by off.ThOff(t). Then the simulation state would be

[1]In continuous systems, usually called continuous state

$$e = \langle 300, \mathsf{off}.\mathrm{ThOff(t)}, 18.1\rangle$$

□

Transitions for engine states depend on the HTS. When a process expression has a transition in the HTS by performing some action, then the engine state containing this process expression can also make a transition by performing this action. A transition changes one, or more state components. Here we describe for each component when it evolves.

Time stamp θ The time stamp has to be updated when time has passed during a transition. Discrete transitions are instantaneous and do not take time. Time can only pass during trajectories. Hence the time stamp can only change after a continuous transition.

Process expression p A BHPC process expression p is an element of the BHPC language, generated by B (Section 2.3.1). The BHPC process expression p has to be updated after each transition according to the SOS rules.

Signal Snapshot $\varphi(\theta)$ The signal snapshot changes after each transition in which the signals have been changed. This can only happen during trajectories. The signal snapshot will not change during discrete simulation.

Since $\mathsf{off}.\mathrm{ThOff(t)} \xrightarrow{\mathsf{off}} \mathrm{ThOff(t)}$ is a transition of our HTS, the engine state from Example 3.4.2 has the following transition

$$\langle 300, \mathsf{off}.\mathrm{ThOff(t)}, 18.1\rangle \xrightarrow{\mathsf{off}} \langle 300, \mathrm{ThOff(t)}, 18.1\rangle$$

3.4.1 Start & end states

After the start of the simulation, the simulation continues until the simulation end time is reached or when it is impossible to take transitions. When the end time is reached, the engine stops the simulation regardless the BHPC process. The simulation can also stop before the simulation end time is reached. This will only happen in absence of transitions for the current state, i.e. when all processes are idling (livelock), or when all processes are semantically equivalent to **0** (deadlock).

Definition 3.4.3. (Start & end states).
Let $\langle\theta, p, \varphi(\theta)\rangle$ be a simulation state, with θ a time-stamp, p a BHPC process expression and $\varphi(\theta)$ a signal snapshot. Then we can define the start state (e_0) for the engine as

$$e_0 = \langle 0, p_0, \perp\rangle$$

Where p_0 denotes the initial process of the BHPC model and all signals are undefined (denoted as $\perp$). At the end of the simulation duration the end state is defined as

$$e_{endtime} = \langle \theta = \theta_{end}, p, \varphi(\theta) \rangle$$

Suppose an end-time θ_{end} for the simulation is given, then the deadlock state is defined as

$$e_{deadlock} = \langle \theta < \theta_{end}, p, \varphi(\theta) \rangle$$

Where the process expression p is strong bisimilar to the stop process, denotes as ($p \sim \mathbf{0}$).

□

3.5 Engine state transitions

For simulation, the engine first determines the set possible transitions. Recall that a transition for an engine state depends on transitions for their process expressions in the HTS. When a process expression has a transition in the HTS by performing some action, then engine states containing this process expression can also have a transition by performing this action.

What we need for simulation is a procedure that determines the possible transitions for a process expression and a procedure that selects a transition and sets the new engine state. Following [32] we define a function Menu that determines the possible transitions and we define a function Next that executes a transition. Both functions follow the semantics of the language given in Chapter 2.3.2.

3.5.1 Menu calculation

Given a BHPC process expression, Menu determines the set transitions, or actually, the set of simulation steps. A simulation step is a tuple that contains an action and a process expression. The process expression represents the behaviour of the current process after performing the action of the tuple.

Definition 3.5.1 (Simulation step). Let HTS = $\langle S, \mathcal{A}, \rightarrow, \mathbb{W}, \Phi, \rightarrow_c \rangle$ be a hybrid transition system, $p, p' \in S$ be BHPC process expressions, $\mathsf{a} \in \mathcal{A}$ an action and $p \xrightarrow{\mathsf{a}} p' \in \rightarrow$ be a transition of the hybrid transition system. Then $(\mathsf{a}, p') \in T$ is a simulation step of the engine, where T represents the set of simulation steps. □

Letting Menu determine pairs (simulation steps) of possible actions and related process expressions for the consecutive behaviour has considerable benefits[32]. There is no need to calculate the process expression for the next engine after selecting an action for simulation because it is already available as part of the simulation step. Determining the process expression in advance is trivial, since consecutive behaviour is part of action prefixes from which actions are determined. Moreover determining the expression after selecting an action can cause difficulties, such as illustrated in Example 3.5.2.

Example 3.5.2 (Storing process expressions). Let $(\mathsf{a}.P + \mathsf{a}.Q)$ be a choice process for which we want to determine the set of possible engine transitions. Storing simulation steps results in $\{(\mathsf{a}, P), (\mathsf{a}, Q)\}$. The process expression for the next state is available when selecting an action.

However, without storing process expression we can choose an action from the set $\{\mathsf{a}\}$. The process expression for the next state needs to be determined and could either be P, or Q and this non-determinism must be solved somehow. $\square$

To determine the set of simulation steps for BHPC processes, we define a recursive function that uses the SOS rules to decompose the process up to the level where actions are described. For most BHPC processes, the definition of such a function is a straightforward application of the SOS rules, but for parallel composition parallelism must be resolved. Chapter 2.4 describes how parallel composition for discrete actions is resolved in an interleaving manner using an expansion law. We use a discrete version of this expansion law for discrete simulation, which does not contain continuous-time behaviour.

Theorem 3.5.3 (Discrete Expansion law). *Let*

$$B = \sum_{i \in I} \mathsf{b}_i.B_i \quad \textit{and} \quad C = \sum_{k \in K} \mathsf{c}_k.C_k$$

for some terms B_i and C_k, actions b_i and c_k, finite index sets I and L. Then

$$B \parallel_A^H C = \sum_{\substack{i \in I \\ \mathsf{b}_i \notin A}} \mathsf{b}_i.(B_i \parallel_A^H C) + \sum_{\substack{k \in K \\ \mathsf{c}_k \notin A}} \mathsf{c}_k.(B \parallel_A^H C_k) + \sum_{\substack{i \in I, k \in K \\ \mathsf{b}_i \in A, \\ \mathsf{b}_i = \mathsf{c}_k}} \mathsf{b}_i.(B_i \parallel_A^H C_k)$$

Note that application of the expansion law requires normal form of the parallel composition components. We define a discrete version of the BHPC normal form presented in [11].

Definition 3.5.4 (Discrete BHPC normal form). A BHPC process expression $p \in B$ is in normal form when it has the form $p = \sum_{i \in I} \mathsf{b}_i.B_i$ $\square$

Transforming processes into normal form and determining the set of simulation steps are similar procedures, are both recursive functions that determine the set of action prefixes by using the SOS rules to decompose a process up to the level where actions are described. The difference is that for determining the set of simulation steps, action prefixes need to be converted into simulation steps.

We define a function $\textsc{Menu}_{NF}$ that converts processes into normal form while determining the set of simulation steps. Therefore we do not convert action prefix processes into simulation steps until $\textsc{Menu}_{NF}$ has finished. We define a function $\textsc{Menu}$ that converts a set of action prefix processes into a set of simulation steps.

Definition 3.5.5 ($\textsc{Menu}$ for BHPC). Let T be the set of transition steps, $(\mathsf{a}\,.B)$ an action prefix with $\mathsf{a} \in A \cup \{@\}$ and let N be the set of BHPC process expressions in normal form such that $\forall p \in N \bullet p = \sum\limits_{i \in I} \mathsf{b}_i.B_i$ for BHPC process expressions p. Then we define $\textsc{Menu}\colon N \to T$ as

$$\textsc{Menu}(N) = \{(\mathsf{a}, B) \mid (\mathsf{a}.B) \in N\}$$

□

Definition 3.5.6 ($\textsc{Menu}_{NF}$ for BHPC). Let B be the set of BHPC process expressions, $\mathsf{a} \in \mathcal{A}$ be an action and T be the set of simulation steps. Then we define the function $\textsc{Menu}_{NF} : B \to T$ as

$$\textsc{Menu}_{NF}(B) = \{(\mathsf{a}.B') \mid B \xrightarrow{\mathsf{a}} B'\}$$

□

We present $\textsc{Menu}_{NF}$ for all BHPC process types in Table 3.2. See Appendix B for detailed derivations for the definitions in the table. Subsequently Example 3.5.7 illustrates how $\textsc{Menu}_{NF}$ can be applied on a choice process using the BHPC SOS rules.

$$
\begin{array}{lcl}
\textsc{Menu}_{NF}(\mathbf{0}) & = & \varnothing \\
\textsc{Menu}_{NF}(\mathsf{a}.B) & = & \{(\mathsf{a}.B)\} \\
\textsc{Menu}_{NF}([\varphi]\,.B) & = & \{(@.B)\} \\
\textsc{Menu}_{NF}(\sum\limits_{v \in I} B(v)) & = & \bigcup\limits_{v \in I} \textsc{Menu}_{NF}(B(v)) \\
\textsc{Menu}_{NF}(B \parallel_A^H C) & = & \textsc{Menu}_{NF}\Big(\sum\limits_{\substack{i \in I \\ \mathsf{b}_i \notin A \cup \{@\}}} \mathsf{b}_i.\,\big(B_i \parallel_A^H C\big)\Big) \\
 & \cup & \textsc{Menu}_{NF}\Big(\sum\limits_{\substack{k \in K \\ \mathsf{c}_k \notin A \cup \{@\}}} \mathsf{c}_k.\,\big(B \parallel_A^H C_k\big)\Big) \\
 & \cup & \textsc{Menu}_{NF}\Big(\sum\limits_{\substack{\mathsf{a}=\mathsf{b}_i=\mathsf{c}_k \\ \mathsf{a} \in A \cup \{@\}}} \mathsf{a}.\,\big(B_i \parallel_A^H C_k\big)\Big) \\
 & & \quad \textbf{where} \quad B = \sum\limits_{i \in I} \mathsf{b}_i.B_i \quad \text{and} \quad C = \sum\limits_{k \in K} \mathsf{c}_k.C_k \\
\textsc{Menu}_{NF}(P) & = & \textsc{Menu}_{NF}(B) \qquad \textbf{where } P \triangleq B \\
\textsc{Menu}_{NF}(B\,[\sigma]) & = & [\sigma]\,(\textsc{Menu}_{NF}(B)) \\
\textsc{Menu}_{NF}(\mathsf{new}\ w.B) & = & [\tau/e_1, \ldots, \tau/e_n]\,(\textsc{Menu}_{NF}(B)) \\
\textsc{Menu}_{NF}(\langle \mathit{Pred} \rangle.B) & = & \begin{cases} \textsc{Menu}_{NF}(B) & \text{, if } \mathit{Pred} \text{ is satisfied} \\ \varnothing & \text{, otherwise} \end{cases}
\end{array}
$$

Table 3.2: $\textsc{Menu}_{NF}$ for (discrete) BHPC processes

Note that in Table 3.2 a trajectory prefix is converted into an action prefix by $\textsc{Menu}_{NF}$. In this action prefix '@' represents abstraction of continuous-time behaviour. Also note that independent evolution for parallel components is only allowed for actions. Abstracted components are only allowed to synchronise with abstracted components.

Example 3.5.7 (Application of $\textsc{Menu}_{NF}$). Applying the $\textsc{Menu}_{NF}$ definition for BHPC, where B is a choice process $(\mathsf{one}.Test + \mathbf{0})$, we first get

$$\textsc{Menu}_{NF}(\mathsf{one}.Test + \mathbf{0}) = \{(\mathsf{a}.B') \mid (\mathsf{one}.Test + \mathbf{0}) \xrightarrow{\mathsf{a}} B'\}$$

Then using the SOS rule for choice

$$\frac{B\,(w) \xrightarrow{\mathsf{a}} B'}{\sum\limits_{v \in I} B\,(v) \xrightarrow{\mathsf{a}} B'} \qquad w \in I$$

results in applying $\textsc{Menu}_{NF}$ on all choices

$$\textsc{Menu}_{NF}(\mathsf{one}.Test + \mathbf{0}) = \textsc{Menu}_{NF}(\mathsf{one}.Test) \cup \textsc{Menu}_{NF}(\mathbf{0})$$

For which we can use the SOS rule for action prefix $\mathsf{a}.B \xrightarrow{\mathsf{a}} B$ to derive

$$\mathsf{one}.Test \xrightarrow{\mathsf{one}} Test$$

For $\mathbf{0}$ there is no SOS rule, hence

$$\begin{aligned}
\textsc{Menu}_{NF}(\mathsf{one}.Test) &= \{(\mathsf{a}.B') \mid \mathsf{one}.Test \xrightarrow{\mathsf{a}} B'\} \\
&= \{(\mathsf{one}.Test) \mid \mathsf{one}.Test \xrightarrow{\mathsf{one}} Test\} \\
\textsc{Menu}_{NF}(\mathbf{0}) &= \{(\mathsf{a}.B') \mid \mathbf{0} \xrightarrow{\mathsf{a}} B'\} \\
&= \varnothing
\end{aligned}$$

So

$$\textsc{Menu}_{NF}(\mathsf{one}.Test_1 + \mathbf{0}) = \{(\mathsf{one}.Test_1)\}$$

□

In summary, $\textsc{Menu}_{NF}$ is a recursive procedure that decomposes process expressions following the BHPC semantics and collects action prefixes. The set of action prefixes for a process at any point in the calculation represents the normal form of that process. The set is converted into a set of simulation steps by $\textsc{Menu}$. The set of simulation steps represents the possible transitions from the current to the next engine state. We not only store the action of an action prefix in the simulation step, but we also store the associated process expression. Therefore after selecting an action for simulation, the process expression for the next state is also known. Due to the normal form, our approach differs from [32] where action prefixes are immediately are converted into simulation steps. As a result the expansion law can be easily applied to resolve parallelism.

3.5.2 Execution of transition steps

Next executes simulation steps and gives the next simulation state using a simulation step and the current simulation state. Recall that simulation steps, calculated by Menu, already contain a BHPC process for the next state. Therefore we only need to determine the new time stamp and signal space for the next state.

For the discrete simulator we only consider discrete transitions. Discrete transitions are transitions for which the event is an action, As described before, the time stamp and signal space are unaffected by discrete transitions.

Definition 3.5.8 (Next). Let HTS = $\langle S, \mathcal{A}, \rightarrow, \mathrm{W}, \Phi, \rightarrow_c\rangle$ be a hybrid transition system, $p, p' \in S$ be BHPC process expressions and $\mathsf{a} \in \mathcal{A}$ be an action. Suppose (a, p') is a simulation step, selected from the set of possible transitions T calculated by Menu and suppose $\langle \theta, p, \varphi(\theta)\rangle \in E$ is the current simulation state of the engine from the set of engine states E. Then we define the function Next : $T \times E \rightarrow E$, such that

$$\textsc{Next}\big((\mathsf{a}, p'), \langle \theta, p, \varphi(\theta)\rangle\big) \quad = \quad \langle \theta, p', \varphi(\theta)\rangle \qquad \text{, if } (p, \mathsf{a}, p') \in \rightarrow$$

□

3.6 Discrete engine

Here we present the discrete simulation engine that only captures the fundamental functionality, namely engine state definition, a function to determine the possible transitions for a simulation state and a function to execute transitions. Similar to [23], we define our simulator by a simulation algorithm in pseudo code.

```
BHPC_Simulate()
1  $e \leftarrow \langle 0, p_0, \varphi(0)\rangle$
2  $T \leftarrow \textsc{Menu}(\pi^p(e))$
3  while $T \neq \emptyset$
4  do $(\mathsf{a}, p') \leftarrow \textsc{Select}(T)$
5      $e' \leftarrow \textsc{Next}((\mathsf{a}, p'), e)$
6      $T \leftarrow \textsc{Menu}(Menu_{NF}(\pi^p(e'))$
```

Where

- $e, e' \in E$ represent engine states
- p_0 represents the initial process expression
- $\pi^p(e)$ is the process expression of the engine state (by projection). Suppose $e = \langle \theta, \mathsf{a.b.P}, \varphi(\theta)\rangle \in E$ is an engine state, then $\pi^p(\langle \theta, \mathsf{a.b.P}, \varphi(\theta)\rangle) = \mathsf{a.b.P}$
- T is the set of transitions

- MENU, MENU$_{NF}$ and NEXT are functions that determine the set of simulation steps, the set of action prefixes and the next engine state respectively
- SELECT is a way to select a simulation step (manually, automatically, or from batch) from the set of simulation steps T

The algorithm consists of an initialisation part (1,2) and a simulation loop (4-6). A more detailed description of the procedures can be found in Chapter 4.

Remark 3.6.1 (Abstraction). Note that abstraction is not a part of the algorithm. Abstraction is performed during the execution of the algorithm by MENU$_{NF}$ by converting trajectory prefixes into abstraction prefixes. □

3.7 Conclusions

We have presented several techniques to abstract from continuous-time behaviour by using representatives for trajectories that are treated like actions. These techniques only produce correct traces for hybrid systems without parallelism. Continuous behaviour can be converted into internal actions (internal abstraction), or into observable actions (observable abstraction). Distinction between normal actions and representatives of continuous-time behaviour is essential for correct abstraction of parallel composition in BHPC.

Our prototype for discrete simulation has been developed as a tool for experimenting with BHPC. Observable abstraction gives more insight and hence we have chosen to use observable abstraction for our prototype. In future tool developments this abstraction technique could be extended by relating abstraction representatives to trajectory expressions. Further investigation is needed to evaluate costs and benefits of this extension.

The definition of the discrete engine is given by the simulation algorithm. The main procedures of this algorithm are MENU (determines simulation steps) and NEXT (executes simulation steps). A simulation step is a tuple of an action and a behaviour expression. The behaviour expression denotes the consecutive behaviour expression for the next state, after execution of the related action. Letting MENU determine simulation steps rather than just actions prevents recalculation of consecutive behaviour after selection of an action from the set determined by MENU. These procedures are all based on [32]. The modular definition of the engine, in which the main procedures are encapsulated, facilitates convenient extensions. Extension to hybrid simulation only focusses on parts of the MENU and NEXT components, while the simulation algorithm itself remains unchanged, we refer to Chapter 6 for more details.

4 Design & Implementation

Bhave is a collection of tools for the analysis of hybrid systems[1]. Bhave is an acronym for *"BHPC's hybrid and visual engine"*. Discrete Bhave is the discrete engine that simulates the discrete behaviour of a hybrid system. The discrete engine is built as a hybrid "sand-box", a place to experiment with BHPC and related developments, rather than industrial tool prototype. Consequently, the architecture and implementation of the tool are being designed in such a way that it is easy to accommodate the changes in the calculus and to test the algorithms developed for hybrid systems in the BHPC framework. This chapter presents the design and implementation details of Discrete Bhave.

First we derive the architecture for the discrete engine from the architecture of Bhave from [11]. For the sake extendibility and generality, we have decomposed the engine, e.g. there is a separate package for event handling that contains all BHPC related methods. Most notably, it contains the implementation of MENU described in Chapter 3. The decomposition of the engine can be found in 4.2. We provide technical details for the implementation in Section 4.3. The remainder describes the implementation details for our prototype. In Section 4.4 we give the details for the engine and in Section 4.5 we describe the details for the event handler.

[1]`http://fmt.cs.utwente.nl/tools/bhave`

4.1 Discrete Bhave architecture

The purpose of the engine is to support the analysis of hybrid systems specified in BHPC. We distinguish two main functions for the engine, namely to simulate models and to yield simulation results. For simulation of hybrid systems, the engine must be able to interpret BHPC models. States are used to depict the system at a particular time and during simulation the engine must be able to manipulate stored states. The engine should determine simulation steps for the current state and execute a selected simulation step. To yield simulation results the engine must be able to build a simulation trace by writing executed simulation steps to a trace file. In Figure 4.1 we give an overview of the required functionality for the engine.

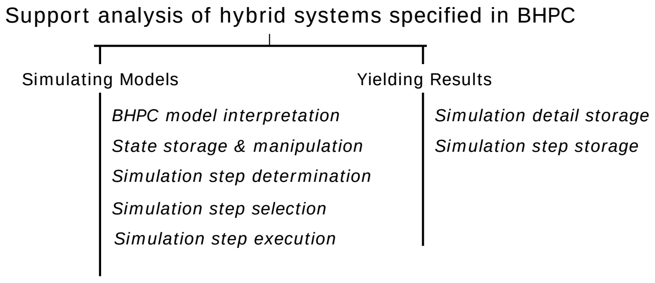

Support analysis of hybrid systems specified in BHPC
Simulating Models
Yielding Results
BHPC model interpretation
State storage & manipulation
Simulation step determination
Simulation step selection
Simulation step execution
Simulation detail storage
Simulation step storage

Figure 4.1: Functionality of Discrete Bhave.

The architecture of Discrete Bhave is based upon the architecture for the Bhave tool set, presented in [11]. For Discrete Bhave we have restricted the architecture to components that are strictly necessary for discrete simulation. For example the connection to the DAE/ODE solvers for the discrete engine is not included in the architecture of Discrete Bhave. The architecture for Discrete Bhave can be found in Figure 4.2, where only the parts relevant for the discrete engine are highlighted.

The central component in Figure 4.2 is the simulation control centre, which in our case is the discrete engine. The engine receives an internal specification with which system behaviour is simulated. The engine uses a library of executable routines to determine the set of simulation steps for the current engine state. The user selects an event from this set and the engine will execute the selected event and engine adds the related transition to a trace file. In the architecture we distinguish the following components that are relevant for the development Discrete Bhave.

Internal Specification The Bhave compiler BHPCC [33] converts a hybrid system model, represented in ASCII, into an internal specification for the simulation control centre (discrete engine). The discrete engine is able to derive all necessary information for simulation from the internal specification, e.g. the initial process

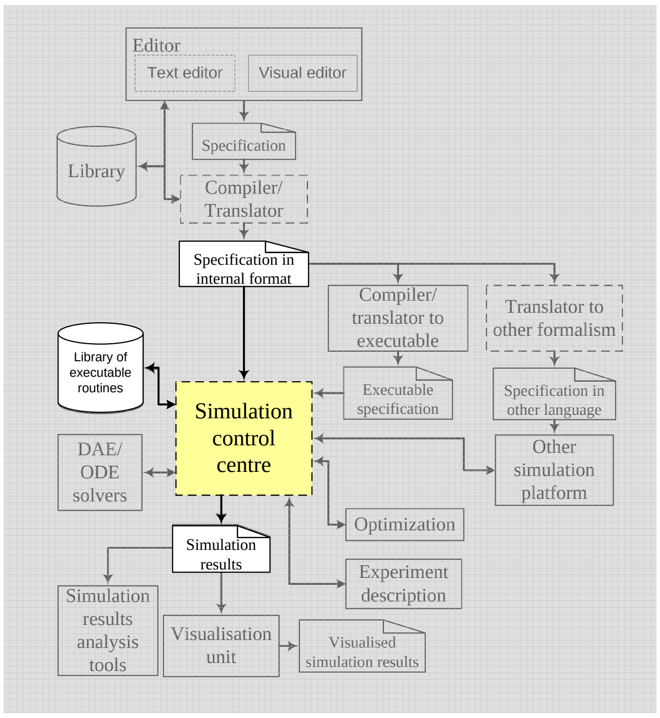

Editor
Text editor
Visual editor
Specification
Library
Compiler/
Translator
Specification in internal format
Compiler/ translator to executable
Translator to other formalism
Library of executable routines
Simulation control centre
Executable specification
Specification in other language
DAE/ ODE solvers
Other simulation platform
Optimization
Simulation results
Experiment description
Simulation results analysis tools
Visualisation unit
Visualised simulation results

Figure 4.2: Overview of the architecture.

and the process definition for a recursive process. The internal representation for BHPC models can be found in Appendix C.

Library of executable routines The library determines the set of simulation steps for the current BHPC process expression.

Simulation Results Discrete Bhave writes engine states and engine state transitions to a trace file.

Simulation Control Centre The simulation control centre is the central part of the architecture. It is the connection between the internal representation, the library of executable routines and simulation results. The simulation control centre is represented by Discrete Bhave in the context of this thesis.

A complete description of the Bhave tool set architecture can be found in [11].

4.2 Decomposition of the engine

In Chapter 3.6 we defined a discrete engine.

```
BHPC_SIMULATE()
1  $e \leftarrow \langle 0, p_0, \varphi(0) \rangle$
2  $T \leftarrow \textsc{Menu}(\pi^p(e))$
3  while $T \neq \emptyset$
4  do $(\mathsf{a}, p') \leftarrow \textsc{Select}(T)$
5      $e' \leftarrow \textsc{Next}((\mathsf{a}, p'), e)$
6      $T \leftarrow \textsc{Menu}(Menu_{NF}(\pi^p(e'))$
```

Where

- $e, e' \in E$ represent engine states
- p_0 represents the initial process expression
- $\pi^p(e)$ is the process expression of the engine state (by projection)
- T is the set of transitions
- MENU, MENU$_{NF}$ and NEXT are functions that determine the set of simulation steps, the set of action prefixes and the next engine state respectively
- SELECT is a way to select a simulation step (manually, automatically, or from batch) from the set of simulation steps T

In this section we will give a decomposition of the engine into a collection of procedures and data structures. Figure 4.3 depicts the decomposition of the engine into procedures. Note that the set of simulation steps is determined by the event routines.

4.2.1 Engine procedures

We distinguish the following procedures

Initialisation procedure The initialisation procedure sets the time, process and signal space for the initial engine state. After setting the initial engine state, the initialisation procedure invokes the event routine to determine the set of simulation steps for the initial engine state.

Simulation loop Selects an event for simulation from the set of simulation steps and subsequently executes the event which sets the new engine state. Transitions to new engine states are written to the trace file. After setting the new engine state, the initialisation procedure invokes the event routine todetermine the set of simulation steps for the new engine state.

Event routine Contains the implementation of MENU and MENU$_{NF}$ and hence determines the set of simulation steps.

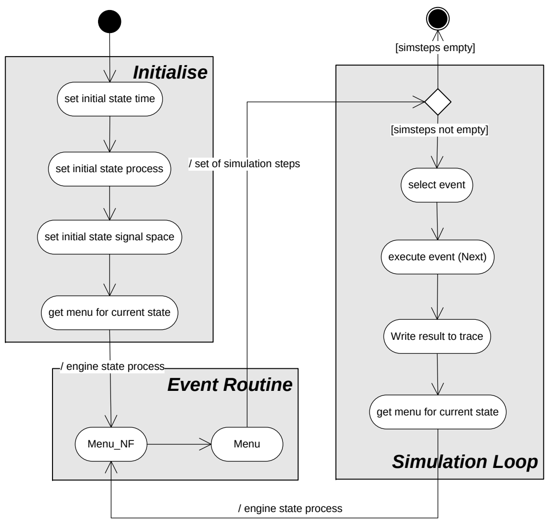


Figure 4.3: Decomposition of the engine.

4.2.2 Data types

Simulation of system behaviour can be seen as evolution of system variables and we use states to depict the system at a particular time. Hence we can represent system evolution by a state machine. To evolve from state to state the engine must be able to determine possible transitions and execute them.

Simulation State The state of the engine that describes the system at a particular time. In Chapter 3 we will define the engine state as triple with a time stamp, a BHPC process and variables.

Simulation Step Transition from the current engine state to the next engine state by some event. Simulation steps are stored by the Results Procedure and together form simulation results.

4.2.3 Software architecture

A description of the software structure can be found in Appendix D. We give a global description of the engine decomposition into packages, classes and methods. A detailed description of the software can be found in the documentation, see [2].

[2]`http://fmt.cs.utwente.nl/tools/bhave/`

4.3 Technical details

Table 4.1 gives an overview of the implementation details.

Language	C++
Compiler	Microsoft (R) 32-bit C/C++
	Optimizing Compiler Version 12.00.8168 for 80x86
Documentation	Doxygen 1.4.6
XML parser	Tiny XML 2.4.1

Table 4.1: Overview implementation details.

Language The engine is implemented in C++, with the standard libraries of Visual Studio 6 for win32 applications. Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86.

Documentation The Documentation is written with Doxygen[3]. Doxygen is a documentation system that can be used for different languages, among which C++. It can generate documentation in various formats.

XML parser TinyXml[4] is an OS independent XML parser, written in C++. It is a very simple and small parser that can be easily integrated by adding some header and cpp files.

4.3.1 Communication

An overview of the communication for the discrete engine is given in Table 4.2.

Component	Type	
Internal Specification	Input:	XML
Simulation Results	Output:	Text, XML
Event Routines	Control:	RPC

Table 4.2: Engine communication details.

Internal specification The BHPC compiler (BHPCC) [33] is the Bhave tool set component that converts BHPC models into an internal specification. The internal specification is written in XML format, we refer to Appendix C for a detailed description of the internal representation.

[3]`http://www.stack.nl/~dimitri/doxygen/`
[4]`http://www.grinninglizard.com/tinyxml/`

Simulation results The simulation results for the engine prototype are nothing more than a trace of transitions made during simulation. We refer to Chapter 5 and Appendix E for examples of simulation traces. The trace file consists of a sequence of all executed events (actions or abstractions) and related BHPC expression for the consecutive state. In the trace events are indicated by the keyword "`EVENT:`" and BHPC expressions by the keywords "`ENGINE STATE PROCESS`". Moreover BHPC expressions are given in XML format. For future developments we propose visual results for hybrid simulation in Chapter 6.3 by using message sequence plots (MSP), a combination of message sequence charts and plots that captures hybrid behaviour.

Event routines Menu calculation is performed by the event handler of the Event Routines package. With remote procedure calls (RPC) the `getMenu` method of the event handler is invoked with a process for which the menu is to be determined. The `getMenu` method returns the list of simulation steps as defined in Chapter 3.5.1. The package further only contains methods for menu calculation.

4.4 Engine procedures

The simulation procedure starts with initialisation of the engine state. Then the Event Handler is invoked to determine the set of simulation steps for the process expression of the current engine state. In the simulation loop, a simulation step from the set is selected and executed by Next, which sets the new engine state. Subsequently, the Event Handler is invoked again to determine the set of simulation steps for the process expression of the updated current engine state. Then, as long as there are simulation steps to be executed and the simulation end time is not over, the loop is repeated.

4.4.1 Data types

The engine needs the following data types for simulation.

model Stores the description (in XML) of the hybrid system.

currState Stores the current state of the engine. Recall that the engine state is described by three components, namely a timestamp, a process expression and a signal space.

simMode Stores the simulation mode of the engine: manual, automatic, or batch.

simDuration Stores the allowed duration of the simulation.

eventHandler Contains Menu and Menu_{NF} for calculation of simulation steps.

```
types
        record EngineState
                time: Time;
                proc: Process;
                space: SignalSpace;
        end

        enum SimMode = eManual, eRandom, eBatch;

        var model: XML;
        var currState: EngineState;
        var simMode: SimMode;
        var simDuration: Time;
        var eventHandler: EventHandler;
end
```

4.4.2 Simulation algorithm

We present the simulation algorithm for discrete simulation. Since Menu determines the simulation steps for continuous behaviour, this simulation also applies for hybrid simulation. The difference in adding continuous behaviour resides in the implementation of Menu and Next.

```
list <SimStep> stepList;

record SimStep
        event: Event;
        proc: Process;
end; step
```

```
BHPC_Simulate()
1  Initialise()
2  stepList ← Menu(currState.proc)
3  while currState.time < simDuration ∧ stepList ≠ ∅
4  do step ← Select(stepList)
5      currState ← Next(step)
6      stepList ← Menu(currState.proc)
```

4.4.3 Initialisation procedure

Description During initialisation, the first engine state is set. The initialisation procedure sets the engine state time to 0, sets the initial engine state process and sets all qualifiers to zero. After initialisation the current engine state is the initial engine state.

```
INITIALISE()
1  $curr\_state.time \leftarrow 0$
2  $curr\_state.proc \leftarrow$ GETINITIALEXPRESSION()
3  $curr\_state.space \leftarrow$ SETINITIALQUALIFIERS()
```

GETINITIALEXPRESSION gets the initial process expression by

1. Getting the *name* of the initial process from the model.
2. Finding the process definition for *name* in the model.
3. Getting the process expression from the process definition.

4.4.4 Menu procedure

Description The Menu procedure of the simulation engine is a procedure that invokes the Event Handler (Event Routines Package) to determine the set of simulation steps. The Event Handler contains the implementation of MENU and MENU$_{NF}$. For a detailed description of the Event Handler, we refer to Section 4.5.

```
MENU()
1  return EventHandler.GETMENU(currState.proc)
```

4.4.5 Selection procedure

Description The SELECT procedure allows selection of a simulation step from the set of simulation steps (by MENU). Depending on the simulation mode of the engine, selection will be done manual, automatic, or from batch.

For manual mode an interaction mechanism with the user is needed. In automatic mode the engine selects the transition. The selection by the engine can be random, or based on heuristics. In batch mode the engine should be able to interpret the selection from a batch file. After selection of the menuItem NEXT will execute the simulation step described by the menuItem. In the prototype only manual selection is implemented.

4.4.6 Next procedure

Description The NEXT procedure executes a simulation step and updates the current engine state. For discrete transitions, only the process expression has to be updated. The process expression for the new engine state is stored in the simulation step and does not have to be recalculated.

```
NEXT($item$ : $MenuItem$)
1  $currState.proc \leftarrow item.proc$
```

4.5 Event routines

Calculation of the set of simulation steps is performed by MENU. MENU is implemented in the EventHandler class of the Event Routines package. This class contains public procedure GETMENU to start calculation of the set of simulation steps. The EventHandler also contains procedures MENU_{NF} and MENU to calculate the set of action prefix processes and the set of simulation steps respectively. The implementation of these procedures follows the definitions of Chapter 3. After calculation GETMENU returns the set of simulation steps. Appendix D contains the software architecture of the discrete engine. For a detailed description of the software we refer to the tool documentation [5].

4.5.1 Menu

MENU is a private procedure of the EventHandler class of the Event Routine Package that converts a list of action prefix processes into a list of simulation steps. This conversion can be done by transforming each action prefix ($a.B$) from the list into a simulation step (a, B).

```
list <ActionPrefix> prefixList;
list <MenuItem>     menuItemList;
```

```
MENU($prefixList$)
1  for $i = 1$ to $prefixList.size()$
2  do $(a.B) \leftarrow prefixList[i]$)
3      INSERT($(a, B), menuItemList$)
4  return $menuItemList$
```

4.5.2 MenuNF

MENU_{NF} is a recursive definition that decomposes BHPC expressions up to the level where actions are defined in the SOS rules. In Table 3.2 we gave a definition of MENU_{NF}. In the implementation we use procedure MENUNF to determine the type of a BHPC expression and to invoke procedures that decompose BHPC expressions of this type.

[5] `http://fmt.cs.utwente.nl/tools/bhave/`

For example Menu$_{NF}$(a.B) selects a procedure to determine the normal form for an action prefix, which is NFaprefix in this case.

```
list <ActionPrefix> prefixList;
```

```
MenuNF($p$ : Process)
 1  switch
 2    case Type($p$) = stop :
 3          prefixList = NFstop($p$)
 4    case Type($p$) = action prefix :
 5          prefixList = NFaprefix($p$)
 6    case Type($p$) = trajectory prefix :
 7          prefixList = NFtprefix($p$)
 8    case Type($p$) = guard :
 9          prefixList = NFguard($p$)
10    case Type($p$) = choice :
11          prefixList = NFchoice($p$)
12    case Type($p$) = parallel composition :
13          prefixList = NFparcomp($p$)
14    case Type($p$) = hiding :
15          prefixList = NFhiding($p$)
16    case Type($p$) = renaming :
17          prefixList = NFrenaming($p$)
18    case Type($p$) = recursion :
19          prefixList = NFrecursion($p$)
20  return prefixList
```

The decomposition procedures invoked by MenuNF (i.e. NFstop, NFaprefix, . . .) are implementations of recursive procedures from Table 3.2 and hence follow the SOS rules of the language.

Recall from Section 3.5.1 that we use an expansion law to resolve parallelism and that application of this law requires normal form for the processes. The result of Menu$_{NF}$ is in normal form, i.e. it is a set of action prefix processes. For example the set $\{(\mathsf{a}.A), (\mathsf{b}.B), (\mathsf{c}.C)\}$ represents the following process in normal form $(\mathsf{a}.A + \mathsf{b}.B + \mathsf{c}.C)$.

In the implementation we use lists instead of sets. After calculation of Menu$_{NF}$, Menu converts the list of action prefixes into a list of simulation steps. The remainder of this chapter is dedicated to description of the decomposition procedures from Menu$_{NF}$. For each procedure we repeat the Menu$_{NF}$ definition from Table 3.2 and present a pseudo-algorithm for the procedure.

4.5.3 NFstop

$$\text{Menu}_{NF}(\mathbf{0}) \quad = \quad \emptyset$$

When the event handler detects a stop process, it will return an empty list of action prefix processes.

```
NFstop(0)
1  return []
```

4.5.4 NFaprefix

$$\text{Menu}_{NF}(\mathsf{a}.B) \quad = \quad \{(\mathsf{a}.B)\}$$

For action prefix the procedure inserts the action prefix into an empty list and returns the list.

```
NFaprefix(a.B)
1  return [(a.B)]
```

4.5.5 NFtprefix

$$\text{Menu}_{NF}([\varphi]\,.B) \quad = \quad \{(@.B)\}$$

A trajectory prefix is converted into an action prefix by replacing the prefix by the abstraction representative. This abstraction prefix is inserted into an empty list and returned.

```
NFaprefix([$f$ | $\Phi$] .B)
1  return [(@.B)]
```

4.5.6 NFguard

$$\text{Menu}_{NF}(\langle Pred\rangle.B) \quad = \quad \begin{cases} \text{Menu}_{NF}(B) & \text{, if } Pred \text{ is satisfied} \\ \emptyset & \text{, otherwise} \end{cases}$$

For evaluation of the guard predicate we need a solver. When the guard is satisfied, Menu_{NF} for B must be determined, otherwise the procedure returns an empty list.

```
NFguard(⟨Pred⟩.B)
1  if Pred
2     then return MenuNF(B)
3     else  return []
```

For our prototype we do not use solvers to evaluate guard expressions. We just assume that the guard predicate is always satisfied. Therefore we ignore the predicate and directly call MenuNF with the consecutive behaviour as can be seen in the pseudo-algorithm for our prototype.

```
NFguard(⟨Pred⟩.B)
1  return MenuNF(B)
```

We exemplify decomposition of guard processes for our prototype in Example 4.5.1.

Example 4.5.1 (Guard handling).

$$\begin{aligned}
\text{NFguard}(\langle \mathit{False}\rangle\,.\text{a.B}) &= \text{MenuNF(a.B)}\\
&= \text{NFaprefix(a.B)}\\
&= [(\text{a.B})]\\
\text{NFguard}(\langle \mathit{True}\rangle\,.\text{a.B}) &= \text{MenuNF(a.B)}\\
&= \text{NFaprefix(a.B)}\\
&= [(\text{a.B})]\\
\text{NFguard}(\langle \mathit{Pred}\rangle\,.\,[f \mid \Phi]\,.\text{B}) &= \text{MenuNF}([f \mid \Phi]\,.\text{B})\\
&= \text{NFtprefix}([f \mid \Phi]\,.\text{B})\\
&= [(@.\text{B})]
\end{aligned}$$

□

Clearly, MenuNF is always invoked, disregarding satisfaction of the guard predicate. The examples moreover depicts the decomposition mechanism which alternates between MenuNF and decomposition procedures. MenuNF determines the process expression type and selects the appropriate decomposition procedure. The decomposition procedure decomposes process expressions according to the SOS rules and collects the list of action prefix processes from its components, which is determined by MenuNF.

4.5.7 NFchoice

$$\text{MENU}_{NF}(\sum_{v \in I} B(v)) \quad = \quad \bigcup_{v \in I} \text{MENU}_{NF}(B(v))$$

NFCHOICE determines the set of action prefix processes by joining sets of action prefix processes for all choice options.

```
list <ActionPrefix> prefixList;
list <Process> choiceList;
```

```
NFCHOICE($\sum_{v \in I} B(v)$)
1  $choiceList \leftarrow$ GETCHOICES($\sum_{v \in I} B(v)$)
2  for $i \leftarrow 1$ to $choiceList.size()$
3  do MERGE(MENUNF($choiceList[i]$), $prefixList$)
4  return $prefixList$
```

This procedure first collects all options in the choiceList. Then we use MENUNF to determine the list of action prefix processes for each option and merge this list with prefixList to collect all action prefix processes. In Example 4.5.2 we exemplify the application of NFCHOICE.

Example 4.5.2 (Choice handling). NFCHOICE$\Big(\mathsf{a}.A + ([f \mid \Phi]\,.B + \mathbf{0})\Big)$

$$\begin{aligned}
&= \text{MERGE}\Big(\text{MENUNF}(\mathsf{a}.B), \text{MENUNF}([f \mid \Phi]\,.B + \mathbf{0})\Big)\\
&= \text{MERGE}\Big(\text{NFAPREFIX}(\mathsf{a}.A), \text{NFCHOICE}(([f \mid \Phi]\,.B + \mathbf{0})\Big)\\
&= \text{MERGE}\Big([(\mathsf{a}.A)], \text{MERGE}\Big(\text{MENUNF}([f \mid \Phi]\,.B), \text{MENUNF}(\mathbf{0})\Big)\Big)\\
&= \text{MERGE}\Big([(\mathsf{a}.A)], \text{MERGE}\Big(\text{NFTPREFIX}([f \mid \Phi]\,.B), \text{NFSTOP}(\mathbf{0})\Big)\Big)\\
&= \text{MERGE}\Big([(\mathsf{a}.A)], \text{MERGE}\Big([(@.B)], []\Big)\Big)\\
&= \text{MERGE}\Big([(\mathsf{a}.A)], [(@.B)]\Big)\\
&= [(\mathsf{a}.A), (@.B)]
\end{aligned}$$

□

First the choice process is decomposed into two processes. The choice process joins the simulation steps resulting from its components. Decomposition continues until the level of stop, action prefix and trajectory prefix.

4.5.8 NFrenaming

$$\text{Menu}_{NF}(B\,[\sigma]) \quad = \quad [\sigma]\,(\text{Menu}_{NF}(B))$$

NFrenaming first determines the set of action prefix processes for B and then applies renaming to this set. Applying renaming on an action prefix process renames the action, if possible, and attaches the renaming to the consecutive behaviour.

```
list <ActionPrefix> originalList;
list <ActionPrefix> prefixList;
ActionPrefix      renamedPrefix;
```

```
NFrenaming($B\,[\sigma]$)
1 $originalList \leftarrow$ MenuNF($B$)
2 for $i \leftarrow 1$ to $originalList.size()$
3 do $(\mathsf{a}.B) \leftarrow originalList[i]$
4     $renamedPrefix \leftarrow$ ApplyRenaming$\big([\sigma], (\mathsf{a}.B)\big)$
5     Insert($renamedPrefix, prefixList$)
6 return $prefixList$
```

```
ApplyRenaming($[\sigma], \mathsf{a}.B$)
1 return $\sigma(\mathsf{a}).B\,[\sigma]$
```

This procedure first determines the list of action prefix processes. Then ApplyRenaming is used to rename the action according to the renaming function and the renaming function is attached to the consecutive behaviour. All (possibly renamed) action prefixes are collected in the prefixList and finally returned. In Example 4.5.3 we exemplify the application of NFrenaming.

Example 4.5.3 (Trace for guard process). In four iterations we give a trace for the following renaming process $\text{Menu}_{NF}\big((\mathsf{a.b.(c.0)}\,[\sigma_2])\,[\sigma_1]\big)$ with $\sigma_1 = \{x/a, y/b, z/c\}$ and $\sigma_2 = \{a/c\}$

Iteration 1 $\text{Menu}_{NF}\big((\mathsf{a.b.(c.0)}\,[\sigma_2])\,[\sigma_1]\big)$

$= \text{NFrenaming}\big((\mathsf{a.b.(c.0)}\,[\sigma_2])\,[\sigma_1]\big)$

$= \text{ApplyRenaming}\big([\sigma_1], \text{MenuNF}(\mathsf{a.b.(c.0)}\,[\sigma_2])\big)$

$= \text{ApplyRenaming}\big([\sigma_1], \text{NFaprefix}(\mathsf{a.b.(c.0)}\,[\sigma_2])\big)$

$= \text{ApplyRenaming}\big([\sigma_1], [(\mathsf{a.b.(c.0)}\,[\sigma_2])]\big)$

$= [\big((\mathsf{x.b.(c.0)}\,[\sigma_2])\,[\sigma_1]\big)]$

MENU converts $[\big((\mathsf{x}.\mathsf{b}.(\mathsf{c}.\mathbf{0})\,[\sigma_2])\,[\sigma_1]\big)]$ into $[\big((\mathsf{x}, \mathsf{b}.(\mathsf{c}.\mathbf{0})\,[\sigma_2])\,[\sigma_1]\big)]$. From this list only action x can be selected.

Iteration 2 (After selection and execution of x)

$\text{MENU}_{NF}\Big(\big(b.(c.\mathbf{0})\,[\sigma_2]\big)\,[\sigma_1]\Big)$

$= \text{NFRENAMING}\Big(\big(\mathsf{b}.(\mathsf{c}.\mathbf{0})\,[\sigma_2]\big)\,[\sigma_1]\Big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \text{MENUNF}\big(\mathsf{b}.(\mathsf{c}.\mathbf{0})\,[\sigma_2]\,\big)\Big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \text{NFAPREFIX}\big(\mathsf{b}.(\mathsf{c}.\mathbf{0})\,[\sigma_2]\,\big)\Big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \big[\big(\mathsf{b}.(\mathsf{c}.\mathbf{0})\,[\sigma_2]\,\big)\big]\Big)$

$= \Big[\Big(\mathsf{y}.\big((\mathsf{c}.\mathbf{0})\,[\sigma_2]\,\big)\,[\sigma_1]\,\Big)\Big]$

Iteration 3 (After selection and execution of y)

$\text{MENU}_{NF}\Big(\big((\mathsf{c}.\mathbf{0})\,[\sigma_2]\,\big)\,[\sigma_1]\,\Big)$

$= \text{NFRENAMING}\Big(\big((\mathsf{c}.\mathbf{0})\,[\sigma_2]\,\big)\,[\sigma_1]\,\Big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \text{MENUNF}\big((\mathsf{c}.\mathbf{0})\,[\sigma_2]\,\big)$

$= \text{APPLYRENAMING}\left(\,[\sigma_1]\,, \text{NFRENAMING}\big((\mathsf{c}.\mathbf{0})\,[\sigma_2]\,\big)\right)$

$= \text{APPLYRENAMING}\left(\,[\sigma_1]\,, \text{APPLYRENAMING}\big(\,[\sigma_2]\,, \text{MENUNF}(\mathsf{c}.\mathbf{0})\big)\right)$

$= \text{APPLYRENAMING}\left(\,[\sigma_1]\,, \text{APPLYRENAMING}\big(\,[\sigma_2]\,, \text{NFAPREFIX}(\mathsf{c}.\mathbf{0})\big)\right)$

$= \text{APPLYRENAMING}\left(\,[\sigma_1]\,, \text{APPLYRENAMING}\big(\,[\sigma_2]\,, [\mathsf{c}.\mathbf{0}]\big)\right)$

$= \text{APPLYRENAMING}\left(\,[\sigma_1]\,, \big[(\mathsf{a}.\mathbf{0})\,[\sigma_2]\,\big]\right)$

$= \Big[\big((\mathsf{x}.\mathbf{0})\,[\sigma_2]\,\big)\,[\sigma_1]\,\Big]$

Iteration 4 (After selection and execution of x)

$\text{MENU}_{NF}\Big(\big(\mathbf{0}\,[\sigma_2]\,\big)\,[\sigma_1]\,\Big)$

$= \text{NFRENAMING}\Big(\big(\mathbf{0}\,[\sigma_2]\,\big)\,[\sigma_1]\,\Big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \text{MENUNF}\big(\mathbf{0}\,[\sigma_2]\,\big)\Big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \text{NFRENAMING}\big(\mathbf{0}\,[\sigma_2]\,\big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \text{APPLYRENAMING}\big(\,[\sigma_2]\,, \text{MENUNF}\big(\mathbf{0}\big)\big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \text{APPLYRENAMING}\big(\,[\sigma_2]\,, \text{NFSTOP}\big(\mathbf{0}\big)\big)$

$= \text{APPLYRENAMING}\Big(\,[\sigma_1]\,, \text{APPLYRENAMING}\big(\,[\sigma_2]\,, []\big)$

$= \textsc{ApplyRenaming}\big([\sigma_1], []\big)$
$= []$

There are no new events to simulate. $\square$

4.5.9 NFhiding

$$\textsc{Menu}(\mathsf{new}\ w.B) \quad = \quad [\tau/e_1, \ldots, \tau/e_n]\,(\textsc{Menu}_{NF}(B))$$

Hiding is very similar to renaming; hiding can be seen as a special case of renaming where everything is renamed to τ. The hiding set can easily be converted into a hiding function by mapping all set elements to τ.

```
list <ActionPrefix> prefixList;
```

```
NFHIDING(new $w.B$)
1  return MENUNF($B\,[\tau/e_1, \ldots, \tau/e_n]$)
```

4.5.10 NFrecursion

$$\textsc{Menu}_{NF}(P) \quad = \quad \textsc{Menu}_{NF}(B) \qquad \textbf{where } P \triangleq B$$

NFRECURSION tries to instantiate the recursive process by searching the process definition for the recursive process. GETEXPRESSION searches the BHPC model to find the definition for P and returns the behaviour expression of this definition. Then we apply MENUNF on this behaviour expression to determine the list of action prefixes.

```
BHPCexpr expr;
```

```
NFRECURSION($P$)
1  $expr \leftarrow$ GETEXPRESSION($P, BHPCmodel$)
2  return MENUNF($expr$)
```

Unguarded processes This approach works well when all recursions are guarded recursions. A recursion is unguarded when the process identifier is not preceded by an action, i.e. $B \triangleq \mathsf{a}.B + B$ contains one unguarded recursion. Without measures to deal with this situation the engine can get stuck in resolving recursion. In [11] two practical approaches are suggested: disallowing unguarded processes and limiting the number of recursion steps the engine may take without intervention.

For our implementation we allow unguarded recursion, since it gives more freedom in defining BHPC processes. When recursion is not resolved after a predefined number of recursion steps, the user is asked for feedback on how many addition steps should be taken to resolve recursion.

4.5.11 NFparcomp

$$\begin{aligned}
\textsc{Menu}_{NF}(B \parallel_A^H C) &= \textsc{Menu}_{NF}\Big(\sum_{\substack{i\in I\\ \mathsf{b}_i \notin A\cup\{@\}}} \mathsf{b}_i.\big(B_i \parallel_A^H C\big)\Big)\\
&\cup\ \textsc{Menu}_{NF}\Big(\sum_{\substack{k\in K\\ \mathsf{c}_k \notin A\cup\{@\}}} \mathsf{c}_k.\big(B \parallel_A^H C_k\big)\Big)\\
&\cup\ \textsc{Menu}_{NF}\Big(\sum_{\substack{\mathsf{a}=\mathsf{b}_i=\mathsf{c}_k\\ \mathsf{a}\in A\cup\{@\}}} \mathsf{a}.\big(B_i \parallel_A^H C_k\big)\Big)\\
&\quad \textbf{where}\quad B=\sum_{i\in I}\mathsf{b}_i.B_i \quad \text{and} \quad C=\sum_{k\in K}\mathsf{c}_k.C_k
\end{aligned}$$

To determine the menu for the parallel composition, we apply the expansion law from Chapter 2.4. The expansion can be applied on the parallel composition when its components (processes) are in normal form, which is the case since we apply MENUNF on the components of the composition.

```
vector<ActionPrefix>  leftV;
vector<ActionPrefix>  rightV;
list<ActionPrefix>    prefixList;
```

```
NFPARCOMP($B \parallel_A^H C$)
 1  $leftV \leftarrow$ MENUNF($B_1$)
 2  $rightV \leftarrow$ MENUNF($B_2$)
 3  for ($\mathsf{b}_i.B_i$) $\in leftV$
 4  do if $\mathsf{b}_i \notin A \cup \{@\}$
 5        then INSERT($\mathsf{b}_i.(B_i \parallel_A^H C), prefixList$);
 6        else $\mathsf{b}_i \in A \cup \{@\}$
 7             for ($\mathsf{c}_j.C_j$) $\in rightV$
 8             do if $\mathsf{c}_j \in A \wedge \mathsf{b}_i = \mathsf{c}_j$
 9                   then INSERT($\mathsf{b}_i.(B_i \parallel_A^H C_j), prefixList$);
10
11  for ($\mathsf{c}_j.C_j$) $\in rightV$
12  do if $\mathsf{c}_j \notin A \cup \{@\}$
13        then INSERT($\mathsf{c}_j.(B \parallel_A^H C_j), prefixList$);
14  return $prefixList$
```

First NFPARCOMP determines the normal form for the left and right parallel composition components. Then all events from the left component and all synchronising events are determined in the first loop. All action prefixes from the left normal form are evaluated; if the action is not a synchronising action, then we add $\mathsf{b_i}.(B_i \parallel^H_A C)$ to the prefixList. For synchronising events a synchronisation partner is searched in the right normal form.

Then all action prefixes from the right component are evaluated in the second loop; if the action is not a synchronising action, then we add $\mathsf{c_j}.(B \parallel^H_A C_j)$ to the prefixList. Finally, NFparcomp returns the list of all action prefixed.

4.6 Conclusions

The implementation of the discrete simulation functionality is based on the definitions of Chapter 3. We decomposed the engine definition (simulation algorithm) into components that map naturally onto the architecture of the engine. Moreover we have made a distinction between BHPC related procedures, e.g. Menu, and other engine procedures, like updating the engine state, or selection of simulation steps. We have implemented all BHPC related procedures in a separate package. As a result, extensions of BHPC only apply to some parts of this package, which is convenient for implementing extensions and changes BHPC.

Most notably, our implementation of the discrete engine is a direct implementation of the discrete engine definition of Chapter 3. Therefore the implementation is based on the operational semantics of BHPC, which gives it a strong mathematical base. During development of the prototype we started with the framework for the tool and gradually added BHPC processes. In essence we have experienced the suitability of the prototype to serve as “sand-box” for extensions and changes to BHPC. From our experience adding new BHPC procedures can be done in a very structural manner and therefore we are confident in the suitability of the prototype as experimental environment for further BHPC developments.

5 Simulation Example

In this chapter we provide a simulation trace performed with our discrete engine. First we give a BHPC model in Section 5.1. For completeness present both the mathematical and the ASCII representation. The simulation trace of the engine is given in Section 5.2. The trace consists of a sequence of actions that have been executed and consecutive engine states. Engine states are given by BHPC expressions in internal format, see Appendix C.

5.1 Model

The model has been chosen to include a variety of BHPC expressions, while keeping the internal representation as short as possible for readability.

BHPC representation

$$\begin{aligned}
\text{aprefixP} &\triangleq \mathsf{a}.\text{guardP}\\
\text{guardP} &\triangleq \langle \mathcal{P}red \rangle .\text{tprefixP}\\
\text{tprefixP} &\triangleq [f \mid \Phi_1] .\text{choiceP}\\
\text{choiceP} &\triangleq \mathbf{0} + \text{parcompP}\\
\text{parcompP} &\triangleq \text{leftProc1} \parallel_A^H \text{rightProc1}\\
\text{leftProc1} &\triangleq \mathsf{b}.\text{leftProc2}\\
\text{leftProc2} &\triangleq [f \mid \Phi_2] .\text{leftProc2}\\
\text{rightProc1} &\triangleq [f \mid \Phi_3] .\text{rightProc1}
\end{aligned}$$

ASCII model

```
actions: a, b
initial aprefixP

proc aprefixP ^= a . guardP
proc guardP   ^= <pred> . tprefixP
proc tprefixP ^= [ f | PHI_1 ] . choiceP
proc choiceP  ^= 0 + parcompP
proc parcompP ^= leftP1 |{}{}| rightP1
proc leftP1   ^= b . leftP2
proc leftP2   ^= [ f | PHI_2 ] . leftP2
proc rightP1  ^= [ f | PHI_3 ] . rightP1
```

5.2 Trace

The following trace is produced by our engine.

```
ENGINE STATE PROCESS

       <behaviourexpression>
           <process>
               <actionprefix>
                   <name>a</name>
                   <process>
                       <recursion>
                           <name>guardP</name>
                       </recursion>
                   </process>
               </actionprefix>
           </process>
       </behaviourexpression>

EVENT: a (action)

ENGINE STATE PROCESS

       <behaviourexpression>
           <process>
               <recursion>
                   <name>guardP</name>
               </recursion>
           </process>
       </behaviourexpression>

EVENT: @ (action)

ENGINE STATE PROCESS
```

```
       <behaviourexpression>
           <process>
               <recursion>
                   <name>choiceP</name>
               </recursion>
           </process>
       </behaviourexpression>

EVENT: b (action)

ENGINE STATE PROCESS

       <behaviourexpression>
           <process>
               <parallelcomposition>
                   <process>
                       <recursion>
                           <name>leftProc2</name>
                       </recursion>
                   </process>
                   <process>
                       <recursion>
                           <name>rightProc1</name>
                       </recursion>
                   </process>
                   <set />
               </parallelcomposition>
           </process>
       </behaviourexpression>

EVENT: @ (action)

ENGINE STATE PROCESS

       <behaviourexpression>
           <process>
               <parallelcomposition>
                   <process>
                       <recursion>
                           <name>leftProc2</name>
                       </recursion>
                   </process>
                   <process>
                       <recursion>
                           <name>rightProc1</name>
                       </recursion>
                   </process>
                   <set />
               </parallelcomposition>
           </process>
       </behaviourexpression>

EVENT: @ (action)

ENGINE STATE PROCESS

       <behaviourexpression>
           <process>
               <parallelcomposition>
                   <process>
                       <recursion>
                           <name>leftProc2</name>
                       </recursion>
                   </process>
                   <process>
                       <recursion>
                           <name>rightProc1</name>
                       </recursion>
                   </process>
                   <set />
               </parallelcomposition>
           </process>
       </behaviourexpression>

               ⋮
```

The trace for the extended bouncing ball example can be found in Appendix E.

5.3 Conclusions

The output format for the simulation results is an alternating sequence of events and engine states. These traces of the discrete engine contain all information necessary for the analysis of hybrid system control structures. For a more convenient analysis of the results, we suggest a connection to a visualisation unit and depict actions in MSC. For future hybrid analysis we propose a new visualisation technique in Chapter 6.

6 Towards Hybrid Simulation

For the analysis of hybrid systems it is not sufficient to just analyse the discrete and the continuous-time behaviour separately. The evolution of system variables can be influenced by discrete events, and vice versa. Therefore analysis of hybrid systems must also cover the relationship between discrete and the continuous-time behaviour.

The hybrid simulation technique for hybrid simulation of BHPC, that is used in [11], is a modified version of the event tracking algorithm from [31]. After initialisation, the continuous-time behaviour is simulated until an event is detected. Subsequently, the event time is determined and corresponding signal space values are computed. Event-time and signal space values are used to set the new engine state. Hereupon the simulation loop repeats these actions for the new engine state.

In this chapter we present some ideas for extending our discrete engine definitions and discrete prototype implementation to a hybrid engine. Moreover we define extensions for some of the definitions of Chapter 3 in Section 6.1. Extending the presented theory for discrete simulation to hybrid simulation essentially means adding support for continuous-time behaviour. This extension is not trivial; for a detailed description of the complexity of supporting continuous-time behaviour we refer to [11]. In Section 6.2 we describe necessary additions for the discrete engine design to add support for continuous-time behaviour. Our most important contribution to hybrid system analysis is the proposal of a new visualisation technique that presents both discrete and continuous-time behaviour and their relation. We describe this visualisation technique in Section 6.3.

6.1 Hybrid engine

6.1.1 Engine State Transitions

All engine state definitions from Chapter 3.4 remain unchanged. For convenience we repeat the general engine definition.

Definition 6.1.1. (Engine state).
Let the tuple HTS=$\langle S, \mathcal{A}, \rightarrow, \mathbb{W}, \Phi, \rightarrow_c \rangle$ be a hybrid transition system. Then we define a simulation state $e \in E$ as

$$e = \langle \theta, p, \varphi(\theta) \rangle$$

Where $\theta \in \mathbb{R}$ is the time stamp of e, $p \in S$ is a BHPC process expression of e and $\varphi(\theta)$ is the signal snapshot for signal space $\mathbb{W}$ of s at time θ. □

We extend our transition definition with continuous transitions and enable the hybrid engine to execute both discrete and continuous transitions. Therefore we generalise actions and trajectories to events and use events in stead of actions in the definitions for simulation step, Menu and Next.

Definition 6.1.2 (Event). Let $HTS = \langle S, \mathcal{A}, \rightarrow, \mathbb{W}, \Phi, \rightarrow_c \rangle$ be the transition system. Then we define a set of events $\mathcal{E}$, where $\mathcal{E} = \mathcal{A} \cup \Phi$. □

Definition 6.1.3 (Hybrid simulation step). Let HTS = $\langle S, \mathcal{A}, \rightarrow, \mathbb{W}, \Phi, \rightarrow_c \rangle$ be a hybrid transition system, $p, p' \in S$ be BHPC process expressions, $\mathsf{e} \in \mathcal{E}$ an action and $p \xrightarrow{\mathsf{e}} p' \in \rightarrow$ or $p \xrightarrow{\mathsf{e}} p' \in \rightarrow_c$ be a transition of the hybrid transition system. Then $(\mathsf{e}, p') \in T$ is a simulation step of the engine, where T represents the set of transition steps. □

Definition 6.1.4 (Menu for BHPC). Let N be the set of BHPC process expression in normal form and let T be the set of transition steps and (e .B) an action or trajectory prefix, with $\mathsf{e} \in \mathcal{E}$. Then we define Menu: $N \rightarrow T$,

$$\text{Menu}(N) = \{(\mathsf{e}, B) \mid (\mathsf{e}.B) \in N\}$$

□

Abstraction is removed from the recursive function Menu$_{NF}$ for trajectory prefix.

$$\text{Menu}_{NF}([\varphi]\,.B) = \{([\varphi]\,.B)\}$$

Obviously a connection to a ODE/DAE solver is needed for simulation of a trajectory prefix, which is done after selection of a simulation step. Parallel composition, renaming and hiding are more complex for hybrid simulation and a solution is not trivial. We refer to [11] for a detailed description of hybrid simulation for these expressions.

During continuous transitions a trajectory is executed. After a continuous transition, the time stamp for next engine state is increased by the duration of the trajectory and also the time for snapshot of the signal space is increased by this duration.

Definition 6.1.5 (Next). Let (e, p') be a simulation step from the set of possible transitions T, calculated by Menu, and S be the set of simulation states and let $\langle \theta, p, \varphi(\theta)\rangle$ be the current simulation state of the engine. If $p \xrightarrow{\mathsf{e}} p'$ is a transition in our HTS, then we define the function $\textsc{Next} : \delta \times S \to S$, such that

$$\textsc{Next}\big((\mathsf{e}, p'), \langle \theta, p, \varphi(\theta)\rangle\big) = \begin{cases} \langle \theta, p', \varphi(\theta)\rangle & \text{if } \mathsf{e} \in \mathcal{A} \\ \langle \theta + \mathsf{t}(\mathsf{e}), p', \varphi(\theta + \mathsf{t}(\mathsf{e}))\rangle & \text{if } \mathsf{e} \in \Phi \end{cases}$$

Where $\mathsf{t}(\mathsf{e})$ denotes the duration of event e. □

6.1.2 Hybrid engine

The simulation algorithm implements the extended versions of Menu and Next, further no changes.

```
BHPC_Simulate()
1  $e \leftarrow \langle 0, p_0, \varphi(0)\rangle$
2  $T \leftarrow \textsc{Menu}(\pi^p(e))$
3  while $T \neq \emptyset$
4  do $(\mathsf{e}, p') \leftarrow \textsc{Select}(T)$
5      $e' \leftarrow \textsc{Next}((\mathsf{e}, p'), e)$
6      $T \leftarrow \textsc{Menu}(\textsc{Menu}_{NF}(\pi^p(e'))$
```

Where

- $e, e' \in E$ represent engine states
- $\pi^p(e)$ is the process expression of the engine state (by projection).
- T is the set of transitions
- Menu, Menu$_{NF}$ and Next are functions that determine the set of simulation steps, the set of action prefixes and the next engine state respectively
- Select is a way to select a simulation step (manually, automatically, or from batch) from the set of simulation steps T

6.2 Design

In Chapter 4 we presented the discrete engine design for BHPC. Extension to a hybrid engine essentially means adding support for continuous-time behaviour.

In BHPC trajectory prefixes represent continuous-time behaviour and usually refer to trajectory set definitions in which continuous behaviour is described by differential equations. Solvers will be used to evaluate differential equations and an interface with these solvers must be designed and implemented.

The hybrid engine also must support equations and expressions in other BHPC processes, such as predicates in guards, (exit) conditions in trajectory prefixes, equations in trajectory set definitions and parameters for actions and processes. Restrictions on allowed equations and expressions for all these processes must be formulated and solvers are needed for evaluation. Furthermore additional data types are needed to support continuous-time behaviour. Here we present an overview of necessary additions for hybrid simulation.

Support for trajectory qualifiers Data structures to store qualifiers values for each process should be designed. The structures that handle processes, should be extended to handle qualifiers related to the process.

Parametrisation of processes Values are passed to processes via parameters. Consequently, data structures to pass parameters to processes should be designed and integrated.

Trajectory-prefixes In the definition of BHPC trajectory-prefixes are defined in an abstract fashion. However, a more practical way to represent them should be chosen. Usually, differential equations (ODE or DAE[1]) are used to represent continuous behaviour. Therefore, data structures to store and handle such equations should be designed. The type of supported equations depends on ODE/DAE solver. Some solvers support only ordinary differential equations, but differential algebraic equations solvers are getting common, too. Consequently, it means, that an interface to solver(s) should be designed and implemented.

Guards In BHPC we do not put any restrictions on predicates. However, in practical application a restrictions to allowed predicates should be chosen and a data structure to represent them designed and implemented. An interface to some solver may be necessary to solve predicates.

Parallel composition The complete expansion law should be implemented. Moreover, if ODE/DAE are used to represent continuous behaviour and an oracle is not available, the expansion law should be modified to allow to make choice amongst trajectory-prefixes on-the-fly, i.e., during simulation.

Hiding and renaming Hiding and renaming of trajectory-qualifiers is not a trivial problem, because it may influence options for choice. Hence, it is better to investigate behaviour of restricted hiding and renaming, evaluate pros and cons of flexibility and error safety and find a nice balance.

Recursion Parameters and qualifiers values management should be added to recursive calls subroutines.

Extending discrete BHPC simulator to simulate hybrid systems is not a trivial problem. However, it provides a very good base for future development of BHPC simulation tools. Moreover, some of the problems related with hybrid extensions are investigated in Bhave prototype tool. See `http://fmt.cs.utwente.nl/tools/bhave` for more information about Bhave tool-set.

6.3 Visualisation

Simulation results usually visualise the evolution of the system in time. Results for hybrid systems are in essence represented by a mapping of state values and events to the time-line.

[1]An *ordinary differential equation* is a relation that contains functions of only one independent variable, and one or more of its derivatives with respect to that variable. It's general form is $F(x, y, \dot{y}, \ddot{y}, \ldots, y^{(n)}) = 0$. A *differential-algebraic equation* is a relation in which the derivatives are not (in general) expressed explicitly, and typically derivatives of some of the dependent variables may not appear in the equations at all. General form of such equations is $F(\dot{x}, x, y, t) = 0$, where $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$ are differential and algebraic variables, respectively.

Event traces or message sequence charts (MSC) [27, 17] adequately represent discrete system behaviour. Graphs are adequate for the ordinary continuous system. In hybrid systems we have both the evolution of system variables and events. Moreover the evolution of system variables may trigger events and events may trigger change of evolution. Hence a combined view is crucial to fully analyse hybrid system behaviour. Several approaches are proposed in [28, 21], [15, p. 74], but none of them provides a combined view.

We propose to use message sequence plots (MSP). MSP is a combination of MSC and plots that captures both discrete and continuous-time behaviour and their interaction. We do not provide a formal definition of MSP, because we do not want to introduce a new formalism here, but rather propose an abstract technique for visualisation of hybrid behaviour. Therefore, we just describe MSP informally and illustrate its application by examples.

6.3.1 Basic MSP

MSP has two main compounds: 90° rotated message-sequence charts (MSC) and plots. MSC represents discrete behaviour and continuous-time evolution is depicted by plots over time-lines. A legend allows selection of qualifiers of interest, that are depicted in the plot. If several processes evolve concurrently, the synchronising qualifiers for both processes appear.

Time line Horizontal lines, connected to the corresponding boxes with process identifiers, represent processes and the *time-line* (*life-line* in MSC terminology). MSP represents each system process by a distinct time-line. Along each time-line, time is assumed to flow to the right at the same speed. Figure 6.1 depicts the time-line for $Process_i$.

Process_i

Figure 6.1: MSP Time-line for $Process_i$

Plots The time-line represent the flow of time and serve as the time axis for plots. The plot only shows the evolution of qualifiers of interest that are selected in the legend along the vertical axis of the plot. In BHPC trajectories are visualised by plots. The beginning and end of a trajectory are visualised by vertical dashed lines. Figure 6.2 shows the time-line for $Process_i$ along which two trajectories are visualised. Only for qualifier $qual_4$ the plot is shown.

Communication Communication in MSP is represented by communication lines, which are solid labelled vertical lines. Directed communication can be visualised by arrows on the communication lines. In BHPC there is only undirected communication by (parametrised) action prefixes. Actions are depicted by a communication line that is connected to the time-line of the originating process. To emphasise the originating process we add dots at the connections with time-lines. Figure 6.3 depicts two processes $Process_i$ and $Process_j$.

In the beginning both processes are synchronizing on their trajectories. After some time they

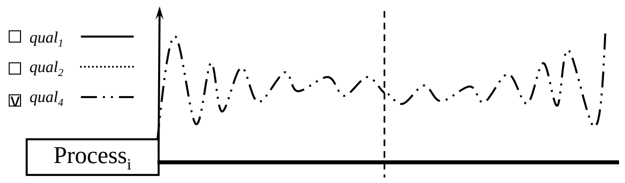


Figure 6.2: MSP plot and time-line for *Process*$_i$

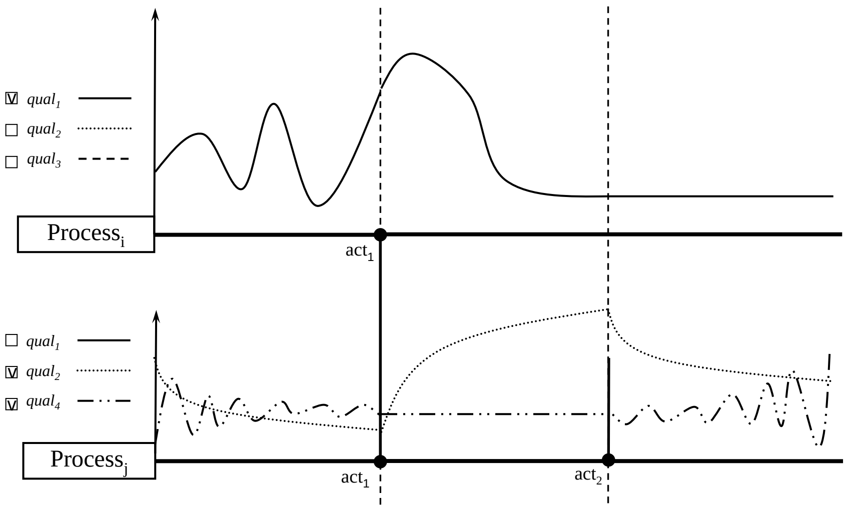


Figure 6.3: Communication in MSP

synchronise on action act$_1$ and subsequently they synchronise their trajectories. Then *Process*$_j$ executes action act$_2$ after which both processes resume trajectory synchronisation again. After the moment of execution of action act$_2$ it is unclear in this figure whether *Process*$_i$, starts a new trajectory or continues with the remainder of an unfinished trajectory. Later we shall describe extensions for MSP that do provide this information.

Remark 6.3.1 (Synchronising qualifiers). Recall that processes can only synchronise on trajectories when their common qualifiers are synchronizing qualifiers and have the same evolution. ☐

6.3.2 Event-lines

MSP supports *event-lines* that can be used to interrupt the representation of time-flow. In Example 6.3.2 we give an example of a situation in that requires event-lines.

Example 6.3.2 (Event-lines for action sequences). Consider process $P \triangleq [f \mid \Phi]\,.\mathsf{a}.\mathsf{b}.\mathsf{c}\,[f \mid \Psi]$. The representation of time-flow must be interrupted to depict event order of this the action sequence in the MSP. Otherwise all actions from the sequence would be depicted on top of each other on the time-line, since actions are instantaneous. □

We use horizontally dashed lines to represent event-lines. In Figure 6.4 shows the MSP with event-lines for Example 6.3.2.

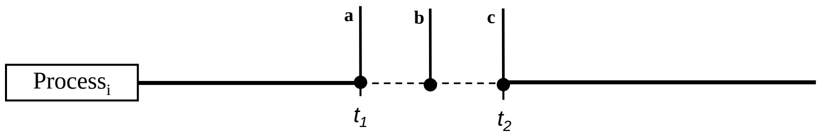


Figure 6.4: MSP action sequence

There are MSP constructs, like an action sequence, that require some space for visualisation. When the time-line is interrupted for one of the processes in the MSP, all processes must interrupt their time-lines by the same width, to stay synchronised. Consider the following example in Figure 6.5 in which a construct in one of the processes interrupts the time-line of that process.

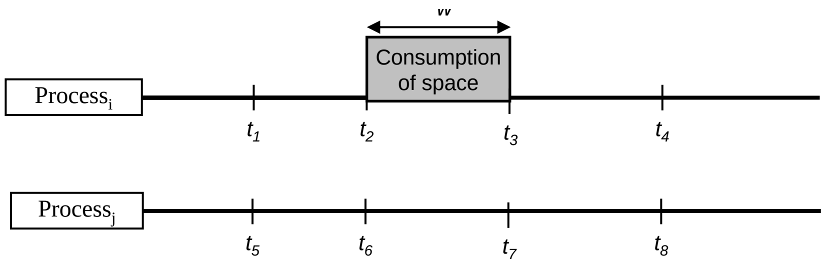


Figure 6.5: Unsynchronised time-lines

The ideal situation is that the position on the time-line denotes the same time point on each time-line. For example that $t_1 = t_5 \wedge t_2 = t_6 \wedge t_3 = t_7 \wedge t_4 = t_8$ holds in Figure 6.5. But in *Process$_i$* the time-line is interrupted by the width w of some construct and $t_2 = t_3$ while $t_6 \neq t_7$ and hence $t_7 \neq t_3$, but $t_7 = t_3 + \frac{w}{timespeed}$, where *timespeed* denotes the speed with which time is represented along the time-line.

As a solution we introduce event lines, which are horizontally dashed lines along which time does not proceed. In Figure 6.6 an event-line is used to keep all time-lines synchronised, . In this figure $t_1 = t_5 \wedge t_2 = t_6 \wedge t_3 = t_7 \wedge t_4 = t_8$ does hold.

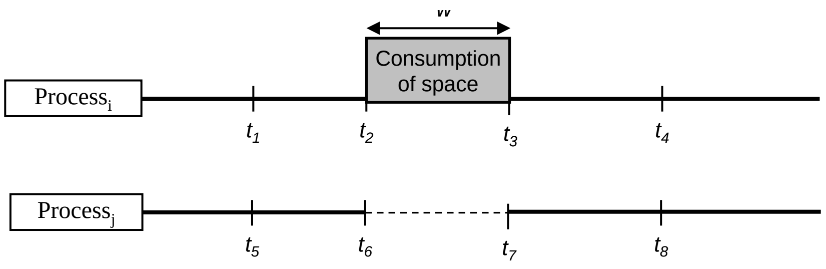


Figure 6.6: Synchronised time-lines

6.3.3 Additional MSP constructs

In this section we introduce some constructs that provide a more elaborate and convenient analysis of BHPC.

Adding BHPC expressions In MSP details of the hybrid model can be added to the simulation results to make their relation more apparent. For example we can decorate the results with trajectory prefixes at the corresponding parts of plots. In Figure 6.7 we have added the MSP trajectory expressions along the plots. The MSP shows two parallel processes: $Process_i$, executing trajectory φ and idling $Process_j$. Then $Process_i$ subsequently executes action a, action b and then synchronises with $Process_j$ by executing action c. Finally both processes synchronise their trajectory prefixes.

Remark 6.3.3 (Trajectory information). By relating trajectory prefixes to plots, we now can see whether a plot represents a new trajectory of the remainder of the previous trajectory which we could not see in basic MSP. □

Recursion Recursive calls can be depicted as boxes with a new process identifier on the time-line of the calling process. Optionally also recursion parameters can be depicted in the box. If necessary we can use event lines to compensate for the space this construct takes on the time-line and keep all time-lines synchronised. In Figure 6.8 we visualise a trace from the process $Process_i \triangleq [f \mid \Phi] .Process_i + Process_j$. The figure shows one process in the composition, clearly three instantiations for $Process_i$ can be seen before $Process_j$ is instantiated.

Renaming We could depict renaming or the result of renaming in MSP. The former gives most insight in when and how renaming takes place. Like trajectory information, renaming functions could be put along the graph. There has to be decided what the order is in case more than one renaming function is applied and therefore added to the MSP. Adding renaming details to a MSP visualises the relation between qualifiers from the plot legend and their renaming. Moreover, if we want to see the result of renaming in the plots, it would be quite difficult to keep the legend up to date and consistent for all plots shown the MSP, i.e. renaming the legend could

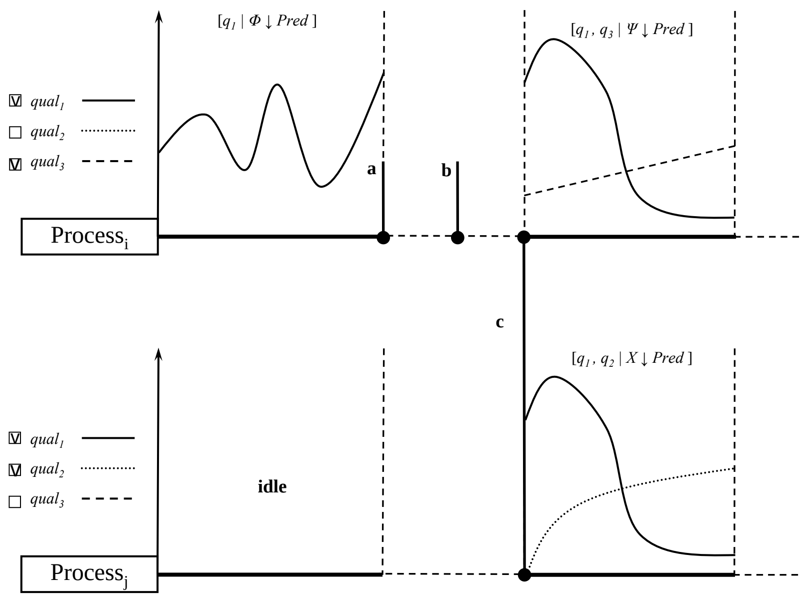


Figure 6.7: Adding MSP expressions

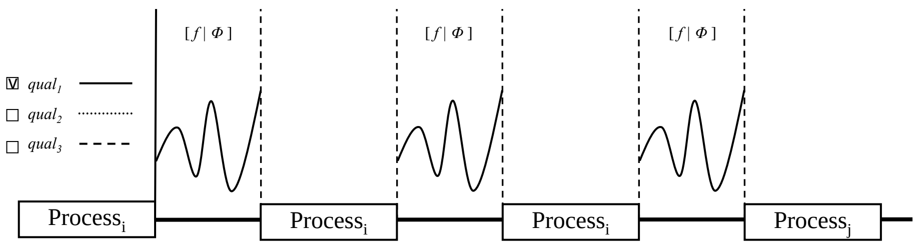


Figure 6.8: Recursion

apply for some, but not all plots in the MSP. Hence the legend always applies to all plots and the current renaming for the qualifier can be found along the plot. Consider BHPC expression

$$\Big([f,g,h \mid \Phi] \,.\, \mathsf{a} \,.\, [f,g,h \mid \Psi] \,.\big(\mathsf{a}\,.\,[f,g,h\mid \Xi]\,.\,\mathbf{0}\big)[y/f, b/a]\Big)[x/f]$$

for which we give the MSP with added renaming in Figure 6.9. In this figure only the second action will be renamed and for the first two trajectory plots qualifier f will be renamed to x and for the third to y.

Introduction of parallelism At the introduction of parallelism, we fork the original process into two processes. Again, we keep the time-lines of all processes in the MSP synchronised. The

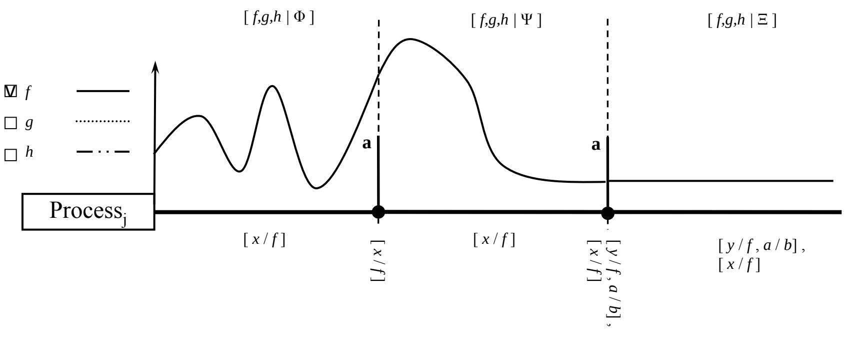


Figure 6.9: Renaming

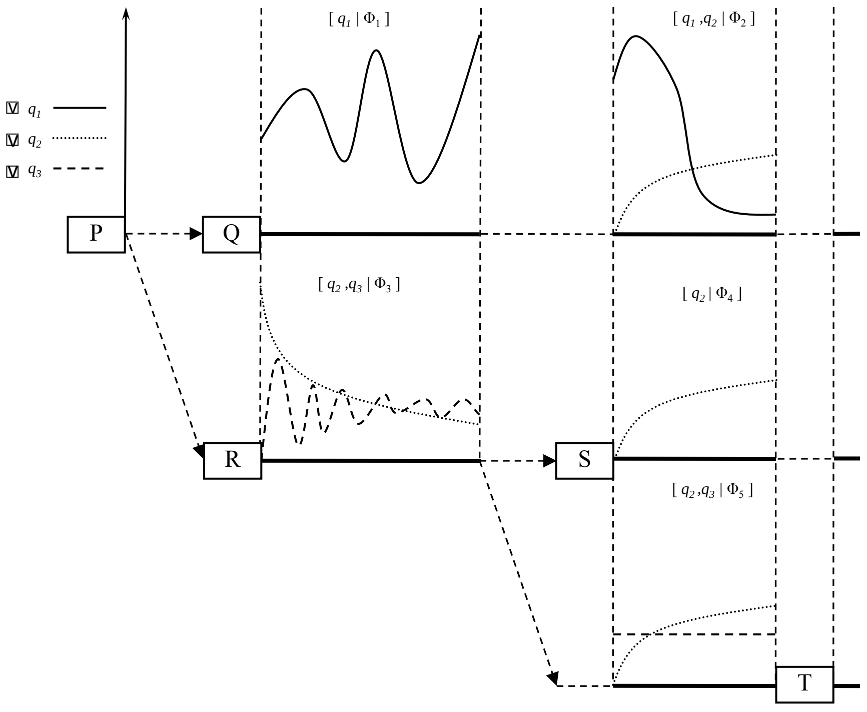
forked time-lines are decorated with a process identifier after execution of the the first recursive call. Consider the following BHPC description where we have omitted trajectory set definitions. Behaviour that will not depicted in the MSP is denoted as "... ".

$$\begin{aligned}
\mathrm{P} &\triangleq \mathrm{Q} \parallel^{q_2} \mathrm{R} \\
\mathrm{Q} &\triangleq [q_1 \mid \Phi_1] \,.\, [q_1, q_2 \mid \Phi_2] \,.\quad \ldots \\
\mathrm{R} &\triangleq [q_2, q_3 \mid \Phi_3] \,.\big(\mathrm{S} \parallel^{q_2} [q_2, q_3 \mid \Phi_5] \,.\mathrm{T}\big) \\
\mathrm{S} &\triangleq [q_2 \mid \Phi_4] \,.\quad \ldots \\
\mathrm{T} &\triangleq \quad \ldots
\end{aligned}$$

Process P is immediately forked into two processes, namely Q and R. Eventually R is forked into process S and a right component that becomes process T after execution of a trajectory. Figure 6.10 depicts the MSP for process P.

In the figure P is forked into process Q and R. We can see that both processes can execute their first trajectory in parallel. Moreover we can see that both trajectories are partially equivalent; they have no qualifiers in common and have the same duration. Subsequently, process R forks into two new processes. Since its left component immediately performs a recursive call, the horizontal line for this process is begins with a process identifier. Then we see execution of three parallel trajectories. Moreover we can see that all trajectories are partially equivalent; they have the synchronising qualifier in common and have the same duration. After parallel execution of this trajectory, the horizontal line for the right component is also decorated with a process identifier.

Folding & unfolding From the previous examples it is clear that the developer and user will have to choose between the amount of represented information and clarity. Therefore, the user should be allowed to choose what he wants to see, and be able to hide (fold) or expose



☑ q_1
☑ q_2
☑ q_3
P
Q
R
S
T
$[\, q_1 \mid \Phi_1 \,]$
$[\, q_1, q_2 \mid \Phi_2 \,]$
$[\, q_2, q_3 \mid \Phi_3 \,]$
$[\, q_2 \mid \Phi_4 \,]$
$[\, q_2, q_3 \mid \Phi_5 \,]$

Figure 6.10: Fork in MSP

(unfold) parts of MSP. For example, folding and unfolding can be introduced to control visibility of the parallel composition components. When the components are folded into a single process, the communication between these processes should be depicted in some other way, e.g. as lines with action names that only cross this process line perpendicularly at both sides.

6.3.4 Plot scales

Choosing a suitable plot scale for MSP is not trivial. When in one plot we want to depict qualifiers on a very different scale we do not want to stretch the vertical scale lines too much, since that can diminish overview of different processes. Moreover, we do not want to depict plots below the horizontal line, since then it can become unclear to which process a plot belongs. We exemplify this by the extended bouncing ball in Figure 6.11. From this figure one might think that parts of the plot for velocity could belong to the controller. Only the depicted trajectory prefix for the bouncing ball suggest that it belongs to the bouncing ball.

$$\begin{aligned}
\mathrm{BB}(h_0, v_0)) &\triangleq [h, v \mid \Phi(h_0, v_0)) \Downarrow h = 0]\,.\mathsf{bounce}(c : [0,1]).\\
&\ [h, v \mid \Phi(0, -cv) \Downarrow v = 0]\,.\mathsf{push}(v : \mathbb{R}).\mathrm{BB}(h, v)\\
\mathrm{Control}(v_0) &\triangleq \mathsf{idle}.\mathsf{bounce}(c : [0,1]).\\
&\ \mathsf{idle}.\mathsf{push}\,((1-c)\,v)\,.\mathrm{Control}\,((1-c)\,v)
\end{aligned}$$

$$\text{System}(h, v) \triangleq \text{BB}(h_0, v_0) \,||^{v_0}_{\text{push,bounce}}\, \text{Control}(v_0)$$

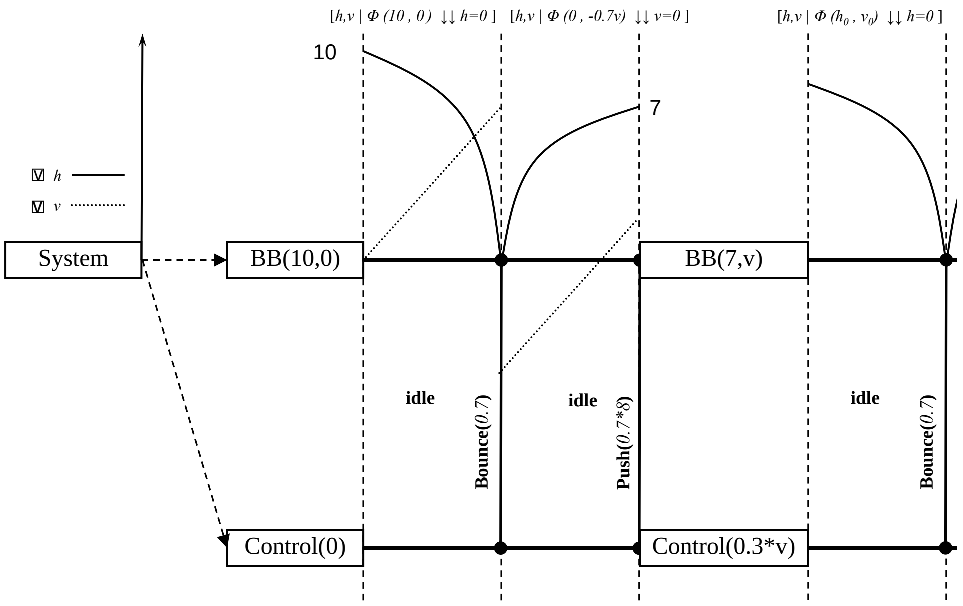


Figure 6.11: MSP for extended bouncing ball

As a solution we propose to only depict plot shapes above the horizontal line within a fixed vertical range. Only then the relation between continuous behaviour, processes and discrete behaviour can be represented in MSP. For detailed analysis of continuous behaviour, the real plot should be easily available, i.e. with a mouse click on the plot shape in MSP.

6.4 Conclusions

Essentially, we need to add support for continuous-time behaviour for a hybrid extension of our discrete engine. We have introduced the notion of event, which represents either an action or a trajectory, and replaced actions in our definitions by events. Also, we have added execution of continuous transitions to our engine, which enable updates of system time and signal space. Most of our definitions, including the simulation algorithm, remain unchanged by these changes.

Obviously the engine needs a connection to a ODE/DAE solver to simulate trajectory prefixes. Trajectory prefixes are simulated after selection of a simulation step. The engine must monitor simulation, i.e. detect events and expiration of the simulation end time, and take corresponding actions. The real challenge lies in solving parallel compositions of trajectories while preserving the order and renaming [11]. There also a restricted prototype is described, that performs limited hybrid simulation. Further research is needed to evaluate suitability of our discrete engine for extension to a hybrid engine.

Our most important contribution to the analysis of hybrid systems is a new visualisation technique (MSP), that can be used to visualise hybrid behaviour. In contrast to other visualisation techniques, where discrete and continuous-time behaviour are represented separately, MSP combines representation of both behaviours and depicts their relation. Analysis of this relation is vital for analysis of hybrid behaviour where both behaviours interact and influence one another. Therefore, we believe in the potential of MSP for hybrid system analysis. The proposed technique can be easily adopted to other hybrid system modelling frameworks with minimal changes, e.g. if communication is directed, arrows can be used to depict it. Formalisation and implementation is needed to investigate the applicability of MSP.

7 Conclusions

With this thesis we take a first step towards hybrid simulation of hybrid systems by developing theory and a prototype for discrete simulation of Behavioural Hybrid Process Calculus (BHPC). BHPC is a hybrid algebraic framework, recently developed by [8, 9, 11], that is suitable for modelling and analysis of hybrid systems. It manifests important properties, such as a strong bisimulation relation that is a congruence with respect to parallel composition of subsystems. However, without a simulation tool, the analysis of hybrid systems modelled in BHPC will be a cumbersome task. Simulation tools can be used for automated analysis of model behaviour by generation of different model scenarios, checking for errors and weaknesses in the model. Results can be used to gain understanding in hybrid systems and improve their behaviour. In this final chapter we summarise and evaluate all individual results given throughout the thesis and present some suggestions for future research.

7.1 Abstraction techniques

To perform discrete simulation for BHPC, we have investigated abstraction techniques to abstract from continuous-time behaviour. For these techniques we assumed that abstraction could be performed by treating trajectories as special actions, or representatives. An extension of the existing semantics was necessary to create distinction between actions and representatives, which was necessary to prevent synchronisation between actions and representatives in parallel composition. These techniques are also applicable to other hybrid algebraic frameworks where discrete and continuous-time behaviour are treated on equivalent terms, i.e. they are both treated as algebraic elements for which SOS-rules are defined.

Information about trajectory durations is inevitably lost during abstraction from continuous-time behaviour. As a result discrete simulation cannot guarantee correct traces in the presence of parallelism. For hybrid systems containing parallelism, discrete simulation traces should always be investigated to estimate their correctness. Such an investigation can be a cumbersome task for more complex systems. Therefore we recommend hybrid simulation for hybrid systems containing parallelism. For all other systems discrete simulation can be used to gain insight in control structures.

7.2 Discrete definition

We have defined a discrete engine, based on BHPC semantics that is extendible for new BHPC developments. Our discrete engine definition is based on our assumption that for relating engine procedures to BHPC semantics, the approach from [32] for determining and execution of simulation steps can be used. Moreover, we presumed that a modular definition of the engine, that encapsulates determination and execution of simulation steps, is necessary for extendibility of the engine.

We modified the definitions for determining and execution of simulation steps to represent the semantics of BHPC. The most notable difference with [32] is that we derive simulation steps from action prefixes as final step of our calculation. The benefit of using action prefixes during calculation is that they represent a normal form and the expansion law can easily be applied to solve parallelism. This approach is also applicable for other algebraic frameworks and due to modularity of the engine definition, this will affect only encapsulated modules.

7.3 Design & implementaion

We have tested our theory and definitions by building a prototype based on the discrete engine definition. There is a clear mapping between the engine definitions and implemented procedures in our prototype and the prototype was able to simulate hybrid systems modelled in BHPC.

During development of the prototype we started with a framework for the tool and gradually added BHPC processes. In essence we have experienced the suitability of the prototype to serve as tool to experiment with extensions and changes to BHPC. From our experience adding new BHPC procedures can be done in a very structural manner and therefore we are confident in the suitability of the prototype as "sand-box" for further developments of BHPC.

7.4 Towards hybrid simulation

7.4.1 Hybrid engine definition

We presented a beginning and some ideas for defining a hybrid engine that is based on our discrete engine. As assumed, most of the original engine definition remains unaffected by the hybrid extensions. The real remaining challenge for hybrid extension is resolving trajectories. We refer to [11] for detailed description of resolving continuous-time behaviour and the description of a restricted prototype that performs limited hybrid simulation. Further research is needed to evaluate suitability of our discrete engine for extension to a hybrid engine. Hybrid definitions should be completed and a restricted simulation as described in [11] can be used as a starting point for hybrid definitions and an implementation.

7.4.2 Hybrid visualisation

Finally we presented a new and promising technique for the visualisation of hybrid behaviour. This technique, called MSP, combines the visualisation of discrete behaviour and continuous-time behaviour, by rotated message sequence charts and plots respectively, and moreover their relation. We also presented some extensions for MSP to facilitate a more elaborate analysis of hybrid systems. We believe that MSP has great potential in visualising hybrid system behaviour since it represents hybrid behaviour instead of a separate representation of the behaviours. Further research is needed to formalise this technique and evaluate its applicability by a prototype.

7.5 Implementation extensions

Here we enumerate some possible extensions for our prototype.

Integration with BHPC parser Once the BHPC parser is finished the engine and the parser should be integrated. Then the parser can generate models in internal representation from ASCII representation, whereas now internal representation is generated by hand.

Integration with a visualisation unit In our prototype simulation traces are represented by a textual format. Integration with a visualisation unit facilitates a more convenient representation of these traces.

Selection Selection of events can only be done manually in our prototype. Other selection modes, like automatic and batch selection, could be added.

Undoing transitions For more freedom, the possibility of going one or more steps back in simulation and undoing transitions would be useful.

7.6 Final remarks

From an engine definition based on BHPC semantics, we presented a general and modular implementation, that proved to be extendible for BHPC developments. As a result the prototype can contribute to further BHPC developments as “sand-box” for testing new theory. Also we presented a new visualisation technique that visualises hybrid behaviour by combining the visualisation of discrete behaviour, continuous-time behaviour and their relation. Analysis of this relation is vital for understanding hybrid behaviour in which both behaviours interact and influence each other. These results have contributed to our goal to develop techniques and a prototype for discrete simulation of BHPC that could be used to analyse and improve hybrid system behaviour.

Bibliography

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[2] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer, 1993.

[3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer, September 2004.

[4] J. Bengtsson, W.O.D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Automated analysis of an audio control protocol using Uppaal. *Journal of Logic and Algebraic Programming*, 52–53:163–181, July 2002.

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

[6] J.A. Bergstra and C.A. Middelburg. Process algebra for hybrid systems. Technical report, Dept. of Math. and Comp. Science, Technical University of Eindhoven (TU/e), P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands, 2003.

[7] T. Bolognesi and H. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[8] E. Brinksma and T. Krilavičius. Behavioural hybrid process calculus. Technical Report TR-CTIT-05.45, CTIT, University of Twente, 2005.

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

[10] L. Carloni, M.D. Di Benedetto, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli. Modeling techniques, programming languages and design toolsets for hybrid systems. Technical Report Deliverable DHS4-5-6, Project IST-2001-38314 COLUMBUS, 2004.

[11] T. Krilavičius. *Hybrid Techniques for Hybrid Systems*. PhD thesis, Formal Methods and Tools, University of Twente, 2006.

[12] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronisation skeletons using branching time logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.

[13] P.J.L. Cuijpers and M.A. Reniers. Hybrid process algebra. Technical report, Dept. of Math. and Comp. Science, Technical University of Eindhoven (TU/e), P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands, 2003.

[14] H. Eertink. *Simulation Techniques for the Validation of LOTOS Specifications*. PhD thesis, University of Twente, 1994.

[15] S. Hedlund. Computational methods for hybrid systems. Licentiate thesis ISRN LUTFD2/TFRT--3225--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, September 1999.

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

[17] ITU-T. Recommendation Z.120. Message Sequence Charts. Technical Report Z-120, International Telecommunication Union – Standardization Sector, Genève, 2000.

[18] A.S.A. Jeffrey, S.A. Schneider, and F.W. Vaandrager. A comparison of additivity axioms in timed transition systems. Report CS-R9366, CWI, Amsterdam, November 1993.

[19] A.A. Julius. *On Interconnection and Equivalence of Continuous and Discrete Systems: A Behavioral Perspective*. PhD thesis, Systems Signals and Control Group, University of Twente, 2005.

[20] D.K. Kaynar, A. Chefter, L. Dean, S. Garland, N. Lynch, T.N. Win, and A. Ramirez-Robredo. The IOA simulator. Technical Report MIT-LCS-TR-843, MIT Laboratory for Computer Science, Cambridge, MA, July 2002.

[21] E.A. Lee and H. Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control: 8th International Workshop, HSCC*, LNCS, pages 25–53, February 2005.

[22] N.A. Lynch, R. Segala, and F.W. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[23] K.L. Man and R.R.H. Schiffelers. *Formal specification and analysis of hybrid systems*. PhD thesis, Technical University of Eindhoven (TU/e), 2006.

[24] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

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

[26] T.A. Henzinger R. Alur and W. Wong-Toi. The theory of hybrid automata. In *Proceedings of the 36th Annual Conference on Decision and Control (CDC)*, pages 702–707. IEEE Press, 1997.

[27] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on message sequence charts. *Comput. Netw. ISDN Syst.*, 28(12):1629–1641, 1996.

[28] A. Samarin. Application de la programmation rèactive á la modélisation en physique, 2002.

[29] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. Report CS-Report 04-37, Technical University of Eindhoven (TU/e), Eindhoven, November 2004.

[30] A.J. van der Schaft. Bisimulation of dynamical systems. In R. Alur and G. J. Pappas, editors, *HSCC*, volume 2993 of *LNCS*, pages 555–569. Springer, 2004.

[31] A.J. van der Schaft and J.M. Schumacher. *An Introduction to Hybrid Dynamical Systems*, volume 251 of *LNCIS*. Springer, London, 2000.

[32] P. van Eijk. *Software tools for the specification language LOTOS*. PhD thesis, University of Twente, 1988.

[33] A.E. van Putten. Behavioural hybrid process calculus parser and translator to Modelica. Technical report, University of Twente, 2006. Draft of master thesis.

[34] B.P. Zeigler, H. Praenhofer, and T.G. Kim. *Theory of Modelling and Simulation*. Academic Press, second edition, 2000.

A
BHPC Modelling Language

Models of hybrid systems in BHPC need additional constructs to provide the compiler all necessary information for simulation. For example the compiler needs identification of actions, processes and the initial process. See [33] for details about additional constructs needed for the compiler. In this appendix we describe the syntax of BHPC and these constructs. The syntax of the BHPC modelling language is represented in EBNF, in which terminal symbols are represented by capital bold words for enhanced readability. For terminals we provide both the ASCII and the mathematical representation. Additional constructs for the compiler only have an ASCII representation. First we will give an informal description of the language, the scope of the BHPC operators and a BHPC model in mathematical and ASCII representation.

A.1 Input format

The compiler requires BHPC models to be written in ASCII format such that models for the compiler can be written in an ordinary editor. As said, the compiler requires more than just an ASCII representation of the BHPC model. In this section we illustrate the difference between mathematical and ASCII models.

Mathematical models are models described in BHPC, which has been described in Chapter 2.3. Figure A.1 shows the mathematical model for the bouncing ball example we have given before. The model contains a process definition for the bouncing ball (BB) and a trajectory set definition (Φ). In Figure A.2 the ASCII representation of the bouncing ball is given.

$$
\begin{aligned}
\mathrm{BB}(h_0, v_0) &\triangleq [h, v \mid \Phi(h_0, v_0) \Downarrow h = 0] \,.\mathrm{BB}(0, -c * v) \\
\Phi(h_0, v_0) &= \{h, v : (0, t] \to \mathbb{R} \mid \\
&\qquad h(0) = h_0, v(0) = v_0, \dot{h} = v, \dot{v} = -g, h \geqslant 0\}
\end{aligned}
$$

Figure A.1: Bouncing ball, mathematical representation

```
constants: (c,10), (g,10)
qualifiers: h,v

initial BB(8,10)

proc BB (h_0,v_0) ^=
    [ h,v | tset phi(h_0,v_0) exit h=0 ] . BB(0,-c*v)

tset phi (h_0,v_0) = {h,v:(0,t]->RR |
             h(0)=h_0, v(0)=v_0, der(h)=v, der(v)=-g , h>=0 }
```

Figure A.2: Bouncing ball, ASCII representation

Consider the ASCII model in Figure A.2. Apart from the differences between mathematical symbols and their ASCII representation, three differences between the mathematical model and the ASCII model are illustrated. Declaration of constants and qualifiers, declaration of the initial process and indication of process definitions and trajectory set definitions by a keyword.

For recognition the parser requires declaration of all constants, actions and qualifiers names are used in the model. These declaration should be in the beginning of the model, in any order. Moreover, all action names, qualifiers and constants have to be unique. Declarations of actions, qualifier, or constants start with the keywords `actions`, `qualifiers`, or `constants` respectively. Figure A.2 illustrates the definition of constants and qualifiers.

The engine must be able to determine the initial process to start simulation. For more freedom in naming process the initial process does not have a fixed name, but this requires a declaration of the initial process. A declaration of the initial process starts with the keyword `initial`. In Figure A.2 process BB is declared to be the initial process of the model and the model contains a process definition for process BB. Simulation can not be performed without an associated process definition for the initial process. Initialisation of the bouncing ball process requires actual parameter values for the initial process .

For convenient recognition of process definitions and the trajectory set definitions, the parser requires the definitions to be preceded by the keywords `proc` and `tset` respectively.

The ASCII representation for BHPC processes is given in Table A.1. In the forthcoming sections we give the scope and syntax of BHPC.

Process	BHPC	ASCII
Stop	$\mathbf{0}$	`0`
Action Prefix	$\mathsf{a}.B$	`a.B`
Trajectory Prefix	$[\varphi]\,.B$	`[phi].B`
Symbolic Trajectory Prefix	$[f \mid \Phi]\,.B$	`[f | tset PHI].B`
Guard	$\langle \mathcal{P}red \rangle\,.B$	`<`*Pred*`>.B`
Choice	$\sum\limits_{i \in I} B_i$	`choice{i in I} B_i`
Binair Choice	$B_1 + B_2$	`B_1 + B_2`
Parallel Composition	$B_1 \parallel_A^H B_2$	`B_1 |{q_1,...,q_n},{a_1,...,a_n}| B_2`
Hiding	$\mathsf{new}\ w.B$	`new{a_1,...,a_n}.B`
Renaming	$B\,[\sigma]$	`B[old_1/new_1,...,old_n/new_n]`
Recursion	P	`P`

Table A.1: Translation table for BHPC processes

A.2 Scope

In Table A.2 operator precedence is defined for BHCP

Description	Operator	Type	Precedence
Action Prefix	$\mathsf{a}.$	prefix	1
Trajectory Prefix	$[\varphi]\,.$	prefix	1
Guard	$\langle \mathcal{P}red \rangle\,.$	prefix	1
Choice	$\sum\limits_{i \in I}$	prefix	2
Choice	$+$	infix	2
Parallel Composition	$\parallel_A^H$	infix	3
Renaming	$[\sigma]$	postfix	4
Hiding	$\mathsf{new}\ w.$	prefix	4

Table A.2: Operator precedence in BHPC

So expression

$$\mathsf{new}\ w.\mathsf{a}.P + [\varphi]\,.Q + R \parallel_A^H S[\sigma]$$

is equivalent to expression

$$\mathsf{new}\ w.\Big(\big((\mathsf{a}.P) + ([\varphi]\,.Q)\big) + (R \parallel_A^H S[\sigma])\Big)$$

Hiding and renaming have the same precedence since they are similar processes. Hiding is a special case of renaming where all elements are mapped to the internal action τ. Precedence in hiding and renaming has to be made explicit by using parenthesis. Hence

$$\mathsf{new}\ w.B\,[\sigma]$$

has no meaning and has to be written as either

$$\mathsf{new}\ w.(B\,[\sigma]) \quad \text{or} \quad (\mathsf{new}\ w.B)\,[\sigma]$$

A.3 Syntax

The syntax of BHPC and additional constructs is presented in BNF (Backus-Naur form), where terminal symbols will be displayed in bold capital font for enhanced readability. The keywords and symbols for the terminals can be found in Appendix A.4.

Model Definition

The definitions of constants, actions, or qualifiers (any number, any order) are followed by an obligatory initial process definition. The remainder of the model consists of process definitions and trajectory set definitions.

$$\begin{aligned}\text{Model} \quad ::= \quad & (\ \text{Const\_Def} \ \big|\ \text{Action\_Def} \ \big|\ \text{Qual\_Def})^{*} \\ & \text{Initial\_Process} \\ & (\text{Proc\_Definition} \ \big|\ \text{TSet\_Definition})^{*}\end{aligned}$$

Constant Definition

Constant definitions are used for recognition of constants in the model and to define their values. Constants can be defined in the beginning of the model. The appearance of "`constants:`" marks the start of a constant definition. A constant definition may contain zero, or more (comma separated) constant declarations.

$$\text{Const\_Def} \quad ::= \quad \textbf{CONST} \quad \textbf{COLON} \quad [\text{Const\_Decl} \quad (\textbf{COMMA} \quad \text{Const\_Decl})^{*}]$$

Constant Declaration

A constant declaration is enclosed in parenthesis and contains both the constants name and its value. The value can be given as expression.

$$\text{Const\_Decl} \quad ::= \quad \textbf{LPAREN} \quad \text{Identifier} \quad \textbf{COMMA} \quad \text{Expression} \quad \textbf{RPAREN}$$

Action Definition

Action definitions are used to recognize actions in the model. Actions can be defined in the beginning of the model. The beginning of an action definition is indicated by "`actions:`". An action definition may contain zero, or more (comma separated) actions.

Action_Def ::= **ACTION** **COLON** [Identifier (**COMMA** Identifier)*]

Qualifier Definition

Qualifier definitions are used to recognize qualifiers in the model. Qualifiers can be defined in the beginning of the model. The beginning of an qualifier definition is indicated by "`qualifiers:`". A qualifier definition may contain zero, or more (comma separated) qualifiers.

Qual_Def ::= **QUAL** **COLON** [Identifier (**COMMA** Identifier)*]

Initial Process

In each model an initial process must be defined. Optionally, a list of parameter expressions is given. These parameters are used to instantiate the process.

Initial_Process ::= **INITIAL** Identifier [ParameterList]

Process Definition

Processes can be defined in a process definition. An example of the mathematical representation of a process definition can be found in Example A.3.1.

Example A.3.1 (BHPC Process Definition).

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

Where:

• BB	Name for the defined process
• (h_0, v_0)	Parameters for the defined process
• $\triangleq$	Start of the process definition
• $[h, v \mid \Phi(h_0, v_0) \Downarrow h = 0]$.	BHPC expression for a trajectory prefix operator
• $\mathrm{BB}(0, -c * v)$	Reference (recursion) to a process named BB, with parameters$(0, -c * v)$

□

The ASCII representation of the process definition is preceded by the keyword "`proc`" and contains the ASCII representation for all mathematical symbols.

ProcessDefinition ::= **PROC** Identifier [ParameterList] **EQDEF** ProcessExpr

Trajectory Set Definition

Trajectory sets can be defined in a trajectory set definition. An example of the mathematical representation of a trajectory set definition can be found in Example A.3.2.

Example A.3.2 (BHPC Trajectory Set Definition).

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

Where:

• Φ	Name for the defined trajectory set
• (h_0, v_0)	Parameters for the trajectory set
• $=$	Start of the set definition
• $h, v : (0, t] \to \mathbb{R}$	Trajectories h and v
• $h(0) = h_0, v(0) = v_0$	Expressions (Assignments)
• $\dot{h} = v, \dot{v} = -g$	Expressions (Differential equations)
• $h \geqslant 0$	Expression (Guard)

□

The ASCII representation of the trajectory set definition is preceded by the keyword "`tset`" and contains the ASCII representation for all mathematical symbols.

TSet_Definition ::= **TSET** Identifier [ParameterList] **EQ**
LCURLY (Trajectory)*
[**PIPE** Expression (**COMMA** Expression)*]
RCURLY

BHPC process expressions

BHPC processes are all kind of expressions that can be made using the BHPC syntax given in Chapter 2.3.

ProcessExpr ::= Stop | ActionPrefix | TrajectoryPrefix | Guard | Choice
| ParallelComposition | Hiding | Renaming | Recursion

Stop	::=	**STOP**
ActionPrefix	::=	Action **SEQC** ProcessExpr
TrajectoryPrefix	::=	TrajectoryPrefixOp ProcessExpr
Guard	::=	**LANGLE** Expression **RANGLE** **SEQC** ProcessExpr
Choice	::=	ProcessExpr (**CHOICE** ProcessExpr)$^+$
ParallelComposition	::=	ProcessExpr **PIPE** NSet **COMMA** NSet **PIPE** ProcessExpr
Hiding	::=	**NEW** NSet **SEQC** ProcessExpr
Renaming	::=	ProcessExpr **LBRACK** RenamingMap **RBRACK**
Recursion	::=	Identifier [ParameterList]

These rules are a straightforward translation into ASCII. Note that we only have binary choice.

Action

An action consists of a name and optionally a list of parameters can be given.

Action	::=	Identifier [ParameterList]

Renaming Map

The renaming map may be empty or contain one or more comma separated renaming elements.

RenamingMap	::=	**LBRACK**
		[RenamingElement (**COMMA** RenameElement)*]
		RBRACK

Renaming Element

A renaming element contains the renaming for an action or qualifier. The old name is preceded by a slash and the new name.

RenamingElement	::=	Identifier **SLASH** Identifier

Name Set

The name set is used as a container for actions and qualifiers. A name set can appear in parallel composition as container for the synchronising names, or in hiding as container for the internal names. The set may be empty or contain one or more comma separated elements.

NSet ::= **LCURLY** [Identifier (**COMMA** Identifier)*] **RCURLY**

Trajectory Prefix Operator

There are two types of trajectory prefix operators: simple trajectory prefix operators and symbolic trajectory prefix operators. The simple trajectory prefix operator (e.g. $[\varphi]$.) contains a trajectory and the symbolic trajectory operator (e.g. $[h, v \mid \Phi(h_0, v_0) \Downarrow h = 0]$.) consists of one or more (comma separated) qualifiers and a reference to a trajectory set. Optionally restrictions and exit conditions can be added to the trajectory set reference.

TrajectoryPrefix ::= SimplePrefix | SymbPrefix

SimpelPrefix ::= **LBRACK** Trajectory **RBRACK**

SymbPrefix ::= **LBRACK** Qualifier (**COMMA** Qualifier)*
PIPE TsetReference **RBRACK**

Trajectory

A trajectory is a mapping of a time interval to signals of the signal space, e.g. $(\varphi_{\mathbb{W}} : (0, t] \to W_1 \times \cdots \times W_n)$. One or more qualifiers can be associated with the same mapping. This is the case in the trajectory set definition we gave earlier

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

where both h and v are associated with the same trajectory.

Trajectory ::= Identifier (**COMMA** Identifier)*
COLON DurationDomain **TO** SignalDomain

TSetReference

A trajectory set reference is part of the symbolic trajectory prefix operator and refers to a trajectory set definition. Reference is made by the name of the trajectory set definition and optionally some parameters. It is possible to define restrictions and exit conditions for the trajectory set.

TsetReference ::= Identifier [Parameters]
[**RESTRICT** Expression (**COMMA** Expression)*]
[**EXIT** Expression (**COMMA** Expression)*]

Parameters

Parameter lists are enclosed in brackets and consist of one or more parameter names.

$$\text{ParameterList} \quad ::= \quad \textbf{LPAREN} \quad \text{Identifier} \quad (\,\textbf{COMMA} \quad \text{Identifier})^{*} \quad \textbf{RPAREN}$$

Identifiers

Actions, qualifiers, process identifiers, parameters and signalnames are all identifiers. Identifiers can be any string of at least one letter, followed by any number of letters and numbers.

$$\text{Identifier} \quad ::= \quad \left('a'..'z' \;\middle|\; 'A'..'Z'\right) \left('a'..'z' \;\middle|\; 'A'..'Z' \;\middle|\; '0'..'9' \;\middle|\; '\_'\right)^{*}$$

Expressions

Basic expressions, Guards, assignment are all expressions. The exact definition of expressions is left open for future developments for hybrid simulation.

Duration domain

The duration domain is a left open interval that defines the begin and end time for the trajectory. The exact definition of this domain is left open for future developments for hybrid simulation.

Signal domain

The signal domain is the domain to which signals are mapped. The exact definition of this domain is left open for future developments for hybrid simulation.

A.4 Language keywords & symbols

In Table A.3 the BHPC modelling language keywords and symbols for all used grammar tokens are presented. There are no mathematical representations for the keywords with which the BHPC modelling language extends the calculus.

Grammar Terminal	ASCII Representation	Mathematical representation
ACTION	`actions`	
BECOMES	`:=`	$:=$
CHOICE	`+`	$+$
COLON	`:`	$:$
COMMA	`,`	$,$
CONST	`constants`	
EQ	`=`	$=$
EQDEF	`^=`	$\triangleq$
LANGLE	`<`	$\langle$
LBRACE	`{`	$\{$
LBRACK	`[`	$[$
LPAREN	`(`	$($
INITIAL	`initial`	
NEW	`new`	new
PIPE	`|`	$\vert$
PROC	`proc`	
SLASH	`/`	$/$
QUAL	`qualifiers`	
RANGLE	`>`	$\rangle$
RBRACE	`}`	$\}$
RBRACK	`]`	$]$
RPAREN	`)`	$)$
SEQC	`.`	$.$
STOP	`stop`	$\mathbf{0}$
TO	`->`	$\rightarrow$
TSET	`tset`	

Table A.3: Language keywords and symbols

B
Menu Calculation

Table 3.2 contains the definition for $\textsc{Menu}_{NF}$for all BHPC expressions. In this Appendix we give derivations for all definitions from the table. Note that these derivation are based on BHPC semantics. For convenience we repeat the definition of $\textsc{Menu}_{NF}$ for BHPC first.

Definition B.0.1 ($\textsc{Menu}_{NF}$ for BHPC). Let B be the set of BHPC process expressions, $\mathsf{a} \in \mathcal{A}$ be an action and T be the set of simulation steps. Then we define the function $\textsc{Menu}_{NF} : B \rightarrow T$

$$\textsc{Menu}_{NF}(B) = \{(\mathsf{a}.B') \mid B \xrightarrow{\mathsf{a}} B'\}$$

□

Stop (**0**)

$\textsc{Menu}_{NF}(\mathbf{0}) = \{(\mathsf{a}.B') \mid \mathbf{0} \xrightarrow{\mathsf{a}} B'\}$
and since $\mathbf{0} \not\rightarrow$, we have

$$\textsc{Menu}_{NF}(\mathbf{0}) \quad = \quad \emptyset$$

Action Prefix (a.B)

$\textsc{Menu}_{NF}(\mathsf{a}.\mathrm{B}) = \{(\mathsf{a}.B') \mid \mathsf{a}.B \xrightarrow{\mathsf{a}} B'\}$
and since

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

$$\textsc{Menu}_{NF}(\mathsf{a}.B) \quad = \quad \{(\mathsf{a}.B)\}$$

Choice $(\sum_{v\in I} B(v))$

$\textsc{Menu}_{NF}(\sum_{v\in I} B(v)) = \{(\mathsf{a}.B') \mid \sum_{v\in I} B(v) \xrightarrow{\mathsf{a}} B'\}$
and since

$$\frac{B(w) \xrightarrow{\mathsf{a}} B'}{\sum_{v\in I} B(v) \xrightarrow{\mathsf{a}} B'} \quad w \in I \tag{B.1a}$$

for each component of the choice, we have that the choice of the components can evolve.

$$\begin{aligned}\textsc{Menu}_{NF}(\sum_{v\in I} B(v)) &= \bigcup_{v\in I} \{(\mathsf{a}.B') \mid B(v) \xrightarrow{\mathsf{a}} B'\} \\ &= \bigcup_{v\in I} \textsc{Menu}_{NF}(B(v))\end{aligned}$$

Parallel Composition $(B \parallel_A^H C)$
Where $B = \bigoplus_{i\in I} \mathsf{b}_i.B_i$ and $C = \bigoplus_{k\in K} \mathsf{c}_k.C_k$

$\textsc{Menu}_{NF}(B \parallel_A^H C)$

$$= \textsc{Menu}_{NF}\Big(\sum_{\substack{i\in I\\ \mathsf{b}_i\notin A}} \mathsf{b}_i.(B_i \parallel_A^H C) + \sum_{\substack{k\in K\\ \mathsf{c}_k\notin A}} \mathsf{c}_k.(B \parallel_A^H C_k) + \sum_{\substack{i\in I,k\in K\\ \mathsf{b}_i\in A,\\ \mathsf{b}_i=\mathsf{c}_k}} \mathsf{b}_i.(B_i \parallel_A^H C_k)\Big) \tag{B.2}$$

$$= \textsc{Menu}_{NF}\Big(\bigoplus_{\substack{i\in I\\ \mathsf{b}_i\notin A}} \mathsf{b}_i.\big(B_i \parallel_A^H C\big)\Big) \tag{B.3}$$

$$\cup \quad \textsc{Menu}_{NF}\Big(\bigoplus_{\substack{k\in K\\ \mathsf{c}_k\notin A}} \mathsf{c}_k.\big(B \parallel_A^H C_k\big)\Big) \tag{B.4}$$

$$\cup \quad \textsc{Menu}_{NF}\Big(\bigoplus_{\substack{\mathsf{a}=\mathsf{b}_i=\mathsf{c}_k\\ \mathsf{a}\in A}} \mathsf{a}.\big(B_i \parallel_A^H C_k\big)\Big) \tag{B.5}$$

B.2 is based on application of the expansion law (Theorem 2.4.2). B.3, B.4,B.5 are based on application of $\textsc{Menu}_{NF}$ for choice.

Recursion (P)

$\textsc{Menu}_{NF}(P) = \{(\mathsf{a}.B') \mid P \xrightarrow{\mathsf{a}} B'\}$
and since

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

$$\begin{aligned}\textsc{Menu}_{NF}(P) &= \{(\mathsf{a}.B') \mid B \xrightarrow{\mathsf{a}} B'\} \quad \text{, where } P \triangleq B \\ &= \textsc{Menu}_{NF}(B) \quad \text{, where } P \triangleq B\end{aligned}$$

Renaming $(B\,[\sigma])$

$\textsc{Menu}_{NF}(B\,[\sigma]) = \{(\mathsf{a}.B') \mid B\,[\sigma] \xrightarrow{\mathsf{a}} B'\}$
using

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

$$\begin{array}{rcl} \textsc{Menu}_{NF}(B\,[\sigma]) & = & \{(\sigma(\mathsf{a}).B'\,[\sigma]) \mid B \xrightarrow{\mathsf{a}} B'\} \\ & = & [\sigma]\,\textsc{Menu}_{NF}(B) \end{array}$$

Guard $(\langle Pred\rangle.B)$

$\textsc{Menu}_{NF}(\langle Pred\rangle.B) = \{(\mathsf{a}.B') \mid \langle Pred\rangle.B \xrightarrow{a} B'\}$
and since

$$\frac{B \xrightarrow{\mathsf{a}} B'}{\langle Pred\rangle.B \xrightarrow{\mathsf{a}} B'} \quad \models Pred \tag{B.6a}$$

$$\begin{array}{rcl} \textsc{Menu}_{NF}(\langle Pred\rangle.B) & = & \begin{cases} \{(\mathsf{a}.B') \mid B \xrightarrow{\mathsf{a}} B'\} & \text{, if } \textit{Guard} \text{ is satisfied} \\ \emptyset & \text{, otherwise} \end{cases} \\ & = & \begin{cases} \textsc{Menu}_{NF}(B) & \text{, if } \textit{Guard} \text{ is satisfied} \\ \emptyset & \text{, otherwise} \end{cases} \end{array}$$

Hiding $(\mathsf{new}\ w.B)$
In essence hiding is a renaming function where all element of the hiding vector are renamed to τ (B.7). Hence we will use the $\textsc{Menu}_{NF}$ definition of renaming in the derivation for hiding (B.8). Let the hiding vector $\mathsf{new}\ w = [\mathsf{a}_1, \ldots, \mathsf{a}_n]$.

$$\begin{array}{rcll} \textsc{Menu}_{NF}(\mathsf{new}\ w.B) & = & \textsc{Menu}_{NF}\big(B\,[\tau/\mathsf{a}_1, \ldots, \tau/\mathsf{a}_1]\big) & \text{(B.7)} \\ & = & [\tau/\mathsf{a}_1, \ldots, \tau/\mathsf{a}_1]\,\textsc{Menu}_{NF}(B) & \text{(B.8)} \end{array}$$

C
Internal Specification

In this appendix we present the internal representation of BHPC models in XML. First we give an informal description of the internal format in Appendix C.1. Then we present the validated XML scheme representation in Appendix C.2. For an example of internal XML format we refer to Appedix E, where this representation is used in the simulation trace.

C.1 Informal XML description

The representation is given by XML fragments that are depicted within in boxes. Each XML fragment is labelled outside its box and references to (other) XML fragments may appear within its box. Figure C.1 depicts three labelled XML fragments.

#EXAMPLE
```
<test>
  #TEST1 *
  #TEST2 +
</test>
```

#TEST1
```
<test1> one </test1>
```

#TEST2
```
<test2> two </test2>
```

```
<test>
  <test1> one </test1>
  <test1> one </test1>
  <test2> two </test2>
</test>
```

Figure C.1: XML fragments (left) and a possible insertion of references (right).

Within the first box the XML fragment for #EXAMPLE is given. According to the XML fragment, an #EXAMPLE fragment is enclosed by test tags. Within the test tags there are two references to other XML fragments. A reference represents the insertion of another XML fragment. The first reference may be inserted zero or more times (#TEST1 *), the second reference must be inserted at least once (#TEST2 $^+$). In the right side of Figure C.1 the fragment for TEST1 and TEST2 has been inserted into EXAMPLE twice and once respectively.

C.1.1 BHPC model

#MODEL

```
<bhpc>
 #CONST_DEF
 #ACTION_DEF
 #QUAL_DEF )
 #INITIAL_PROCESS
 ( #PROCESS_DEFINITION ) *
 ( #TSET_DEFINITION ) *
</bhpc>
```

#CONST_DEF

```
<constantdef>
 ( #CONSTANT ) *
</constantdef>
```

#CONSTANT

```
<constant>
  #IDENTIFIER
  #EXPRESSION
</constant>
```

#ACTION_DEF

```
<actiondef>
 ( #IDENTIFIER ) *
</actiondef>
```

#QUAL_DEF

```
<qualifierdef>
 ( #IDENTIFIER ) *
</qualifierdef>
```

#INITIAL_PROCESS

```
<initialprocess>
  #IDENTIFIER
  #PARAMETERS
</initialprocess>
```

#PROCESS_DEFINITION

```
<processdefinition>
  #IDENTIFIER
  #PARAMETERS
  <behaviourexpression>
    #PROCESS
  </behaviourexpression>
</processdefinition>
```

#TSET_DEFINITION

```
<tsetdefinition>
  #IDENTIFIER
  #PARAMETERS
  #TRAJECTORY *
  #CONDITIONS
</tsetdefinition>
```

#PROCESS

STOP: PROCESS

```
<stop> 0 </stop>
```

ACTION_PREFIX: PROCESS

```
<actionprefix>
  #IDENTIFIER
  #PARAMETERS
  #PROCESS
</actionprefix>
```

TRAJECTORY_PREFIX: PROCESS

```
<trajectoryprefix>
  #TPREFIXOP
  #PROCESS
</trajectoryprefix>
```

GUARD: PROCESS

```
<guard>
  #PROCESS
  #EXPRESSION
</guard>
```

CHOICE: PROCESS

```
<choice>
  #PROCESS
  #PROCESS
</choice>
```

PARALLEL_COMPOSITION: PROCESS

```
<parallelcomposition>
  #PROCESS
  #PROCESS
  #SET
</parallelcomposition>
```

HIDING: PROCESS

```
<hiding>
  #PROCESS
  #SET
</hiding>
```

RENAMING: PROCESS

```
<renaming>
  #PROCESS
  #MAP
</renaming>
```

RECURSION: PROCESS

```
<recursion>
  #IDENTIFIER
  #PARAMETERS
</recursion>
```

#SET

```
<set>
  ( #IDENTIFIER ) *
</set>
```

#MAP

```
<map>
  ( #MAPITEM ) *
</map>
```

#MAPITEM

```
<mapitem>
  <old> #NAME </old>
  <new> #NAME </new>
</mapitem>
```

#TPREFIXOP

TRAJPREFIX: TPREFIXOP

```
<trajprefix>
  #TRAJECTORY
</trajprefix>
```

SYMBPREFIX: TPREFIX

```
<symbprefix>
  ( #IDENTIFIER ) *
  #TSET_REFERENCE
</symbprefix>
```

#TRAJECTORY.

TRAJECTORY: TRAJECTORY

```
<trajectory>
  ( #IDENTIFIER ) +
  #MAPPING
</trajectory>
```

#TSET_REFERENCE

```
<tsetreference>
  #IDENTIFIER
  #PARAMETERS
  #RESTRICTIONS
  #EXITCONDITIONS
</tsetreference>
```

#PARAMETERS.

```
<parameters>  ( #EXPRESSION ) *  </parameters>
```

#IDENTIFIER

`<name>` *spelling* `</name>`

C.1.2 Fragments for future design

The following fragments are irrelevant for discrete simulation and therefore left open as future work.

#MAPPING

`<mapping>` *spelling* `</mapping>`

#EXPRESSION

`<expression>` *spelling* `</expression>`

#RESTRICTIONS

`<restrictions>` *spelling* `</restrictions>`

#EXITCONDITIONS

`<exitconditions>` *spelling* `</exitconditions>`

#CONDITIONS

`<conditions>` *spelling* `</conditions>`

C.2 XML scheme

The following XML scheme has been validated in `http://www.w3.org/2001/03/webdata/xsv`. Elements that are not fully designed can be found as "unfinished elements". For this project the contents of these elements is not relevant and is represented by a string.

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


  <!-- SIMPLE ELEMENTS -->

  <xs:element name="old"         type="xs:string"/>
  <xs:element name="new"         type="xs:string"/>


  <!-- UNFINISHED ELEMENTS -->

  <xs:element name="mapping"          type="xs:string"/>
  <xs:element name="expression"       type="xs:string"/>
  <xs:element name="restrictions"     type="xs:string"/>
  <xs:element name="exitconditions"   type="xs:string"/>
  <xs:element name="conditions"       type="xs:string"/>


  <!-- COMPLEX ELEMENTS -->
```

```
<xs:element name="bhpc">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="constantdef" />
      <xs:element ref="actiondef" />
      <xs:element ref="qualifierdef" />
      <xs:element ref="initialprocess" />
      <xs:element ref="processdefinition" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="tsetdefinition"  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- constant definition tag -->


<xs:element name="constantdef">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="constant" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- constant definition tag -->


<xs:element name="constant">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" />
      <xs:element ref="expression" />
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- constant tag -->


<xs:element name="actiondef">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>


<xs:element name="qualifierdef" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- qualifier definition tag -->
```

```
<xs:element name="initialprocess" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" />
      <xs:element ref="parameters" />
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- initial process tag -->


<xs:element name="processdefinition" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" />
      <xs:element ref="parameters" />
      <xs:element name="behaviourexpression" >
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="process" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- process definition tag -->


<xs:element name="tsetdefinition" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" />
      <xs:element ref="parameters"/>
      <xs:element ref="trajectory"  minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="conditions" >
        <xs:complexType>
          <xs:all>
            <xs:element ref="expression" />
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- trajectory set definition tag -->


<xs:element name="process" >
  <xs:complexType>
    <xs:choice>
      <xs:element ref="stop" />
      <xs:element ref="actionprefix" />
      <xs:element ref="trajectoryprefix" />
```

```
        <xs:element ref="guard" />
        <xs:element ref="choice" />
        <xs:element ref="parallelcomposition" />
        <xs:element ref="hiding" />
        <xs:element ref="renaming" />
        <xs:element ref="recursion" />
      </xs:choice>
    </xs:complexType>
  </xs:element> <!-- process tag -->


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


  <xs:element name="actionprefix" >
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="identifier" />
        <xs:element ref="parameters" />
        <xs:element ref="process" />
      </xs:sequence>
    </xs:complexType>
  </xs:element> <!-- action prefix tag -->


  <xs:element name="trajectoryprefix" >
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="tprefixOp" />
        <xs:element ref="process" />
      </xs:sequence>
    </xs:complexType>
  </xs:element> <!-- trajectory prefix tag -->


  <xs:element name="guard" >
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="process" />
        <xs:element ref="expression" />
      </xs:sequence>
    </xs:complexType>
  </xs:element> <!-- guard tag -->


  <xs:element name="choice" >
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="process" />
        <xs:element ref="process" />
      </xs:sequence>
  </xs:complexType>
</xs:element> <!-- choice tag -->


<xs:element name="parallelcomposition" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="process" />
      <xs:element ref="set" />
      <xs:element ref="process" />
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- parallelcomposition tag -->


<xs:element name="hiding" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="process" />
      <xs:element ref="set" />
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- hiding tag -->


<xs:element name="renaming" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="process" />
      <xs:element ref="map" />
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- renaming tag -->


<xs:element name="recursion" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" />
      <xs:element ref="parameters"/>
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- recursion tag -->


<xs:element name="map" >
  <xs:complexType>
    <xs:sequence>
      <xs:element name="mapitem" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
```

```
            <xs:element ref="old" />
            <xs:element ref="new" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- map tag -->


<xs:element name="set" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- set tag -->


<xs:element name="tprefixOp" >
  <xs:complexType>
    <xs:choice>
      <xs:element ref="trajprefix" />
      <xs:element ref="symbprefix" />
    </xs:choice>
  </xs:complexType>
</xs:element> <!-- trajectory prefix operator tag -->


<xs:element name="trajprefix" >
  <xs:complexType>
    <xs:all>
      <xs:element ref="trajectory" />
    </xs:all>
  </xs:complexType>
</xs:element> <!-- traj prefix tag -->


<xs:element name="symbprefix" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="identifier" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="tsetreference" />
    </xs:sequence>
  </xs:complexType>
</xs:element> <!-- trajectory set reference tag -->


<xs:element name="trajectory" >
  <xs:complexType>
    <xs:sequence>
```

```
        <xs:element ref="identifier" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="mapping"  />
      </xs:sequence>
    </xs:complexType>
  </xs:element> <!-- trajectory tag -->


  <xs:element name="tsetreference" >
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="identifier" />
        <xs:element ref="parameters" />
        <xs:element ref="restrictions" />
        <xs:element ref="exitconditions" />
      </xs:sequence>
    </xs:complexType>
  </xs:element> <!-- tsetrefrence tag -->


  <xs:element name="parameters">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="identifier">
    <xs:complexType>
      <xs:all>
        <xs:element name="name" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

D Software Architecture

This appendix globally describes the software structure for the discrete engine. For a detailed description of all procedures we refer to the tool documentation [1]. In this appendix we shall describe all packages developed for the prototype for Discrete Bhave. A high level overview of these packages is given in Figure D.1. The figure shows two packages, namely the Event Routines Package and the package for the discrete engine, the Discrete Bhave package. Recall that the Discrete Bhave package uses the Event Routines package to determine the set of simulation steps.

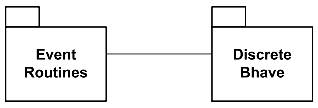


Figure D.1: High level view.

First we shall describe the Discrete Bhave package in Section D.1. In Section D.2 we will describe the Event Routines package. The process package, which is part of the Discrete Bhave package, is described in Section D.3.

D.1 Discrete Bhave Package

The architecture for the Discrete Bhave package is depicted in Figure D.2.

[1] `http://fmt.cs.utwente.nl/tools/bhave/`

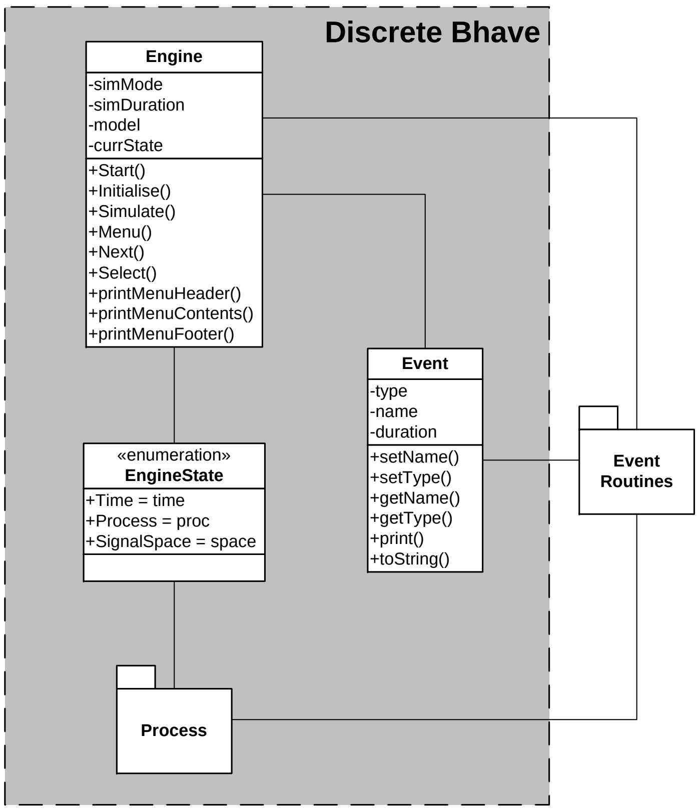


Figure D.2: Discrete Bhave Package.

The engine package consists of the following components

Engine Class The Engine class contains procedures and attributes for simulation. The Simulate procedure implements the simulation algorithm. Menu uses the Event Routine package to determine the set of simulation steps. The print procedures support depiction of the list of simulation steps determined by Menu.

EngineState Enumeration Enumeration EngineState has a time, process expression and signal space component and represents the engine state.

Event Class The Event class is the class that represents actions and abstracted trajectories in the discrete engine. In future developments, this class can also be used for trajectories.

Process Package The process package contains classes for all processes and an XML parser.

D.2 Event Routine Package

The Event Routine Package is responsible of handling events following BHPC semantics. The architecture of this package is depicted in Figure D.3. The EventHandler class contains all necessary procedures for Menu calculation.

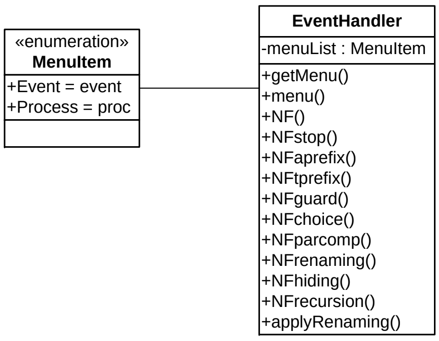


Figure D.3: Event Routines Package.

Procedure description

getMenu Returns the list of simulation steps determined by Menu for the given process. GetMenu is called by the Engine class Menu procedure.

menu Converts a list of action prefix processes into a list of simulation steps. The list of action prefix process is determined by NF.

NF procedures Normal form procedures decompose a process expression into a list of action prefix processes. The procedure NF checks the type of the current expression and invokes the related NF procedure to decompose this process type. For example, when the NF procedure detects an action prefix, NFaprefix is invoked.

applyRenaming Given an action and a renaming map, it returns the renamed action, or the original action in case no renaming for this action is defined in the action map.

D.3 Process Package

The process package contains classes for all processes, the XML parser package TinyXML and a ProcessParser class. TinyXML is the XML parser package that is freely available [2]. Process class procedures call procedures from the ProcessParser for parser interaction. The ProcessParser is a collection of procedures that use the TinyXML parser. For replacement of the TinyXML parser only the ProcessParser class needs to be adapted. Figure D.4 depicts the Process Package.

[2] `http://sourceforge.net/projects/tinyxml`

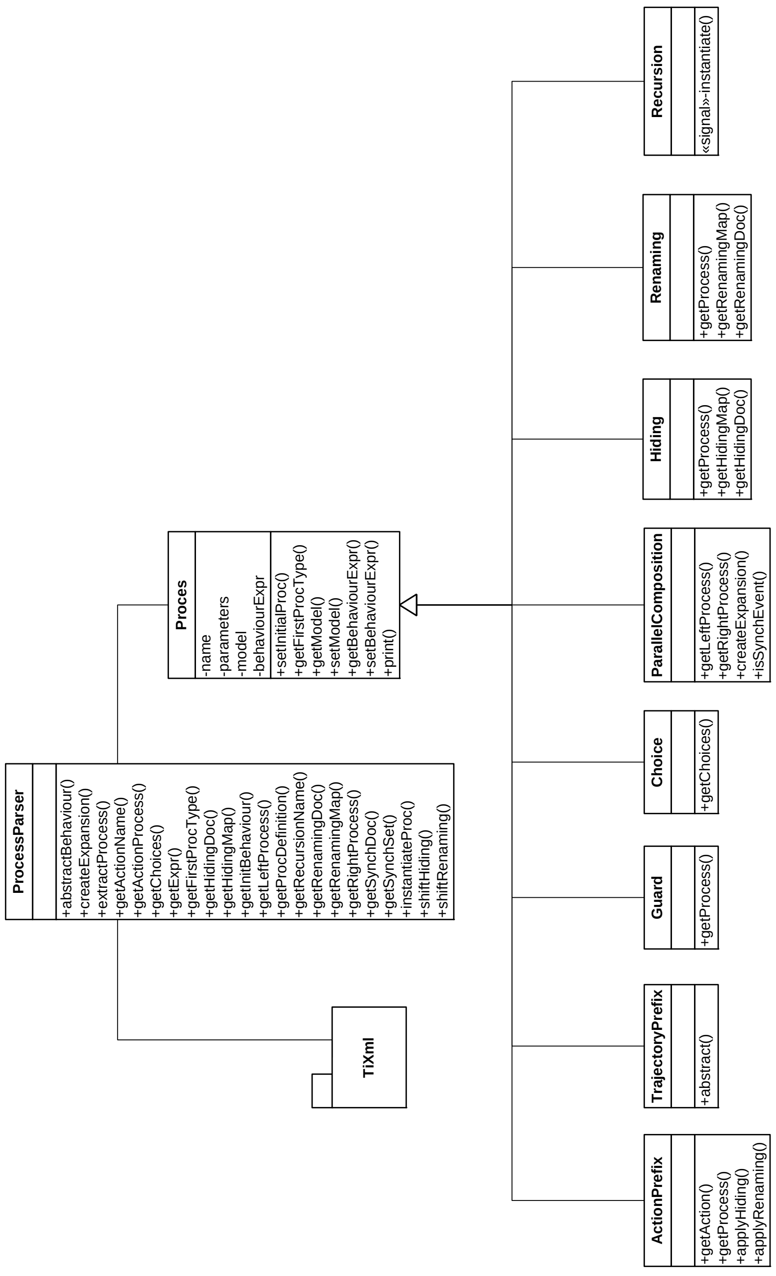


Figure D.4: Process Package.

E Simulating Extended Bouncing Ball

We use the extended bouncing ball from Chapter 2.6. In this example the hybrid system consists of a bouncing ball and controller which are in a parallel composition. The bouncing ball process models a bouncing ball that loses kinetic energy while bouncing, while the controller compensates for the loss of energy and pushes the ball.

Initially the ball loses height, until the floor is reached ($h = 0$) while the controller is idling. At that moment BB and Controller synchronize on bounce. Then the ball gains height, until all energy is used ($v = 0$) while the controller is idling again. At that time they synchronize on push and the controller pushes the ball back to the floor and the whole cycle starts over again.

E.1 BHPC model

$$\begin{aligned}
\mathrm{BB}(h_0, v_0)) &\triangleq [h, v \mid \Phi(h_0, v_0)) \Downarrow h = 0]\,.\mathsf{bounce}(c : [0,1]).\\
&\quad [h, v \mid \Phi(0, -cv) \Downarrow v = 0]\,.\mathsf{push}(v : \mathbb{R}).\mathrm{BB}(h, v)\\
\mathrm{Control}(v_0) &\triangleq \mathsf{idle}.\mathsf{bounce}(c : [0,1]).\\
&\quad \mathsf{idle}.\mathsf{push}\,((1 - c)\,v)\,.\mathrm{Control}\,((1 - c)\,v)\\
\mathrm{System}(h, v) &\triangleq \mathrm{BB}(h_0, v_0)\,||^{v_0}_{\mathsf{push,bounce}}\,\mathrm{Control}(v_0)
\end{aligned}$$

E.2 ASCII model

In which we let a ball drop ($v = 0$) from 10 metres ($h = 10$). The ball is a bouncing ball that loses 30% of its kinetic energy ($c = 0.7$).

```
constants: (c,0.7), (g,9.8)
qualifiers: h,v
initial System(10,0)
proc System(h,v)  ^= BB(h_0,v_0) ||{v_0}{push,bounce} Control(v_0)
proc BB (h_0,v_0) ^= [ h,v | tset phi(h_0,v_0) exit h=0 ] . bounce (c:0,1] .
                        [ h,v | tset phi(0,-c*v) exit v=0 ] . push (v:RR] . BB(h,v)
proc Control(v_0) ^= idle . bounce(c:[0,1]) .
                        idle . push((1-c)*v) . Control((1-c)*v)
tset phi (h_0,v_0) = {h,v:(0,t]->RR |
                        h(0)=h_0, v(0)=v_0, der(h)=v, der(v)=-g , h>=0 }
```

E.3 Trace

The trace for the extended bouncing ball example is

$$(@, \mathsf{bounce}, @, \mathsf{push})^*$$

The engine trace is given in the following (part of the) output file.

```
ENGINE STATE PROCESS

       <behaviourexpression>
           <process>
               <parallelcomposition>
                   <process>
                       <recursion>
                           <name>BB</name>
                           <parameters>
                               <expression>h_0</expression>
                               <expression>v_0</expression>
                           </parameters>
                       </recursion>
                   </process>
                   <process>
                       <recursion>
                           <name>Control</name>
                           <expression>v_0</expression>
                       </recursion>
                   </process>
                   <set>
                       <name>v_0</name>
                       <name>push</name>
```

```
                    <name>bounce</name>
                </set>
            </parallelcomposition>
        </process>
    </behaviourexpression>

EVENT: @ (action)

ENGINE STATE PROCESS

    <behaviourexpression>
        <process>
            <parallelcomposition>
                <process>
                    <actionprefix>
                        <name>bounce</name>
                        <parameters>
                            <expression>c:[0,1]</expression>
                        </parameters>
                        <process>
                            <trajectoryprefix>
                                <tprefixOp>
                                    <symbOp>
                                        <tsetreference>
                                            <name>Phi</name>
                                            <parameters>
                                                <expression>0</expression>
                                                <expression>c*v</expression>
                                            </parameters>
                                            <restrictions />
                                            <exitconditions>v=0</exitconditions>
                                        </tsetreference>
                                    </symbOp>
                                </tprefixOp>
                                <process>
                                    <actionprefix>
                                        <name>push</name>
                                        <parameter>v:RR</parameter>
                                        <process>
                                            <recursion>
                                                <name>BB</name>
                                                <parameters>
                                                    <expression>h</expression>
                                                    <expression>v</expression>
                                                </parameters>
                                            </recursion>
                                        </process>
                                    </actionprefix>
                                </process>
                            </trajectoryprefix>
                        </process>
```

```
                       </actionprefix>
                   </process>
                   <process>
                       <actionprefix>
                           <name>bounce</name>
                           <parameters>
                               <expression>c:[0,1]</expression>
                           </parameters>
                           <process>
                               <trajectoryprefix>
                                   <tprefixOp>
                                       <idle />
                                   </tprefixOp>
                                   <process>
                                       <actionprefix>
                                           <name>push</name>
                                           <parameters>
                                               <expression>(1-c)*v</expression>
                                           </parameters>
                                           <process>
                                               <recursion>
                                                   <name>Control</name>
                                                   <parameters>
                                                       <expression>v_0</expression>
                                                   </parameters>
                                               </recursion>
                                           </process>
                                       </actionprefix>
                                   </process>
                               </trajectoryprefix>
                           </process>
                       </actionprefix>
                   </process>
                   <set>
                       <name>v_0</name>
                       <name>push</name>
                       <name>bounce</name>
                   </set>
               </parallelcomposition>
           </process>
       </behaviourexpression>

EVENT: bounce (action)

ENGINE STATE PROCESS

       <behaviourexpression>
           <process>
               <parallelcomposition>
                   <process>
                       <trajectoryprefix>
```

```
        <tprefixOp>
            <symbOp>
                <tsetreference>
                    <name>Phi</name>
                    <parameters>
                        <expression>0</expression>
                        <expression>c*v</expression>
                    </parameters>
                    <restrictions />
                    <exitconditions>v=0</exitconditions>
                </tsetreference>
            </symbOp>
        </tprefixOp>
        <process>
            <actionprefix>
                <name>push</name>
                <parameter>v:RR</parameter>
                <process>
                    <recursion>
                        <name>BB</name>
                        <parameters>
                            <expression>h</expression>
                            <expression>v</expression>
                        </parameters>
                    </recursion>
                </process>
            </actionprefix>
        </process>
    </trajectoryprefix>
</process>
<process>
    <trajectoryprefix>
        <tprefixOp>
            <idle />
        </tprefixOp>
        <process>
            <actionprefix>
                <name>push</name>
                <parameters>
                    <expression>(1-c)*v</expression>
                </parameters>
                <process>
                    <recursion>
                        <name>Control</name>
                        <parameters>
                            <expression>v_0</expression>
                        </parameters>
                    </recursion>
                </process>
            </actionprefix>
        </process>
```

```
                </trajectoryprefix>
            </process>
            <set>
                <name>v_0</name>
                <name>push</name>
                <name>bounce</name>
            </set>
        </parallelcomposition>
    </process>
</behaviourexpression>

EVENT: @ (action)

ENGINE STATE PROCESS

<behaviourexpression>
    <process>
        <parallelcomposition>
            <process>
                <actionprefix>
                    <name>push</name>
                    <parameter>v:RR</parameter>
                    <process>
                        <recursion>
                            <name>BB</name>
                            <parameters>
                                <expression>h</expression>
                                <expression>v</expression>
                            </parameters>
                        </recursion>
                    </process>
                </actionprefix>
            </process>
            <process>
                <actionprefix>
                    <name>push</name>
                    <parameters>
                        <expression>(1-c)*v</expression>
                    </parameters>
                    <process>
                        <recursion>
                            <name>Control</name>
                            <parameters>
                                <expression>v_0</expression>
                            </parameters>
                        </recursion>
                    </process>
                </actionprefix>
            </process>
            <set>
                <name>v_0</name>
```

```
                    <name>push</name>
                    <name>bounce</name>
                </set>
            </parallelcomposition>
        </process>
    </behaviourexpression>

EVENT: push (action)

ENGINE STATE PROCESS

    <behaviourexpression>
        <process>
            <parallelcomposition>
                <process>
                    <recursion>
                        <name>BB</name>
                        <parameters>
                            <expression>h</expression>
                            <expression>v</expression>
                        </parameters>
                    </recursion>
                </process>
                <process>
                    <recursion>
                        <name>Control</name>
                        <parameters>
                            <expression>v_0</expression>
                        </parameters>
                    </recursion>
                </process>
                <set>
                    <name>v_0</name>
                    <name>push</name>
                    <name>bounce</name>
                </set>
            </parallelcomposition>
        </process>
    </behaviourexpression>

EVENT: @ (action)

ENGINE STATE PROCESS

    <behaviourexpression>
        <process>
            <parallelcomposition>
                <process>
                    <actionprefix>
                        <name>bounce</name>
                        <parameters>
            <expression>c:[0,1]</expression>
        </parameters>
        <process>
            <trajectoryprefix>
                <tprefixOp>
                    <symbOp>
                        <tsetreference>
                            <name>Phi</name>
                            <parameters>
                                <expression>0</expression>
                                <expression>c*v</expression>
                            </parameters>
                            <restrictions />
                            <exitconditions>v=0</exitconditions>
                        </tsetreference>
                    </symbOp>
                </tprefixOp>
                <process>
                    <actionprefix>
                        <name>push</name>
                        <parameter>v:RR</parameter>
                        <process>
                            <recursion>
                                <name>BB</name>
                                <parameters>
                                    <expression>h</expression>
                                    <expression>v</expression>
                                </parameters>
                            </recursion>
                        </process>
                    </actionprefix>
                </process>
            </trajectoryprefix>
        </process>
    </actionprefix>
</process>
<process>
    <actionprefix>
        <name>bounce</name>
        <parameters>
            <expression>c:[0,1]</expression>
        </parameters>
        <process>
            <trajectoryprefix>
                <tprefixOp>
                    <idle />
                </tprefixOp>
                <process>
                    <actionprefix>
                        <name>push</name>
                        <parameters>
```

```
                                        <expression>(1-c)*v</expression>
                                    </parameters>
                                    <process>
                                        <recursion>
                                            <name>Control</name>
                                            <parameters>
                                                <expression>v_0</expression>
                                            </parameters>
                                        </recursion>
                                    </process>
                                </actionprefix>
                            </process>
                        </trajectoryprefix>
                    </process>
                </actionprefix>
            </process>
            <set>
                <name>v_0</name>
                <name>push</name>
                <name>bounce</name>
            </set>
        </parallelcomposition>
    </process>
</behaviourexpression>

⋮
```

F User Manual

The directory “`Discrete Bhave`” contains the discrete engine “`engine.exe`” and a directory for the models, named “`Models`”. The engine comes with a build-in XML parser and no additional parsers are needed. To use the engine, just copy the “`Discrete Bhave`” directory to a convenient location on your computer.

F.1 Start simulation

1. Use “`cmd.exe`” to run the tool from by selecting “`start`” and then “`run`”, type “`cmd`” and press enter.
2. Switch in “`cmd.exe`” to the “`Discrete Bhave`” directory location by using the “`cd`” command. Suppose the path to the “`Discrete Bhave`” directory is “`yourPathTo\DiscreteBhave`”, then “`cd yourPathTo\DiscreteBhave`” should set your prompt to “`yourPathTo\DiscreteBhave>`”.
3. Run the engine from the prompt “`yourPathTo\DiscreteBhave>.\Engine.exe Models\yourModel.xml manual 0`” where “`yourModel.xml`“ is the model of your choice from the “`Models`” directory.

Normally the BHPC model is then loaded and a menu appears. When there are problems with the arguments to run the engine, a message appears that states the correct use of arguments. Problems could occur in case the model could not be found, which is indicated by a “`Model could not be loaded`” message. Finally when a model without initial process is loaded, a “`No initial process could be found`” message is given.

F.2 Event selection

Events can be selected from the menu.

```
------------------------------------
MENU
------------------------------------
0:   STOP SIMULATION
1:   @ (action)
------------------------------------
Choose an option:
------------------------------------
_
```

Is an example of a menu. In each menu the first option (option 0), is the option to stop simulation. All other options are options determined by the engine, like option 1 for abstraction in this menu. By typing the identifier for the option and pressing enter the option is selected and executed. Then the menu is calculated for the new engine state. If there are no transitions to take, simulation is stopped automatically.