Institut für Informatik
der Technischen Universität München

An Approach for Hybrid Modelling and Formal Verification in Focus

Alarico Campetelli

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende(r):	Univ.-Prof. Dr. Uwe Baumgarten
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Univ.-Prof. Dr.-Ing. Matthias Althoff

Die Dissertation wurde am 12.05.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.09.2015 angenommen.

Abstract

Nowadays, safety-critical embedded systems are in use e.g. in vehicles, aircrafts, and medical devices. In these domains, the combination of embedded systems with physical components requires a suitable design that unites continuous time and discrete elaboration in so-called hybrid systems, as provided by the Henzinger's hybrid automaton formalism. The scope of this thesis is the introduction of sampling algorithms and the study of formal verification support for hybrid systems. The theoretical foundation of our approach is the modelling theory FOCUS, which formally specifies distributed and interactive systems. We study the problem of a full continuous simulation of FOCUS hybrid systems in a modelling tool (AutoFOCUS 3) and explore new ideas (with a prototypical implementation) based on the sampling of the continuous time elaborations into discrete time elaborations. In contrast to traditional approaches based on the error minimization, we propose sampling solutions guided by desired requirements/scenarios or by value variation of sampled variables.

System failures may lead to a considerable loss of money due to warranty costs or - in the worst case - even endanger human lives. The use of formal methods for the system definition permits employing formal verification, which offers an exhaustive and automatic verification. This work also explores formal verification methods, such as model checking and Satisfiability Modulo Theories (SMT) solving, as well as their practical application. Our integration of formal verification is performed based on tight coupling of verification properties with model elements, different specification languages for the formulation of properties, as well as visualization and simulation of counterexamples. We provide an implementation of such verification approach in the tool AutoFOCUS 3.

Acknowledgments

Firstly, I would like to give my thanks to my supervisor Professor Manfred Broy, who provided me the chance to be his Ph.D. student without hesitation when I applied with my research proposal. I am grateful for the great support in the development of this thesis and for his continuous mentoring that helped me to be always inspired for my research. Without his advises and discussions with him this thesis would not been realized. I would like to thank my second supervisor Professor Matthias Althoff for the very useful comments and discussions.

I would like to thank my colleagues at the TU Munich. I am particularly grateful to Philipp Neubeck, Georg Hackenberg, Maximilian Irlbeck, Mario Gleirscher, Stefan Kugele, Alexander Gruler and Maximilian Junker for their useful discussions and help for work and thesis issues. I am glad to Xiuna Zhu, Klaus Lochmann, Diego Marmsoler, Vasileios Koutsoumpas and all others chair colleagues for the nice time by coffee breaks. A special thanks goes to Wolfgang Böhm for his excellent feedback on an earlier version of this thesis. Finally, I would like to dedicate this thesis to my wife and my family. The belief and the sacrifice they made for me to support my life choices and my work are ineffable. I thank my wife for always encouraged me and for her care and tender.

Contents

Chapter 1

Introduction

Nowadays, safety-critical embedded systems are in use in vehicles, machines aircraft, and medical devices. In these domains, the combination of embedded systems with physical components determines the necessity of a suitable design that unites continuous time and discrete elaboration. The most important challenge lies in modelling the interaction between the system and the environment including its physical constraints. In fact, software components operate in discrete program steps, while physical components function over continuous time intervals following physical constraints. The term Cyber-Physical Systems (CPSs) is used in literature to indicate such evolutions of embedded systems. In some cases, new specialised methodologies or languages are introduced to deal with CPSs, where the already existing techniques could be suitable to represent them, especially after an extension or adaptation to cover some special domain features, which are not present or not handled for embedded systems. In dealing with CPSs, the main challenge here is to combine two worlds, the physical and the virtual one. Nevertheless, many physical properties can be represented similar to software properties, e.g., in many cases it is possible to switch from continuous time to discrete time representation without losing the essential properties of the represented system [69]. In several industrial projects we verified that, speaking about the system architecture and properties on a certain abstraction level, did not reveal differences between physical signals and discrete elaborations from the software components [32]. Thus, until we are considering systems modelled with FOCUS, which are a logical representation of the systems, we can benefit from using the software development models and methods. An important formalism for describing CPSs from the perspective of computer scientists is known as hybrid automaton [68]. Systems that combine discrete/continuous time and data are usually known as hybrid systems [4].

Software and hardware in hybrid systems are complex: these systems consist of a high number of modules or programs, where the software part is in the magnitude of several millions lines of code. System failures may lead to a considerable loss of money due to warranty costs or – in the worst case – even endanger human lives. This motivates the need for well-defined formal modelling theories, languages, and tools, which help to improve system quality. A formal modelling theory provides important concepts for development such as different conceptional levels of the system, as for instance views and different levels of granularity of the system under development. Moreover, a suitable modelling theory for these systems helps in their development, maintenance, simulation, and verification. Correct behaviour of the modelled systems is fundamental. It can be guaranteed through analysis techniques, mostly in form of formal verification (cf. [43]). Their comprehensive use permits the construction of more reliable systems, even though system complexity. The use of formal methods for the system definition allows for formal verification techniques, which offer exhaustive and automatic checks. Model checking is a successful formal verification technique, which builds a formal model of a system and checks whether it satisfies a desired property, proving the correctness of the system under development.

We concentrate our work mainly in the domain of model-based development for embedded systems and CPSs. Our contribution starts with the extension of the modelling theory FOCUS [22] for embedded systems to the domain of CPSs. Based on this theoretical foundation we address the simulation of such systems using a discretization of time, introducing the sampling of the involved continuous time variables. This way the state space of the CPSs is reduced to a discrete set. Since CPSs often can be found in safety-critical domains, they require rigorous checking techniques. We contribute to handling these requirements with formal verification techniques, not only for the introduced hybrid systems, but also for tackling the usability of such analysis techniques in supporting tools. The foundations of our work are evaluated already successfully by means of case studies from the embedded system domain, as for instance in the automotive and automation branches; we propose to extend its general ideas for the development of CPSs.

1.1 Motivation

CPSs are composed of discrete time components, as for instance software parts or switches, and continuous ones such as sensors or mechanical parts. Usually discrete transitions are instantaneous state changes, meanwhile elab-

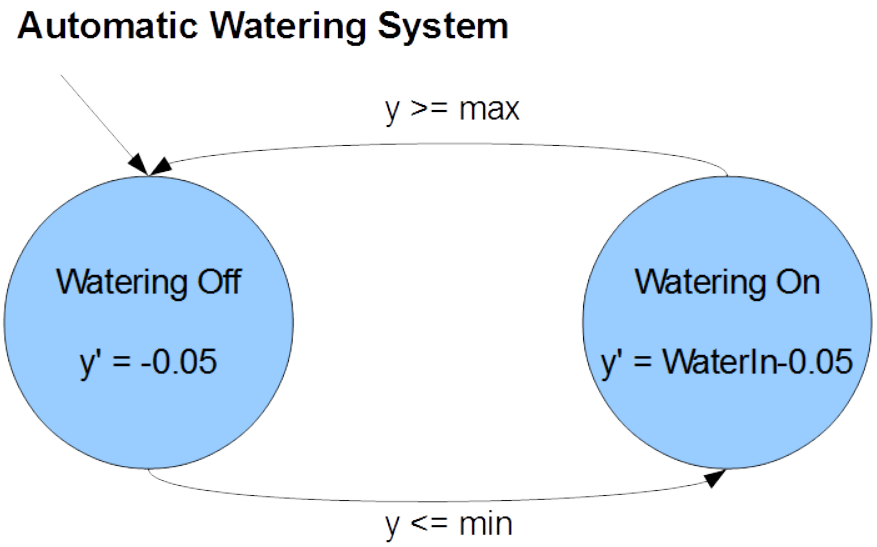

Figure 1.1: Hybrid system for the control of a simple watering system.

orations in continuous time changes the system state according to differential equations. Moreover, in CPSs embedded computers perform the control of the physical processes. The combination of digital and analog as well as software and physical elements involves several fields of study, principally electrical engineering, mechanical engineering and computer science. In order to model software intensive CPSs are commonly used hybrid systems. Figure 1.1 shows a simple hybrid system for the control of an automatic watering system. This automaton shall prevent that the tank level (y) from emptying or filling up above a certain level, also considering a constant water dispersion from the tank to the watering system.

Modelling Designers of embedded systems and CSPs are increasing their interest for integrated model-based development approaches. Model-based development advocates the pervasive use of models throughout the entire development process. Early defined models capture requirements on the system, and are subsequently transformed and enriched until an implementation is completed. Models exist on different levels of detail and formalisation, and are used for different purposes by different stakeholders during the process phases of the system life-cycle. A precise and formal modelling theory abstracts from certain details, from each involved discipline. On a logical level,

the system has already the interfaces between the subcomponents and their logical behaviour, but excluding the implementation details, which are then explicitly defined in the technical level. The complexity of the development is reduced using these separated designs and verification at early design stages, if possible missing knowledge about the complete hardware implementation.

Hybrid systems are natural candidates for model-based development to support the different physical and software requirements, the different nature of the components as well as other interdisciplinary issues. The abstraction of the models plays an important role in the design to hide details, manage complexity and show domain-specific views. Embedded control systems need a combination of traditional automata based models for the discrete control and differential equations based models for the continuous computation of physical components or the environmental conditions and elements. We want to tackle with this work the modelling of CPSs with a suitable theory and with a concrete tool support. FOCUS [22] is a modelling theory for the formal specification of distributed, discrete-event systems. It forms the foundation for our work and is extended to support the modelling of hybrid systems. FOCUS components are defined in a hierarchical and interconnected net, where each component as a typed i/o interface. The internal behaviour of each component can be implemented using different formalisms, for instance functional specifications or finite state machines. The FOCUS theory is supported and implemented in the modelling tool AutoFOCUS 3[1] [73]. Our work starts with the continuous functional specifications as defined by Scholz and Müller [94] and is extended to provide a wider framework for real-time and hybrid systems. In order to support these features, the authors modified the semantics of the interface functions between components (streams) introducing dense streams.

Sampling The real-time simulation and elaboration of hybrid systems is also addressed. We study not only the formalization of the modelling theory for hybrid systems, but also aspects that are related to tool support and execution of them. Our models represent CPSs with a focus on software behaviour. In a software tool, a full continuous simulation for hybrid models is without approximation not realizable and the final hardware of the system under development may be different to the hardware of the supporting tool. For these reasons, we propose sampling techniques to transform the continuous time into discrete steps. There are already many works that address this subject as a central point of the control and simulation of hybrid systems

[1] `http://autofocus.in.tum.de`

[10, 95, 103, 90]. We study an approach that fits at best with our extension to the FOCUS theory and for an implementation in the tool AutoFOCUS 3.

In mathematics, the term sampling is defined as the transformation of an analogue signal into a digital one. Our ideas are mainly based on the work of Petreczky et al. [103], where a static sampling over continuous time is presented. It is static because the discrete time step remains constant, also called "equidistant sampling". Instead, we introduce two solutions, including a prototypical implementation, based on the dynamic sampling approach. The adjective "dynamic" refers to the discrete time step, which is variable during the simulation. In general dynamic time step modifications can produce a more flexible simulation as opposed to static sampling. Traditional dynamic sampling approaches vary the period according to an estimation of the error of the approximated differential equations. These approaches can be too time consuming especially when applied to large systems of differential equations. We want to overcome this problem by adopting a high approximation quality only for specific regions of the differential equations. At the same time, we aim at an intuitive way to define such regions of interest. A lower approximation can be accepted in regions that have a lower interest in the simulation. Our algorithms make the step shorter or longer with two different solutions: depending on the values reached by the continuous variables or the slope between the previous and the actual values of the continuous variables. We claim that such variations can produce a better simulation, because we can select important value intervals for each variable from the requirements, and establish a corresponding length for the sampling period. On the other hand analysing the slope of the variables and use a shorter period when the slope is steep better adapts to the variation of the variable, producing more values in the same time, increasing the precision of the simulation, in a certain sense. Overall, our algorithms in some scenarios can be faster than the traditional solutions.

Verification Hybrid systems often are implemented in highly safety-critical systems such as control systems for vehicles, machines aircraft, and medical devices. Verification is then a crucial aspect of the development, but also challenging due to its complexity. One prominent problem is that verification cannot easily handle the state space dimension of hybrid systems, whose size is determined also by continuous variables. Representations of hybrid systems result in infinite state spaces and model checking solutions may need an abstraction or a symbolic representation in order to deal with them, since the complexity of the verification process. Approaches based on deductive formal verification, that employ a deductive proof system developed for the

used logic, may overcome some limitations [104]. We address formal verification in our integration of model checking in the modelling tool AutoFocus 3 (cf. [42]). In this integration, also the usability and the integration in support tools are important aspects, because the effectiveness of powerful verification solutions may be invalidated by a suboptimal integration, or by the high skills required to manage the tools and the verification properties. We believe that system design should guarantee integrated and easy to use formal verification facilities. We study and perform preliminary tests of an extension of such techniques for the introduced hybrid components, using supporting verification based on Satisfiability Modulo Theories (SMT) [13].

1.2 Subject

As explained in the introduction, we describe an attempt to define formal modelling of CPSs, based on the hybrid automaton paradigm. Considering their simulation in a modelling tool, we introduce sampling of the continuous time variables. Finally, we provide an initial formal verification support. This work starts with the definition of a suitable modelling theory, considering the problems and limitations of actual approaches; we describe the extension of a supporting tool for the modelling, study the complexity and the simulation issues and introduce an integrated user-friendly formal verification, with particular focus on the methods for high level and intuitive definition of properties.

The proposed modelling theory is not defined from scratch; instead, it is based on the Focus theory, a model-based method for the specification of distributed embedded systems implemented in the support tool AutoFocus 3. The basic structure of a Focus model is a network of components communicating through input and output channels between them. An initial approach for supporting real-time systems in Focus was already conducted by Scholz und Müller, through functional specifications. Our extension of Focus to support continuous dynamics is inspired by this work. We encapsulate the hybrid automata model as internal implementation of Focus components and we introduce then a new approach for the dynamic sampling of differential equations [27, 30].

One goal is to promote the introduction of model-based development in practice. Therefore, a tool support for the design of CPSs is fundamental. AutoFocus 3 permits the modelling and analysis of distributed and reactive systems. We propose a prototypical implementation of the hybrid systems artefacts in AutoFocus 3. We analyse simulation issues of the designed

models, consider complexity and provide an implementation. We also introduce sampling algorithms for the execution of the continuous dynamics, in a discrete environment as in AutoFocus 3.

Conceiving the safety-critical domain of CPSs, many efforts have been made to provide a suitable formal verification. The formal verification approach is addressed studying the possibility of model checking, SMT solving, as well as the combination of both. The implementation of analysis techniques for discrete systems in AutoFocus 3, such as model checking, debugging support and specification of properties (assertion languages [14], templates and specification patterns [53]), is then extended towards a verification environment for hybrid systems. Therefore, our contribution is not limited to the support of formal verification, but also improves the integration and usability in support tools.

1.3 Outline

The thesis has the following structure: Chapter 2 presents the modelling foundations, which motivate the following design decisions. In Chapter 3, the modelling elements used for the definition of hybrid components are proposed. Chapter 4 introduces the simulation and discretisation of hybrid components with sampling techniques. The formal verification capabilities for Focus components are introduced in Chapter 5. In Chapter 6 a case study representing a train velocity control system is presented to show our contributions. Finally, in Chapter 7, we give a summary of the concepts and ideas, and we briefly discussed possible future work directions.

Chapter 2

Fundamentals and Design Methodology

In this Chapter the fundamental notions are explained, which will be used for the definition of our work in Chapter 3. The design method introduced by FOCUS is modular and allows for abstraction levels by stepwise refinement. Based on the models defined in FOCUS we propose our solution and introduce in this chapter the fundamentals of FOCUS and hybrid automata theories. To represent a hybrid component of a system at the abstraction level where the continuous and discrete calculations are modelled, we construct our models inspired to the hybrid automata from Henzinger [68]. Our contribution is inspired by our system development methodology, which is presented in the last section.

2.1 System Classification

In this work, we are focused on the design of embedded systems with continuous dynamic, for instance systems that interact with physical processes. In order to present our application domain, we briefly explain the characteristics that determine whether a system is discrete, continuous or hybrid.

Characteristics For software systems their internal state, inputs and outputs change over time according to a precise dynamic. Following the classification in [91], we briefly describe system characteristics that affect this evolution and dynamic. Based on the nature of their state there are the following classes:

- *Continuous*, if the system state is defined over values in $\mathbb{R}^n$ ($n \in \mathbb{N}$), therefore a system state is denoted as $x \in \mathbb{R}^n$.
- *Discrete*, if the system states are defined in a countable set or finite set, where we denote a state $q \in M = \{q_1, \ldots, q_i, \ldots\}$.
- *Hybrid*, if the system state is partially defined with values in $\mathbb{R}^n$ and in a countable set.

The evolution of a system is usually determined following time or driven by events. Thus, a system may also be classified using the time parameter over which the states evolve:

- *Continuous time*, if the time parameter is defined over the set of real number; usually an ordinary differential equation describes the evolution of the states in this case.
- *Discrete time*, if the time or set of times is a subset of $\mathbb{N}$, usually a difference equation describes the evolution of the state. The state space with discrete time is defined as $x_{k+1} = Ax_k$, where $k \in \mathbb{N}$ denotes the discrete time.
- *Hybrid time*, if the time is continuous but there are also discrete transition or evolution.

Equations that involve dependent variables and their derivatives with respect to the independent variables are called *differential equations*. Parts of physical and engineering systems are often modelled trough differential equations. Differential equations in which all the derivatives are respect to only one independent variable are called *ordinary differential equations (ODEs)*. ODEs describe usually the evolution of the state of continuous time systems. A differential equation is called *linear* if: (1) any dependent variables and their derivatives appear to the first power; (2) products of dependent variable are not allowed. In fact, to determine if a differential equation is linear are used only the function and its derivatives. Differential equations are *nonlinear* if they do not satisfy the definition of a *linear*.

In this work we concentrate on systems with an hybrid state space, there is a further classification for the equations used:

- *Linear hybrid systems*, if the evolution of the system is determined by a linear differential equation. Usually, linear differential equations are involved for the dynamics of control systems in mathematical design.

- *Nonlinear hybrid systems*, if a nonlinear differential equation determine the evolution of the system. In this category of systems, nonlinear ODEs delineate the continuous state space and nonlinear constraints the discrete changes.

In linear hybrid systems the discrete transitions are restricted by linear inequalities over the source and target values of the variables, the continuous flows are restricted by linear inequalities over the possible values of the variables during the flow, and the possible derivatives of the variables during the flow are restricted by linear inequalities on the derivatives. In each discrete state linear constraints on the first derivatives, determine the behaviour of all variables. Hybrid automata and its linear variant are explained in Section 2.3, meanwhile a definition of the linear hybrid system based on the approach in [4] is given in Appendix A.

2.2 Introduction to Focus

We concisely present the foundations of FOCUS [22], as a discrete modelling theory. We can model a system in FOCUS, beginning at an abstract requirement specification, which can be formalised as either a trace or a functional specification. These specifications represent the foundation for the following phase, which is a concrete implementation description (FOCUS models are structured as hierarchical components connected with input and output channels). It provides a formal semantics for define relationships between the different descriptions. We shortly explain these elements, and then in the next sections we give formal definitions for discrete systems, trace specifications and functional specifications.

Streams Infinite and finite sequences of elements from given sets are called streams. Streams can consist of actions (called traces) or of messages (called communication histories).

Components Basically, a system is composed of a number of subsystems (called its components). The composed system is a component itself and can in turn be a part of a larger system. A component is specified by its interface to communicate with its environment and (optionally) an encapsulated state.

2.2.1 Discrete Components

Streams are a central concept in FOCUS theory. Depending on the considered specification, streams consist of *actions* in trace specifications and of *messages* in functional specifications and programs.

Definition 1 (Stream). *Let S be the set element, we denote with S^ω the set of streams over S. It results from the union of the set of finite and infinite sequences $S^\omega = S^* \cup S^\infty$.*

Over the streams, the following operations and relations are defined, where s, t, and u are streams and a, b, and c are elements of S:

- $\langle\rangle$ is the empty stream.
- $\langle s_1, \ldots, s_n\rangle$ indicates the stream containing the elements $s_1, \ldots, s_n$ with $s_i \in S$.
- $ft(s)$ returns the first element of s, if s is not empty, otherwise it returns $\perp$, that represents an undefined result.
- $rt(s)$ returns the stream in which the first element of s is deleted.
- $a\&s$ indicates the stream where a is prefixed to s. Thus, if a is defined, that is $a \neq \perp$, we have $ft(a\&s) = a$ and $rt(a\&s) = s$.
- $s \circ t$ denotes the concatenation of s and t. $s \circ t$ returns s, if s is infinite. Abuses of the notation for streams of length 1 are $a \circ s$ for $a\&s$ and $a \circ b$ for $a\&b\&\langle\rangle$.
- $s \sqsubseteq t$ indicates that s is a *prefix* of t. We can formal express it by $\exists u : s \circ u = t$.
- a *in* s produces *true* exactly if a occurs in s.
- $\#s$ returns the length of s. The result may also be "infinite": ∞
- $a©s$ denotes the substream of s that contains only the a-items. It is a filter operator, for example, $a©\langle a, b, a, c, \rangle = \langle a, a\rangle$. The first operand may also be generalised to a *set* of items.

Transition systems, or *automata*, in terms of states and state transitions, can model a distributed system. A single state represents a snapshot of the system, and through the actions the system processes from a state to the next, determining an evolution of the system from one snapshot to the following.

Definition 2 (Transition system). *A transition system is a tuple* $(Act, State, \rightarrow, Init)$ *where:*

Act *is a set of actions,*

$State$ *is a set of states,*

$\rightarrow \subseteq State \times Act \times State$ *is the transition relation,*

$Init \subseteq State$ *is the set of initial states.*

The transition $(\sigma, a, \sigma') \in \rightarrow$ is also written as $\sigma \xrightarrow{a} \sigma'$. The executions of a transition system are sequences of states and actions:

$\sigma_0 \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \sigma_2 \ldots$

where σ_0 is an initial state and $\sigma_i \xrightarrow{a_{i+1}} \sigma_{i+1}$ is true for all i.

The semantics transition system is described by its executions represented sequences of states and actions, while the interface of a system is described by its input and output behaviour. This behaviour is the component *interface*, which is defined by input and output actions and a predicate on input/output traces. The set of input actions (I) must be disjoint from the set of output actions (O) for distinguishing inputs from outputs in a trace. Predicates are described by trace logic and/or transition systems.

Definition 3 (Component). *A component is a tuple* (I, O, C) *with*

$I \cap O = \emptyset$ *and* $C : (I \cup O)^\omega \rightarrow \{true, false\}$.

The function C returns *true* if the input/output combination may be generated from the component, otherwise it returns $false$. I represents the set of input and O the set of outputs of the components. A system is composed of a set of components (I_1, O_1, C_1), ..., (I_n, O_n, C_n) where $O_i \cap O_j = \emptyset$ for $i \neq j$. The condition of disjointness for output actions is necessary, in order to guarantee unique output ports for components. In case, the specification represents a closed system, the environment is modelled from at least one of these components; otherwise, it is modelled as input port(s) to one or more components. If an output port of one component is an input port of another one, we say that the two components are *connected*.

Definition 4 (Component composition). *Components composition is defined by the following predicate:*

$Sys : (I_1 \cup \cdots \cup I_n \cup O_1 \cup \cdots \cup O_n)^\omega \rightarrow \{true, false\}$ *defined by*

$Sys(t) \Leftrightarrow \forall i \in \{1, \ldots, n\} : C_i((I_i \cup O_i) \copyright t)$

Sys is the global specification of a system constructed by n components.

2.2.2 Channel Types

We assign a data type to each channel by a function $t : c \to T$, where c is a channel and T is a set of types $\tau \in T$ and each type represents a set of data elements.

2.2.3 Trace Specification

A distributed system can be described by a trace specification, as the set of all the runs of this system by sequences of *actions*, which are elements of a given set *Act*. The sequences are named as *traces*. These specifications are suitable for defining requirements. Trace specifications are constructed over actions, which represent basic activities in a system. Activities are atomic and instantaneous as, for instance, the exchange of a message between system components or the response to a user command. A *trace* specification comprises a record of a run of the system. Modern systems have naturally also infinite runs, so we consider infinite and finite traces.

Definition 5 (Trace). *Let Act be a set of actions. A trace is a stream of actions defined over the set Act:*

$Act^{\omega} = Act^{*} \cup Act^{\infty}$

where Act^{*} are the finite traces and Act^{∞} the infinite ones.

2.2.4 Functional Specification

Functional specifications are less abstract compared to trace specifications presented in the previous section. The systems are modelled following the paradigm of communicating system components, in a network connected by direct channels. The connection is made between an output port to an input port. In this component network, each component receives input messages, which are then processed in some manner, and produces output messages. In the structure there is no control on what is received as input, instead there is a full control over the produced output. The input and output ports of a component are channels and are connected to its environment. A component must have at least one output port and it must have at least one input port only if the action set I is not empty. In the component specification, every input (output) action must be associated with exactly one input (output) port. Therefore, the sets I and O of input and output actions have to be partitioned in respectively: $I_1, \ldots, I_p,\ p \geq 0$ and $O_1, \ldots, O_q,\ q > 0$.

Definition 6 (Functional Specification). *A component can be represented through a functional specification with the following signature:*

$$I_1^\omega \times \cdots \times I_p^\omega \to O_1^\omega \times \cdots \times O_q^\omega$$

This definition asserts the *syntactic interface* of a system component, listening the number of input and output ports and the type of messages that can be sent over each port. The behaviour of a system described by a functional specification is represented as a network of functions. The single function receive as input infinite streams of messages and produce in output infinite streams of messages. The denotational semantics maps the specifications to sets of stream processing functions or sets of traces.

2.3 Hybrid Automata

Automatic systems have a growing and important role in the control of real-life processes, such as in cars, aircrafts, elevators, etc. The environment interacts with these systems with a continuous exchange of inputs and outputs. In many cases interaction and communication occurs in real-time.

Hybrid systems are formed by discrete software, hardware components, physical components and interact with an analog environment. For the specification and modelling of hybrid systems, Henzinger introduced hybrid automata [68] that can be considered as a generalisation of timed automata. We combine in Chapter 3 the FOCUS theory introduced in 2.2 with hybrid automata defined in [10]. The formal definitions used to specify hybrid systems and automata are reported from [4]. So far, this paradigm does not support composition and abstraction of systems, because it has no input or output. The following definition is taken from [91].

Definition 7 (Hybrid automaton). *A hybrid automaton H is a tuple $H = (Q, X, f, I, Dom, E, G, R)$ constituted by eight components:*

- *A set $Q = \{q_1, q_2, \dots\}$ of discrete states.*
- *A set $X = \mathbb{R}^n$ of continuous states.*
- *A vector field function $f : Q \times X \to \mathbb{R}^n$.*
- *A set of initial states $I \subseteq Q \times X$.*
- *A domain function $Dom : Q \to \mathcal{P}(X)$.*

- *A set of edges* $E \subseteq Q \times Q$.
- *A guard condition function* $G : E \to \mathcal{P}(X)$.
- *A reset map function* $R : E \times X \to \mathcal{P}(X)$.

We represent hybrid automata as directed graphs with vertices Q and edges E. A set of initial states $\{x \in X | (q, x) \in I\}$, a vector field $f(q, \cdot) : \mathbb{R}^n \to \mathbb{R}^n$ and a domain $Dom(q) \subseteq \mathbb{R}^n$ are associated with each element $q \in Q$. Each edge $(q, q') \in E$ has a guard $G(q, q') \subseteq \mathbb{R}^n$ and a reset function $R((q, q'), \cdot) \subseteq \mathbb{R}^n$.

A set of continuous states is assigned to each discrete state with the function Dom. When the automaton is in one of these states, there is the continuous evolution in the state. A state of the automaton is referred as a pair $(q, x) \in Q \times X$. The evolution in the continuous state space from an initial state (q_0, x_0) is determined by the differential equation

$$\dot{x} = f(q_0, x)$$
$$x(0) = x_0$$

and the discrete state q is constant

$$q(t) = q_0.$$

In the time interval when x is still in $Dom(q_0)$ the continuous evolution can be executed. When the guard $G(q_0, q_1) \subseteq \mathbb{R}^n$ of some edge $(q_0, q_1) \in E$ is reached from the continuous state x there may be a discrete transition to the state q_1. In this case the reset map function is used to set some value in $R((q_0, q_1), x) \subseteq \mathbb{R}^n$ to perform a reset of the continuous evolution.

2.3.1 Example

We report an example from [91], which is a simple tank system, composed of two water tanks that leak with a constant rate. To add water at a constant rate there is a hose that can switch instantaneously between the two tanks. The amount of water in a tank is denoted as x_i and $v_i > 0$ is the rate of water that leaks out of a tank, with $i = 1, 2$. Water is added to the system with a flow of w. Assuming that the initial levels are above r_1 and r_2, the scope is to keep the water levels of the two tanks above r_1 and r_2. For that there is a controller that changes the tank to be filled to the first if $x_1 \leq r_1$ and to the second if $x_2 \leq r_2$. Now we define the automaton components:

- $Q = \{q_1, q_2\}$, a discrete state for the active tank in the filling operation;
- $X = \mathbb{R}^2$, there are two continuous states derived from the water level for each tank;
- We assign the vector field in each discrete state, considering that only one tank will be filled at a time and the other one will be losing water:

$$f(q_1, x) = \begin{bmatrix} w - v_1 \\ -v_2 \end{bmatrix} \quad f(q_2, x) = \begin{bmatrix} -v_1 \\ w - v_2 \end{bmatrix}$$

- The initial state is $I = \{q_1, q_2\} \times \{x \in \mathbb{R}^2 | x_1 \geq r_1 \wedge x_2 \geq r_2\}$, that means starting with water levels above the delimitation marks;
- The domain is denoted with $Dom(q_1) = \{x \in \mathbb{R}^2 | x_2 \geq r_2\}$ and $Dom(q_2) = \{x \in \mathbb{R}^2 | x_1 \geq r_1\}$, that is the current tank is filled until the water level in the other tank sinks below the volume mark;
- The edges are $E = \{(q_1, q_2), (q_2, q_1)\}$ it is possible to switch the filling tank.
- The state guards are $G(q_1, q_2) = \{x \in \mathbb{R}^2 | x_2 \leq r_2\}$ and $G(q_2, q_1) = \{x \in \mathbb{R}^2 | x_1 \leq r_1\}$, so the tank is switched when the water level of the other tank reach the mark;
- $R((q_1, q_2), x) = R((q_2, q_1), x) = \{x\}$ the discrete transitions do not change the continuous state.

The defined hybrid automaton is graphically represented in Figure 2.1.

2.3.2 Linear Hybrid Automaton

A linear differential equation is an equation of the first degree only in respect to the dependent variable or variables and their derivatives. An equation of the first degree has in each term either a constant or the product of a constant and the first power of a single variable. The evolution of a hybrid automaton in the state $q \in Q$ is determined from the differential equation $\dot{x} = f(q, x)$.

A function $f : \mathbb{R} \times ... \times \mathbb{R} \to \mathbb{R}$ is linear iff:

$$f(\alpha x_1 + \beta y_1, ..., \alpha x_n + \beta y_n) = \alpha f(x_1, ..., x_n) + \beta f(y_1, ..., y_n) \quad (\alpha, \beta \in \mathbb{R})$$

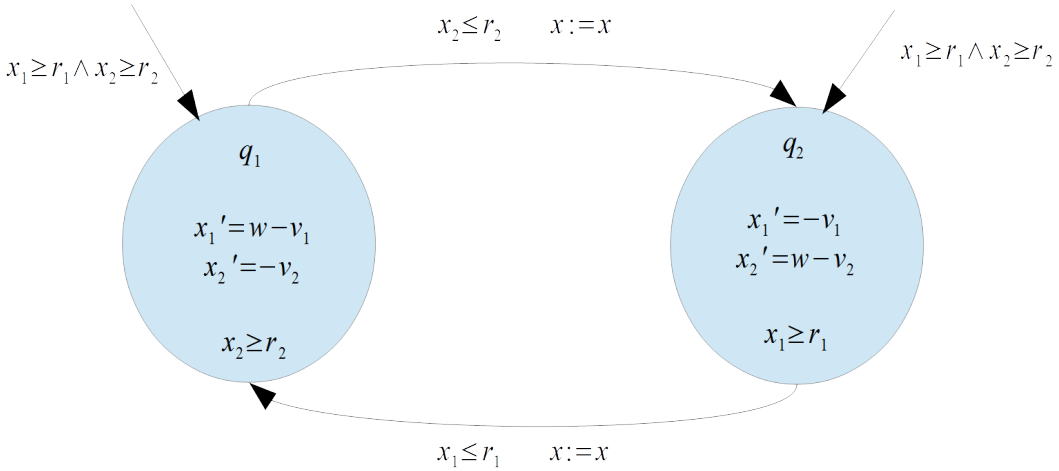

Figure 2.1: The hybrid automaton of the water tank controller [91].

A *linear term* is a linear combination of state variables with integer coefficients. A *linear formula* is a boolean combination of inequalities between linear terms. Hybrid automata can be defined as linear systems expressing vector field, guard condition, reset and domain functions as linear terms and linear formulas.

Definition 8 (Linear hybrid automaton). *A hybrid automata* $H = (Q, X, f, I, Dom, E, G, R)$ *is linear if: (1) for all the states* $q \in Q$*, the vector field* $f(q, \cdot)$ *is expressed by a set of linear differential equations; (2) for all states* $q \in Q$ *the domain function is defined as a linear formula over* X*; (3) for all the edges* $(q, q') \in Q$ *the guard condition function is defined as a linear formula over* X*; (4) for all the edges* $(q, q') \in Q$ *the reset function* $R((q, q'), \cdot) \subseteq \mathbb{R}^n$ *is defined in form of a linear formula over* X*,* $x := \alpha_x | x \in X$*, where the* α_x *have to be linear terms.*

As we explained in Section 2.1, systems can be classified as linear or nonlinear. Tools for analysis and design of nonlinear systems are limited to special classes. Compared to the number of methods available for linear systems, there are only few for nonlinear ones. Model checkers can verify automatically and effectively linear hybrid systems, meanwhile for nonlinear systems an automatic verification is hard to obtain. The first solution (when possible) to handle nonlinear systems is commonly to reduce them to linear systems. We provide related work about this transformation in Section 3.5.2.

In the introduction, we mentioned the importance of simulation in the design of hybrid systems. In Chapter 4, we propose dynamic sampling approaches for hybrid systems and their numerical solutions to be executed on a computer. These solutions are well understood on linear hybrid systems. Therefore, these are the main reasons, why we concentrate our methods on linear systems.

2.4 System Development Methodology

The development of systems usually applies different levels of abstraction, to manage and reduce the overall complexity. A model-based development environment provides model types to design different aspects of the system, e. g., for embedded systems its functional behaviour, its software architecture and its hardware architecture. These models are used to automatically generate the implementation code of the software system w.r.t. its hardware and execution environment. In [111] a framework is described where models are defined that are used in the development process of embedded systems, with their seamless integration at different stages. The authors considered two different dimensions: the levels of granularity and the software development views. The views support a stepwise refinement of information from black box functionality to the realisation on hardware. The same concepts are presented and extended in research projects SPES and its successor SPES XT[1] [108], sponsored by the German ministry of research. In these projects a large consortium of industrial and academic partners defined a method for model-based development of embedded systems, with the scope to challenge their develop model based engineering methods and that should guarantee verification, validation, handling of contractors, quality assurance, definition of system modes and systematic reuse. The SPES development framework, also called the SPES matrix, provides modelling concepts to design different aspects of the system, e.g. for embedded systems its functional behaviour, its software architecture and its hardware architecture. The first dimension of the SPES matrix defines four viewpoints: requirements, functional, logical, and technical. The viewpoints compose the SPES methodology for the definition of system architecture. The order already suggests phases of the design process, where the artefacts of one viewpoint serve as input for the artefacts of the next viewpoint. However, the SPES framework does not prescribe a fixed development process: an evolution or change of a model in a viewpoint, it may require modifications to models in others viewpoints, in

[1] `http://spes2020.informatik.tu-muenchen.de`

order to maintain the consistency between system representations. The second dimension of the matrix is the level of granularity of the artefacts system under development. We report a representation of the SPES matrix in Figure 2.2. The development of a system under development (SuD in Figure 2.2)

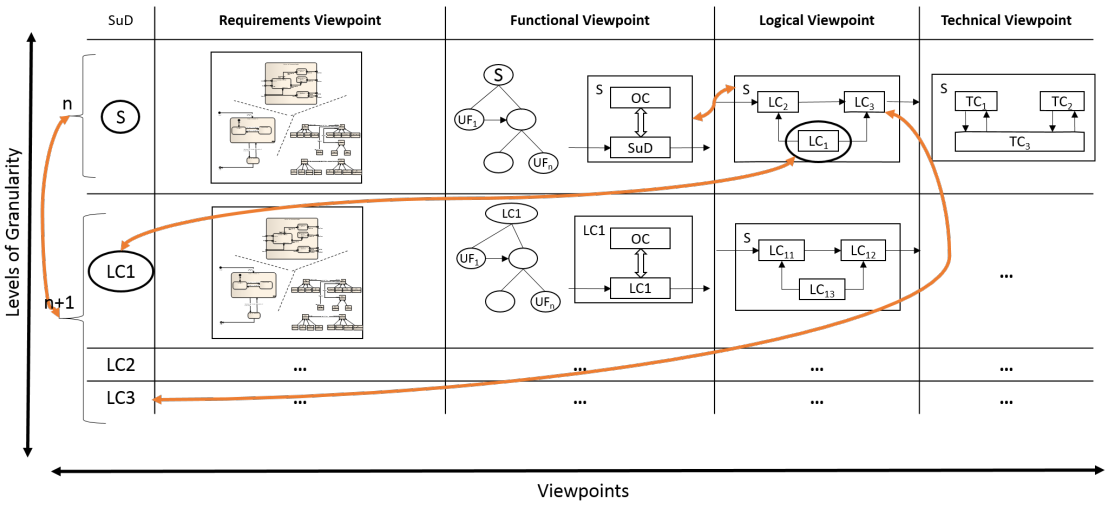

Figure 2.2: Two-dimensional abstraction: level of granularity and software development views, presented in SPES XT project. In the depicted artefacts, SuD indicates the system under development, S is a proper subset of the system, OC means operational context, UF means user function, LC means logical compo-nent and TC means technical component.

is usually starts with its requirements, which can have several representations (e.g. textual or graphical) and levels of formality (e.g. formal or informal). The requirements artefacts can be represented for instance by use case diagrams in terms of message sequence charts representing scenario descriptions. The functional viewpoint describes the system functions in a structured and hierarchical way looking at the system as a black box modelling the interface behaviour of the system. The functional viewpoint can be directly derived from the system requirements. The artefacts, in this viewpoint, describe the system by a set of functions, which has to be realized by the system implementation. From the functional view the logical view is derived, where the architecture and hierarchy of the components and subcomponents is defined. The logical behaviour is observable through the signal/message flow between the components. Functions are seamlessly traced with an association to the correspondent subsystems in the logical models that implement them. The artefacts in the logical viewpoint sketch a first structuration of the system

into a hierarchy of interconnected units, called components, with defined i/o interfaces and behaviour. Functions are seamlessly traced with an association to the correspondent subsystems in the logical models that implement them. The behaviour of the system is observable in different ways, as for instance through observations of the signal/message flow between the components. Therefore, it is possible to systematically analyse the behaviour of the system and verify requirements and system properties. Using system simulation techniques at this stage of the development a modification to correct an erroneous execution of the system will be less complicated and require lower costs, since the deployed hardware and software are not yet involved. The early definition and design of i/o interfaces and a late separation between disciplines permit the creation of integrated impact models. Finally, the technical view contains the implementation, hardware, and planning of the system. We consider the viewpoints as development stages with a seamless integration. For instance from the functional view the logical view is derived. Our work focuses mainly on the domain of the logical representations of the system and establishes an extension of the logical view (according to the SPES matrix) for CPSs. From the system models, implementation code for the final hardware and execution environment can be generated automatically.

There are different concepts and methodologies to describe and implement the development process of software and hardware systems. Model-based development advocates the use of formal and semiformal models throughout the entire development process. During the development life cycle, models are used at different levels of detail, and formalisation, informally starting from the requirements of the system, which is subsequently transformed and enriched until an implementation is completed.

In Figure 2.3, the relation between tools, modelling languages, and modelling paradigms is shown. In this picture, a modelling paradigm appears in different forms or dialects in different languages. In addition, many dialects of modelling languages can be understood and classified in terms of the modelling paradigms they are based on. We classified well-known modelling paradigms in [28], in order to provide an overview that may be used by engineers for modelling tasks.

2.4.1 Waterfall model design process

We show a typical Waterfall model design process for software systems in Figure 2.4 [116]. Requirements analysis is the first step, and describes the major characteristics and system properties to be developed. The second phase is the design, which is focused on the development of a high-level

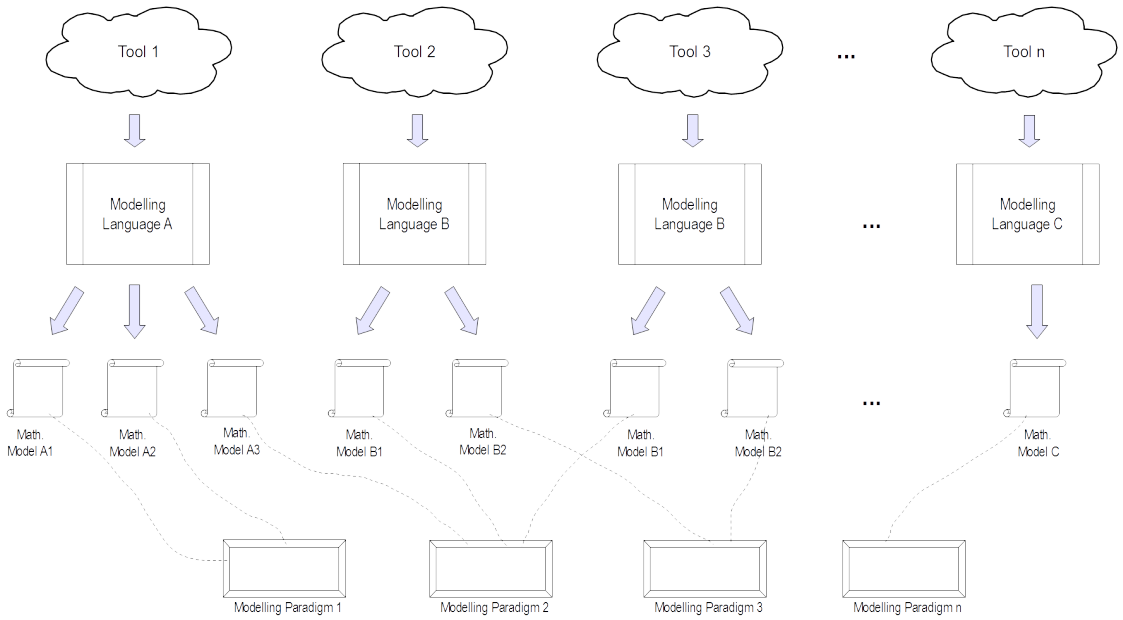

Figure 2.3: Tools, modelling languages, and modelling paradigms.

architecture, which then is refined to a more detailed one describing system components. This phase is followed by coding that typically also includes testing activities of the individual modules or parts of the systems under development. Subsequent to the coding phase, system testing is applied intensively. Finally, the product will be released and maintained if all test have been passed successfully. During the development and afterwards, in each of these phases it is possible to introduce errors or not correct behaviours that influence the correctness of the whole system.

Considering the single parts of the development, we treat mainly the following phases: design, implementation, and analysis. After collecting the requirements the design phase describes with models what the system does, then the implementation chooses the destination code that represents the intended system, and finally the validation and verification of the system. The schema in Figure 2.4 has returning edges to the precedent phases. In fact, the defects and mistake that could happen in the implementation phase can determine modifications of the design. The same could happen during the analysis phase, where detected errors can determine revisions in the other

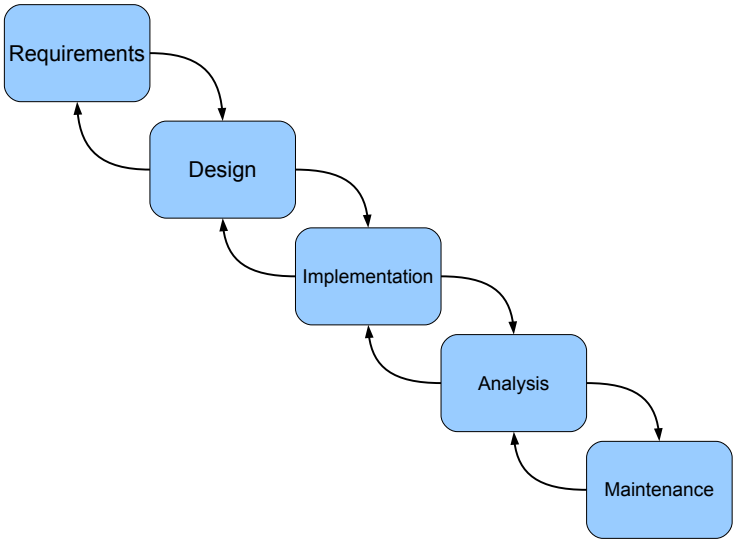

Figure 2.4: A waterfall model design process for software systems.

phases. The precedent phases, which need at some point of the development a revision, are not necessarily all the precedent phases but also some of them.

2.4.2 AutoFocus 3

AutoFocus 3 is a research tool for modelling. Its semantics is founded on the Focus theory. AutoFocus 3 is designed for the development of reactive, software-intensive, embedded systems. It is implemented on top of the Eclipse[2] platform. In this tool, systems are modelled by software architectures composed of components with executable behaviour descriptions (e. g., automata with input and output). As explained in Section 2.2, Focus model of computation is characterized by the notion of streams, traces and stream processing functions. AutoFocus 3 supports the timed synchronous streams with a discrete notion of time, that is, a subdivision of the time in logical ticks or steps, in which the model components synchronously interact according to global clocks. In the tool are implemented development views, in order to support the different level of abstraction of a system. Systems specifications are supported by different development views and the models have more levels of abstraction. A requirement framework called Model-based Requirements Analysis (MIRA) supports the specification of system require-

[2] `http://www.eclipse.org`

ments informally guided by templates [127]. The following formalization step of these informal specifications is done using a set of integrated formal notations. Formalized requirements are checked with automated analysis techniques, as simulation, formal verification, and model-based testing. In AutoFocus 3 the logical architecture of the system is graphical described, according to the model of computation introduced by Focus: the system model is given by a set of communicating components, each one with a defined interface and an implementation. The components exchange typed messages instantaneously through i/o ports. The tool provides an interactive graphical simulation environment and testing/verification capabilities for the logical architecture. Formal verification is guaranteed by a user-friendly integration of model checking techniques, as better explained in Section 5.2. The technical architecture of the system, in terms of execution environment represented by execution control units and communication buses, has also a modelling view in AutoFocus 3 and can be related to the logical architecture of the system. Hardware aspects are captured in a topology model, which describes execution and transmission units such as electronic control units and bus systems. A deployment model allocates components to execution units and allows generating C code of the system, which can be compiled and installed into the demonstration hardware.

2.4.3 Our Methodology

As presented in [32], our methodology for the development of embedded system was already used in industrial case studies, describes the system at the logical level, and requires further extensions to work with CPSs. By developing a large system, a number of iterations are needed to cope with gain of knowledge about the system, as well as modifications in the requirements. Fig. 2.5 illustrates the structure of our generalized development methodology in a top-down manner. The boxes represent artefacts that have been developed and the arrows show which other artefacts are derived. The process starts by structuring of initial requirements using specific syntactic patterns: this first step raises the level of precision by transforming the free text requirements into a structured form using specific pre-defined syntactic patterns, as presented in [59]. An informal specification consists of a set of words, which can be classified into two categories: content words and keywords. Content words are system-specific words or phrases, e.g., *"Off-button is pressed"*. The set of all content words forms the *logical interface* of the system with its environment, which can be understood as a special kind of domain specific glossary that must be defined in addition. Keywords are domain-independent

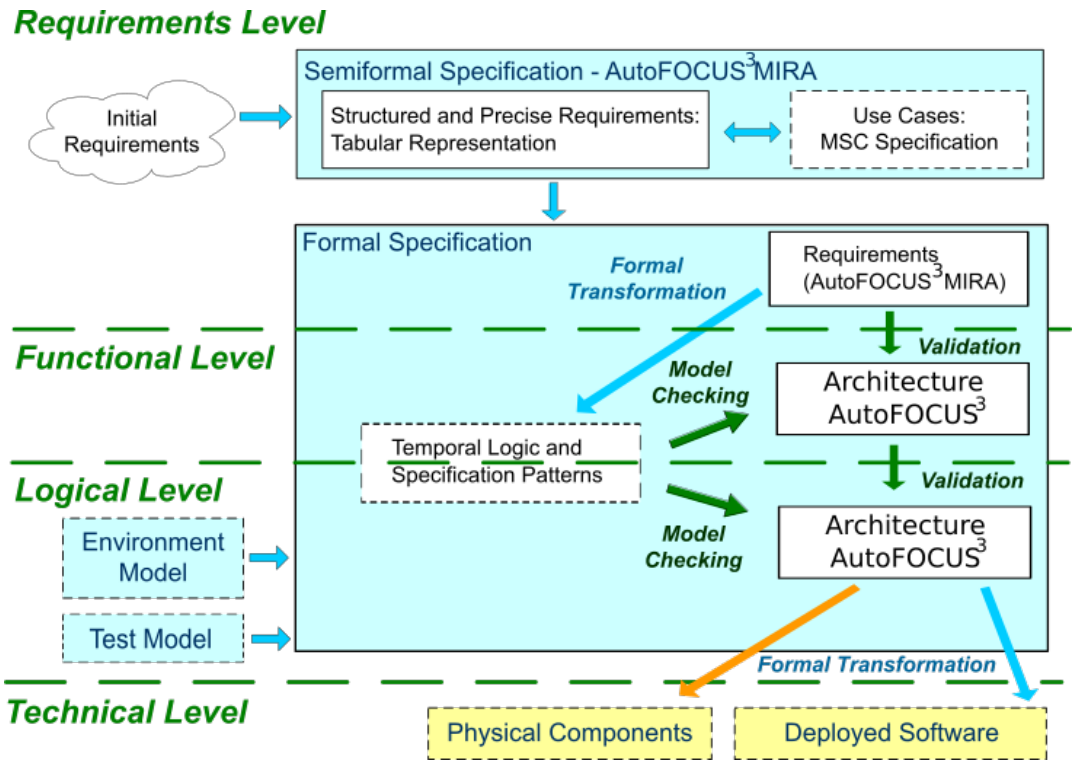

Figure 2.5: A representation of our generalized development methodology.

and form relationships between the content words (e.g., "if", "then"). Thus, a semiformal specification consists of a number of requirements described via textual patterns, which can be easily understood even by engineers unfamiliar with formal methods. Using this description to structure the informal specification, missing information can be already discovered. Furthermore, possible synonyms are identified that must be unified before proceeding to a formal specification. Analysis of the semiformal specification document should also detect sentences, which need to be reformulated or extended. This specification can now be schematically transformed to a *Message Sequence Charts* (MSCs) [65] representation, as an optional step relevant for highly interactive systems.

The methodology proceeds with the translation of semiformal specification to AutoFocus 3 (MIRA). As mentioned above AutoFocus 3 is a modelling tool based on the semantics and model of computation of the Focus modelling theory. Development views in the tool support the viewpoints from the SPES matrix.

We model with AutoFocus 3 two kinds of specifications: a formal specification of *system requirements* and the corresponding *architecture specification* (both are based on the semiformal specification). This prepares the basis to verify the system architecture specifications against the requirements by translating both to the model checker. The requirements specification is schematically translated to temporal logic or specification patterns, which gives basis to model-check the model (cf. [31]). There is also support for a verification with the theorem prover Isabelle/HOL [97] via the framework "Focus on Isabelle" [122]. AutoFocus 3 system model is encoded in Isabelle/HOL, as well as proof of theorems that support subsequent verification of properties in Isabelle/HOL. The following features support the integration of model checking in AutoFocus 3: tight coupling of verification properties with model elements, visualization and simulation of counterexamples, and different specification languages for the formulation of properties. This is explained in Section 5.2.

This twofold verification firstly guarantees that actual properties of the design model are verified and the implementation code and secondly, different results in both techniques may indicate an implementation fault in one of the generators.

Finally, we proceed from the logical to the technical level, where we split our model into software and physical components. Transformation to a C code can be done using the code generator of the tool: we have shown that the C program produced by the AutoFocus 3 code generator is a reasonable simulation of the model. Altogether, the methodology guides us from informal specifications via stepwise refinement to a verified formal specification, a corresponding executable verification model, and a C code implementation.

2.4.4 Methodology Case Studies

One of typical example of a hybrid system from the automotive field is a Cruise Control System (CCS), which includes software and physical parts. With our methodology, we modelled two different variants of the system using two development methodologies with similar strategies but different focal points. The first case study [56] developed an Adaptive Cruise Control (ACC) system with Pre-Crash Safety functionality. This was motivated and supported by DENSO Corporation. The second case study [124, 123], developed a CCS with focus on system architecture and verification, and was supported by Robert Bosch GmbH. Third case study [57], motivated and supported by DENSO Corporation, was the development of a Keyless Entry-System, a system with a huge state space, which in addition required a

distributed deployment. In the logical representation, we did not distinguish between physical and virtual parts of the system. A sample-property of a Cruise Control System can be represented as follows: *if the driver pushes the ACC-button while the system is on and none of the switch-off constraints occurs, the system must accelerate the vehicle during the next time unit respectively to the predefined acceleration schema.* This implies that the system must analyse the information from sensors to check whether any switch-off constraint occurs, e.g., if the battery voltage is too low or if the gas pedal sensor fails.

In the ACC case study [56], we apply model checking techniques for temporal logic formulas, ranging from SAT solving to interactive verification with respect to its notational power and complexity. The verification was facilitated by the SMV tool [93] and the properties were specified in LTL (Linear Temporal Logic, e.g. [107]), a widespread specification notation suitable for automatic model checking. In this formalism supports boolean logical operators and temporal operators and allows for formalising properties of system states and their changes during system execution. We used SMV to check requirements in form of temporal logic formulas also in the Keyless Entry-System case study [57]. Where necessary we abstracted variables/data types and/or restricted value ranges, especially for integer variables. In the Cruise Control System case study [124, 123], we applied semi-automatic verification with a theorem prover and full automatic by a model checker. Formal requirements could be checked using a theorem prover, through a translator, which generates Isabelle/HOL theories from AutoFocus 3 models.

Chapter 3

Focus Hybrid Specifications

In Chapter 2, we presented the fundamental concepts for the hybrid systems and our methodology for modelling of embedded systems. The theory is only for discrete time, discrete elaboration and data type systems suited, and does not support continuous time. An extension towards continuous interactive systems has been presented in [21]. In this chapter we introduce a new type of components, which are based on the concepts already explained in the modelling fundamentals chapter and are now extended to support the design of CPSs in Focus.

3.1 Hybrid Components

At the requirement level, a specification of a distributed system should contemplate a suitable definition for system components as well as for the environment. The Focus approach covers these definitions, providing requirements for the components and assumptions for the environment. For discrete systems, the communication histories or traces are represented by infinite sequences of messages at discrete time intervals. Thus, if the set of the exchanged messages is M, then the trace function can be represented by a function $f : \mathbb{N} \to M^*$. This is a limiting restriction for reactive systems, which interact in continuous real-time with physical components and their environment, as for instance CPSs. Therefore, in order to permit a real-time execution of the systems, time should be represented in the $\mathbb{R}_+$ realm, that is, the set of all non-negative real numbers. Consequently, the system needs to have continuous data types and an instrument to define the continuous evolution of such data types. Differential equations are a solution for systems that work with physical or mechanical parts.

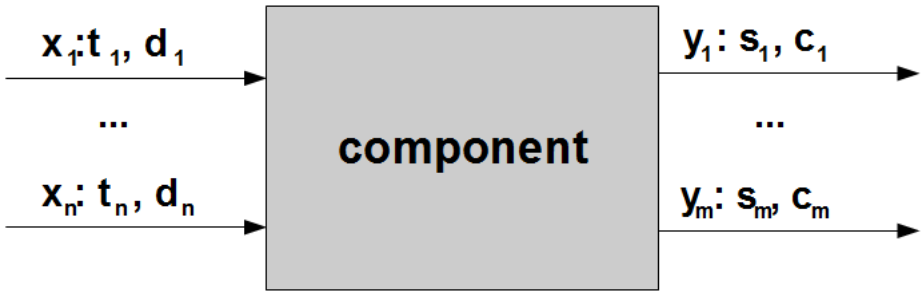

Figure 3.1: Hybrid component representation with its interface.

Considering the definitions in the precedent chapter for discrete elaboration (Section 2.2) and the continuous dynamic of hybrid automata (Section 2.3), we define a *hybrid component* based on dense streams [94], which introduces continuous behaviour in FOCUS. Also the concepts presented in [21] for the definition of continuous time systems are considered.

A component is an elaboration unit that communicates with other components or with the environment through input and output channels. We denote with I and O, the set of input and output channels of a component respectively. Associated with each channel there is a data type, which specifies the type of messages sent or received. The communication may be in discrete or continuous time, so the channels are continuous or discrete depending on the modelling necessities. With $I \rhd^{+} O$ we denote the *syntactic interface* of a hybrid component. A schematic representation of it is depicted in Figure 3.1, where x_i are input streams, y_i output streams, t_i and s_i the type of the messages transmitted and, d_i and c_i indicate the type of the stream $(1 \leq i \leq n, m)$.

In an analogous manner, the interface behaviour of a hybrid system with identifiers of the input and output streams and the type of the exchanged messages is defined in [21]. This work uses specific prefixes in the identifier instead of an added symbol for the stream type. During the system execution, a system component exchanges messages through the channels. The sequences of messages determine the behaviour. In a network of components, connected by channels, each channel connects an *output port* to an *input port*. Streams (with elements from M) may be infinite (M^{∞}) or finite (M^{*}). A discrete component modelled in FOCUS communicates through channels using time intervals of equal length, where in each interval a finite sequence of messages eventually empty is transmitted. By $(M^{*})^{\infty}$ we denote the infinite

streams of sequences of elements of set M, which correspond to a function $\mathbb{N} \setminus \{0\} \to M^*$.

3.2 Hybrid Streams

Scholz and Müller [94] propose an extension of FOCUS with continuous elaborations. The authors modify the semantics of the stream functions introducing dense streams. In the following definitions, a total function is only defined for all possible input values a correspondent output value, whereas a partial function defines on a subset of the input values.

Definition 9 (Dense stream). *Let M be the set of all messages (potentially infinite), a dense stream x over a set M is described by a total function:*

$$x : \mathbb{R}_+ \to M$$

We define hybrid streams to represent continuous signals and discrete signals over continuous time.

Definition 10 (Hybrid stream). *Let M be the set of all messages (potentially infinite), a hybrid discrete stream x over M is described by a function:*

$$x : I \to M^*, with\ I \subseteq \mathbb{R}_+$$

whereas a hybrid continuous stream y over a set N of messages (typically $N = \mathbb{R}$) is described by a total function:

$$y : \mathbb{R}_+ \to N$$

$M^{\mathbb{R}_+}$ refers the set of all hybrid streams over M, for which we define the following relations, where x, y, and z are streams and a, b, and c are elements of M:

- $\langle\rangle$ is the empty stream, that is a stream without elements with an empty domain.
- $\langle m_1, \ldots, m_n, \ldots \rangle$ indicates the finite or infinite stream containing elements $m_i \in M^*$.
- $el(x, t)$ returns the element of x at time $t \in \mathbb{R}_+$, if x is not empty or has no element defined at time t, otherwise it returns $\perp$, that represents an undefined result.

- *a in x* produces *true* exactly if a occurs in x.
- $a©x$ denotes the substream of x that contains only the a-items. It is a filter operator, for example, $a©\langle a, b, a, c, \rangle = \langle a, a \rangle$. The first operand may also be generalised to a *set* of items. We define it formally, given a hybrid stream x over the set of messages M and $v \subseteq M^*$, $v©x = \langle m_1, m_2, \dots \rangle$ with $m_i \in v$, where i can be finite or infinite.
- with $x \downarrow t$ the restricted to the stream $x|_{[0,t]}$, that is the stream restricted to the time interval $[0, t]$.

A restriction of a stream to a finite interval can be affected by the Zeno's paradox. Zeno's paradox argues that a time interval defined on $\mathbb{R}_+$ cannot be executed in a finite amount of time, since the interval is composed of infinite discrete atomic moments. In Chapter 4, we sample continuous streams in finite time intervals, dynamically reducing and incising their dimensions. By doing so, we are not affected by Zeno's paradox because there is an inferior limit to the dimension of the interval based on the computational power of the hardware in which the system is sampled and simulated.

In [21] streams with continuous domains called *timed streams* are introduced. Timed streams are *dense* if the domain $TD \subseteq \mathbb{R}_+$ is an interval in $\mathbb{R}_+$; and are *discrete* if the domain $TD \subseteq \mathbb{R}_+$ is a finite interval. We defined, according to the Definition 10, hybrid continuous streams as functions with domain equal to $\mathbb{R}_+$, and hybrid discrete streams as functions with domain in $I \subseteq \mathbb{R}_+$. Therefore, it is possible to define a dense time stream as a hybrid continuous stream if its domain $TD = \mathbb{R}_+$. Instead, if its domain is a proper subset of $\mathbb{R}_+$, we can define it with a hybrid discrete stream. In the same manner, a discrete timed stream can be defined with a hybrid discrete stream.

Stream processing functions are specified over dense streams as follows.

Definition 11 (Component function). *A component description function f with m input and n output ports is a function*

$$f : (M_1^{\mathbb{R}_+})^m \to (M_2^{\mathbb{R}_+})^n$$

with sets of input and output messages M_1 and M_2 respectively, which are not necessarily different.

A system defined with the Definition 11, $S : X \to Y$, where X and Y are the sets of messages, is linear if satisfies the following property $\forall x_1, x_2 \in X$ and $\forall a, b \in \mathbb{R}$:

$$S(ax_1 + bx_2) = aS(x_1) + bS(x_2)$$

This definition describes components only as continuous streams, but we also want to have the possibility to specify systems with discrete streams. We define our syntactic interface for hybrid components, considering the following processing functions, a function f for the description of discrete streams with m inputs and n outputs; and a function g for the description of continuous streams with k inputs and l outputs:

$$f : M_1^\omega \times \cdots \times M_m^\omega \to N_1^\omega \times \cdots \times N_n^\omega$$

$$g : L_1^{\mathbb{R}+} \times \cdots \times L_k^{\mathbb{R}+} \to O_1^{\mathbb{R}+} \times \cdots \times O_l^{\mathbb{R}+}$$

Definition 12 (Syntactic interface). *Hybrid components have the following syntactic interface, where each element is a hybrid stream (Definition 10), a function with ($m + k$ inputs and $n + l$ outputs):*

$$h : (M_1^*)^{\mathbb{R}+} \times \cdots \times (M_m^*)^{\mathbb{R}+} \times L_1^{\mathbb{R}+} \times \cdots \times L_k^{\mathbb{R}+} \to \mathcal{P}((N_1^*)^{\mathbb{R}+} \times \cdots \times (N_n^*)^{\mathbb{R}+} \times O_1^{\mathbb{R}+} \times \cdots \times O_l^{\mathbb{R}+})$$

where m may be equal to zero if there are discrete streams in the input, also n may be equal to zero if there are no discrete streams in the output. k may be equal to zero that means no input continuous streams, also l may be equal to zero that means no output continuous streams. Anyway, n and l cannot be both at the same time equal to zero. The sets of messages are not necessarily different.

This syntactic interface is defined in continuous time as in Definition 11 for continuous streams. Streams that represent a discrete behaviour are also continuous, but produce a valid output only at a time corresponding to a discrete computation step, predefined with a length of $\Delta \in \mathbb{R}_+$. Therefore, the elaboration of the hybrid component is defined following.

Definition 13 (Elaboration function). *The elaboration function el describes the functional behaviour of hybrid components with a value $\Delta \in \mathbb{R}_+$, denoted as the discrete computation step:*

$$el(h(x_1, \ldots x_n, y_1, \ldots y_m), t) \begin{cases} (n_1, \ldots n_k, o_1, \ldots o_l) \in h(x_1, \ldots x_n, y_1, \ldots y_m), \\ \qquad \text{if } t = \Delta * i, \text{ for some } i \in \mathbb{N} \\ (\bot, \cdots \bot, m_1, \ldots m_j) \in h(x_1, \ldots x_n, y_1, \ldots y_m), \\ \qquad \text{otherwise} \end{cases} \tag{3.1}$$

where $t \in \mathbb{R}_+$, $x_1, \ldots x_n \in (M_h^*)$ are input discrete variables, $y_h \in (L_h)$ are input continuous variables, $n_h \in (N_h^*)$ are discrete output variables, and $m_h \in (O_h)$ are continuous output variables. The variables associated to discrete output streams return no value at time values $t \in \mathbb{R}_+$ that are not in the sequence $\Delta * i$. In fact, the discrete streams have a domain set $I \subseteq \mathbb{R}_+$ in accord with Definition 10.

Functional specifications (in Section 2.2.4) need a mathematical foundation for implementing feedback loops. In the discrete case, least fixed points over domains have been used to describe the loop semantics [22]. In analogy, dense streams for stream processing functions use Banach's fixed point theory [94]. The functional behaviour of hybrid systems defined in [21] is a function that maps input values to output values, as presented in the syntactic interface of hybrid components. The behaviour of the hybrid systems and components can be deterministic or nondeterministic in both the definitions. It is deterministic if the function h returns only one output sequence for each input sequence.

3.2.1 Causal Components

If the output produced by a component depends on the current and past inputs this component is called *causal*. Consider a hybrid stream $x : \mathbb{R}_+ \to M$ or $x : I \to M^*$ for some message set M, we define the restriction in time of x as $x|_{t \leq \alpha}$, that is the function defined for time $t \leq \alpha$ as $x|_{t \leq \alpha}(t) = x(t)$. Now considering a deterministic component $S : X \to Y$ with hybrid streams X and Y over some set of messages $M_1 \ldots M_k$, where k is the total number of streams. A component is *weak causal* if for all $x_1, x_2 \in X$ and $\alpha \in \mathbb{R}_+$,

$$x_1|_{t \leq \alpha} = x_2|_{t \leq \alpha} \Rightarrow S(x_1)|_{t \leq \alpha} = S(x_2)|_{t \leq \alpha}$$

Therefore, for a weak causal component if two inputs is equal up to and including time α then the outputs are also identical up to and including time α.

A component is *strongly causal* if for all $x_1, x_2 \in X$ and $\alpha \in \mathbb{R}_+$,

$$x_1|_{t < \alpha} = x_2|_{t < \alpha} \Rightarrow S(x_1)|_{t \leq \alpha} = S(x_2)|_{t \leq \alpha}$$

The difference to a weak causal component is that the inputs for a strong causal component has to be identical up to α but not including it, i.e. considering the output at some time t does not depend on its input at time t.

The elaboration of a data input to produce a data output, as for hybrid components, takes computing time. This computing time interval is included in the output elaboration of a strongly causal component. This notion of causality describes the computational elaboration also for nondeterministic components. The timed streams defined in [21] have the same causality property.

3.3 Hybrid I/O Interfaces

We introduced in FOCUS continuous and discrete streams determining hybrid i/o interfaces between hybrid components. This way we support the heterogeneous nature and increasing complexity of CPSs. In fact, CPSs are composed by discrete and continuous components, which interact with each other. In this section, we analyse the hybrid i/o interface between FOCUS hybrid components.

In Definition 12 we specified the syntactic interface $h : I \rhd^{+} O$ of a hybrid component, as a function with input channels I and output channels O. Typically, a model is composed of more than one hybrid component, in order to subdivide functionalities in more interconnected logical components. The connection is established from one (or more) output channel of a component to one (or more) input channel of another one. Moreover, FOCUS theory permits to hierarchical define components inside other components, defining different levels of abstraction. The logical behaviour is then specified at the last level of the component hierarchy. A syntactic interface of a hybrid component is composed by a set of typed i/o channels. Each channel has is functional behaviour described by a hybrid discrete or hybrid continuous stream function. The different type of stream determines different output production frequencies, as shown in Definition 13.

If the output channels of a FOCUS hybrid component are described only by hybrid discrete streams, then the model of computation of the component is *discrete*. In such discrete component, the time flows discretely in steps of predefined length. Otherwise, a hybrid component with at least a hybrid output continuous stream is executed in *continuous* time. If an output port of a component is associated to an input port of another component, then the two components are interconnected. The connection between components with the same model of computation obviously does not need modifications to the execution of the components. In our modelling theory, it is possible to connect continuous and discrete components obtaining a system that has to handle different computation models. The simplest case is the i/o connection

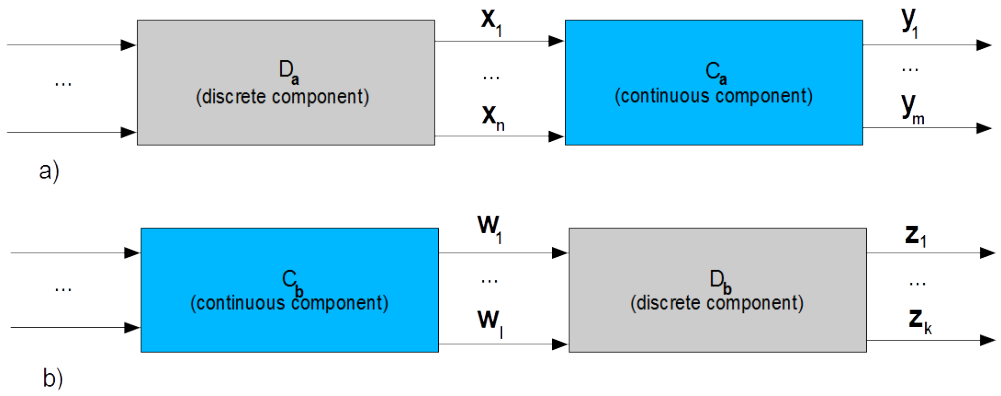

Figure 3.2: The simplest i/o connections between two hybrid components with different models of computation.

between two components, where each one has a different model of computation, as depicted in Figure 3.2. In this interface analysis, we can reduce the execution of interconnected components with different model of computation to these mentioned cases also when more than two components are connected. In this picture, two kinds of connection are described. The first kind of connection is when output ports of a discrete component are input ports of a continuous component (3.2a); in the second kind, output ports of a continuous component are input ports of a discrete component (3.2b).

3.3.1 Component Connection: Discrete to Continuous

In 3.2a, the discrete component D_a provides input values to the continuous component C_a at each discrete elaboration step, through the streams $x_1 \ldots x_n$. At least one of the output streams $y_1 \ldots y_m$ of the component C_a has associated a hybrid continuous stream, which is a continuous time function. D_a provides the input values to C_a at discrete steps with a constant time interval. In Figure 3.3, we depicted the communication between the C_a and D_a FOCUS hybrid components, where the black arrows describe the internal discrete or continuous elaboration function. This function produces the output signals of the component. The discrete component D_a produces output values at each discrete time points t_i with $i > 0$. Each time interval has the same predefined length. The input messages given from D_a to C_a at discrete time interval can be commands, which might influence the elaboration of the C_a component. In an application scenario the discrete component

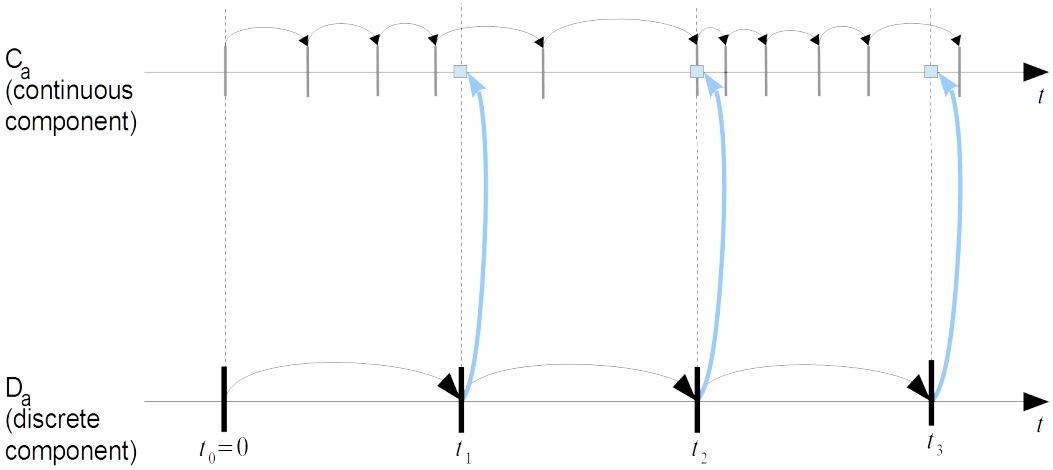

Figure 3.3: Message interaction between the hybrid components, described in Figure 3.2a, with different models of computation.

D_a can be a software controller connected to a continuous time actuator represented by the C_a component. The discrete input commands received by the component C_a can be logical commands to change its internal state and as consequence its output. As depicted in Figure 3.3 at time t_2 the component C_a received a message from the D_a component and instantly change its internal elaboration function and so its output. The change of the elaboration might also happen sometime after a message is received. As for instance in Figure 3.3 at time t_1: the C_a component receives a message and after some time change the elaboration function. Typically, in a component architecture as in 3.2a, the discrete component controls or influences the behaviour of the continuous component.

3.3.2 Component Connection: Continuous to Discrete

The second type of i/o connection between components with different models of computation is depicted in 3.2b; where the continuous component C_b provides messages to the discrete component D_b through the streams $w_1 \dots w_l$. The output of the component C_b is produced at continuous time, whereas the component D_b, according to its model of computation, receives and reads the input messages only at discrete time points t_i with $i > 0$. The communication between C_b and D_b is depicted in Figure 3.4. In this figure the internal

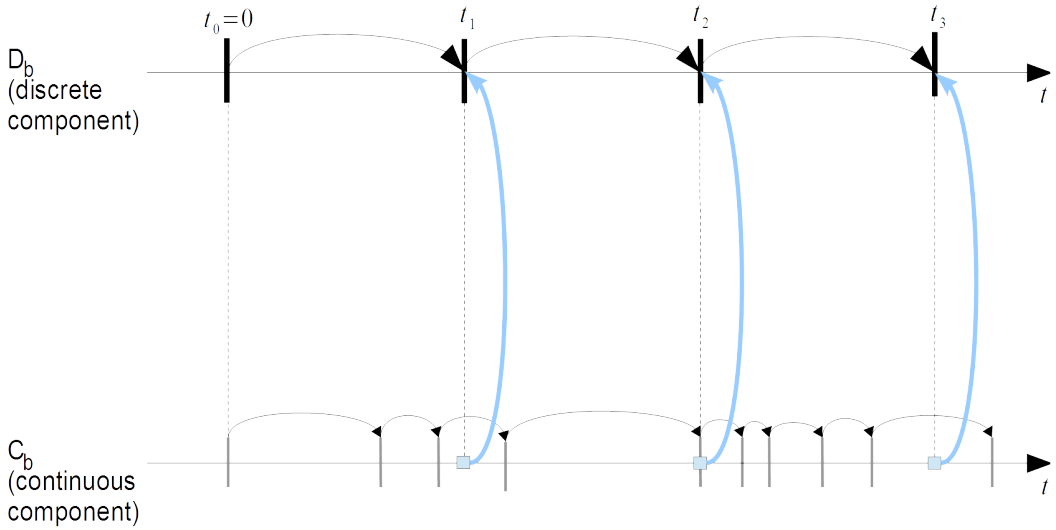

Figure 3.4: Message interaction between the hybrid components, described in Figure 3.2b, with different models of computation.

elaboration function of the components are described by black arrows. The component D_b receives at discrete time steps the output values produced by the component C_b. This configuration may represent, for instance a system where a software component (D_b) periodically reads data from a continuous time component (C_b). In some applications, it is enough for D_b to read only the messages produced at discrete time steps, but in some circumstances, also the produced messages between a discrete step are important. In order to read this data from the component C_b, the streams functions $w_1 \dots w_l$ can be extended: a modified stream w_j provide at discrete steps not its actual value ($w_j(t_i)$) but a calculated value based on all the value of w_j in the time interval $[t_{i-1}, t_i]$. We can obtain this extending the hybrid stream function with an additional function α_j, which at discrete time steps t_i reads the values produced by w_j in the time interval $[t_{i-1}, t_i]$ and produces $\alpha_j(w_j|_{[t_{i-1},t_i]}, t_i)$ with a suitable function. The function α_j may calculate for instance the maximal value, minimal value, the average or median in the interval $w_j|_{[t_{i-1},t_i]}$. The component D_b receives values from the component C_b only at discrete time steps, but with the α_j function also the values produced between two discrete steps are considered. This way the modelling theory permits to reduce the information gap, when continuous components provide input data to discrete components.

3.3.3 I/O Interface and Logical Implementation

In this section, we shown some aspects of our modelling theory by the definition hybrid i/o interfaces. It is important to have a modelling theory with formal and mathematical definition of the interfaces, because a CPS that contains both discrete and continuous components utilize the i/o interfaces for coordination, control and feedback operations. In our modelling approach, the i/o interfaces are formally defined by the streams.

In our system development methodology (Section 2.4) we introduced a logical representation of the system. In this system representation, the formal interface and the internal logical implementation of the component are its characteristic elements. The mentioned characteristics of our i/o interface behaviour are defined independently to the internal logical implementation of the components. In the next section, we present a possible logical implementation for FOCUS hybrid components: the hybrid i/o state machine.

3.4 Hybrid I/O State Machines

We define now the implementation of a hybrid component based on the definition of hybrid automata and our notion of component in FOCUS.

Definition 14 (Hybrid i/o state machine). *A hybrid i/o state machine is a tuple* $H = (\Sigma, Var, Init, I, O, Dom, E, f, G, R)$ *with:*

- *A state space* $\Sigma = (Q \times V)$ *where* Q *is a set of discrete states* $Q = \{q_1, q_2, \dots\}$ *and* V *a set of continuous states* $V \subseteq (M_1^*)^{\mathbb{R}_+} \times \cdots \times (M_l^*)^{\mathbb{R}_+} \times (M_{l+1})^{\mathbb{R}_+} \times \cdots \times (M_n)^{\mathbb{R}_+}$*, where* n *is the total number of variables. To each element of* V *a variable in the variable set* Var *is associated.*

- *A set of initial states* $Init \subseteq \Sigma$*.*

- *A set of input* $I \subseteq V$ *and output* $O \subseteq V$ *states, respectively for input and output channels, with* $O \notin \emptyset$*. To each element of* I *and* O *corresponds a hybrid stream as specified in Definition 10.* $O_d \subseteq O$ *and* $O_c \subseteq O$*, with* $O_c \cap O_d = \emptyset$ *are respectively the output state space associated to continuous and discrete stream functions.*

- *A set of internal continuous* $Int \subseteq V$ *state space and internal discrete* $Int_d \subseteq V$ *state space both with no channels associated, with* $I \cap O \cap Int \cap Int_d = \emptyset$*. We denote with* $W = Int \cup O_c$ *the set of internal and*

output state spaces. Any variable in Var can only be in one of these sets or in one of the i/o sets defined above.

- *A domain function $Dom : Q \to \mathcal{P}(V)$.*
- *A set of edges $E \subseteq Q \times Q$ that represent the discrete state transitions.*
- *A vector field function $f : Q \times V \to W$.*
- *A guard condition function $G : E \to \mathcal{P}(V)$.*
- *A reset map function $R : E \times V \to \mathcal{P}(W \cup O_d)$ that represent together with the vector field the continuous dynamics.*

Then a hybrid component has a directed graphs encapsulated in a component with input and output communications ports. Vertices are in Q and edges in E. Associated with each discrete state $q \in Q$ there is a set of initial states $\{v \in V \mid (q, v) \in Init\}$ and a vector field $f(q, \cdot) : (M_1^*)^{\mathbb{R}_+} \times \cdots \times (M_l^*)^{\mathbb{R}_+} \times (M_{l+1})^{\mathbb{R}_+} \times \cdots \times (M_n)^{\mathbb{R}_+} \to W$ for the evolution of the variables in W. In each discrete state, the vector field function describes the evolution of the continuous variables, which are guided by differential equations. Differential equations are widely used to describe the logical behaviour of CPSs that work with physical or mechanical parts. A guard $G(q, q') \subseteq (M_1^*)^{\mathbb{R}_+} \times \cdots \times (M_l^*)^{\mathbb{R}_+} \times (M_{l+1})^{\mathbb{R}_+} \times \cdots \times (M_n)^{\mathbb{R}_+}$ and a reset function $R((q, q'), \cdot) \subseteq W \cup O_d$ are defined for each edge.

If t_1 and t_2 are linear functions, then a *linear inequality* is $t_1 \leq t_2$. An ordinary differential equation wherein the derivatives with respect to the independent variable have order of at most one is a first order ordinary differential equation.

Definition 15 (Linear hybrid i/o state machine). *An hybrid i/o state machine $H = (\Sigma, Var, Init, I, O, Dom, E, f, G, R)$ is linear if the following three conditions hold:*

1. *the domain and the guard condition functions are boolean combination of linear inequalities;*
2. *for each variable $w \in W \cup O_d$ the reset function is a linear formula over V, $w := \alpha_v$, where the α_v have to be linear terms;*
3. *for each variable $w \in W \cup O_d$, state $q \in Q$ and $v \in V$ the vector field $\dot{w} = f(q, v)$ is expressed by a linear differential equation;*

We defined a hybrid state machine that communicates through input and output channels over hybrid streams in Definition 10. The continuous state space V is subdivided into internal variables, output and input subsets. The i/o variable subsets are associated with i/o channels of the component. In fact, we could abstract the internal automata considering only the syntactical interface as depicted in Definition 12. In this definition, there is a distinction between discrete and continuous streams. Continuous states are defined considering each discrete state and the Dom function, when the system is in such a discrete state the system evolves in the continuous space according to this function. From an initial state (q_0, v_0) the evolution of the system is guided by the differential equation over $w \in W$

$$\dot{w} = f(q_0, v)$$
$$v(0) = v_0$$

and remains constant in the discrete state q

$$q(t) = q_0.$$

If the actual state v remains in $Dom(q_0)$ then the continuous elaboration is active and executed. A discrete transition to q_1 is determined by the current value of v, when it is in the guard $G(q_0, q_1) \subseteq (M_1^*)^{\mathbb{R}_+} \times \cdots \times (M_l^*)^{\mathbb{R}_+} \times (M_{l+1})^{\mathbb{R}_+} \times \cdots \times (M_n)^{\mathbb{R}_+}$ of some edge $(q_0, q_1) \in E$. After a discrete transition, the reset function sets some values in $R((q_0, q_1), v) \subseteq (M_1^*)^{\mathbb{R}_+} \times \cdots \times (M_k^*)^{\mathbb{R}_+} \times (M_{k+1})^{\mathbb{R}_+} \times \cdots \times (M_m)^{\mathbb{R}_+}$ for the variables in $W \cup O_d$, that is a subset of V and therefore with $m \leq n$, this way the continuous evolution can be reset.

The execution of the state machine is determined by a sequence of continuous and discrete modifications, induced by the transitions, the discrete state elaborations and guided by the input streams. We define the executions using the notations presented in [79].

Definition 16 (Step execution). *A step execution is a sequence:*

$$\pi = (q_0, v_0) \xrightarrow{t_0} (q_0, v_1') \rightarrow (q_1, v_1) \xrightarrow{t_1} \ldots \xrightarrow{t_k} (q_{k-1}, v_k') \rightarrow (q_k, v_k)$$

with $i \in \{1, \ldots, k\}$, q_0, $q_i \in Q$, v_0, v_i, $v_i' \in V$ and $t_i \in \mathbb{R}^+$. The state (q_0, v_0) is an initial state of the hybrid state machine. The continuous elaboration of a discrete state q_i, for the time interval $[0, t_i)$, is represented by the transition $\xrightarrow{t_i}$. The discrete and atomic transition from the state (q_{i-1}, v_i') to (q_i, v_i) is represented by this arrow $\rightarrow$. The variables in Var corresponding to the input elements $I \subseteq V$ evolve their values according to the corresponding input streams.

We have denoted with v the actual states of the variables in V in our explanation. We introduce a notation for specifying the state of a hybrid state machine at time $t \in \mathbb{R}^+$ as $v(t) \in V$, that is the value of all variables at time t. The actual value for the output variables is defined as following.

Definition 17 (Output variables evaluations). *$o(t)$ is a function:*

$$o(t) = \begin{cases} & \text{where } v(t) = \{x_1, \ldots, x_n\} \text{ with } v(t) \in V,\ i \in \mathbb{N}_+,\ i \leq n \\ \{y_1, \ldots, y_m\} & \text{and } y_l = x_j \in O \text{ with } l \in \mathbb{N}_+,\ l \leq m \leq n,\ j \in \{1, \ldots, n\}. \\ & \text{If } t \notin I \text{ the discrete output variables } y_h \in O_d, \text{with} \\ & h \in \mathbb{N}_+ \textit{ and } h \leq m \leq n, \text{ are equal to } \perp . \end{cases} \tag{3.2}$$

The discrete output variables follow their elaboration as explained in Definition 13. Therefore, there are values in the output different from $\perp$, when the time corresponds to a "discrete" step. The value Δ of the step is constant for the whole components architecture. In our hybrid state machine during the execution the reset map function R can set values to discrete output variables in O_d at any time $t \in \mathbb{R}^+$. The elaboration function, as defined in Definition 13, returns in each step $i+1$ the values collected in the time interval $(\Delta * i, \Delta * (i+1))$ as a single output sequence at time $t = \Delta * (i+1)$.

Predicates specified for hybrid components, can be defined considering the syntactic i/o interface $I \rhd^+ O$ only. In addition, step executions, as shown in Definition 16, can be considered from the point of view of the received input and generated output messages. From a step execution $\pi = (q_0, v_0) \xrightarrow{t_0} (q_0, v_1') \rightarrow (q_1, v_1) \xrightarrow{t_1} \ldots \xrightarrow{t_k} (q_{k-1}, v_k') \rightarrow (q_k, v_k)$ we can derived a trace, that is a sequence of messages over the time interval $[0, \varrho]$ where $\varrho = \sum_{i=0}^{k} t_i$, in which at time $t \in [0, \varrho]$ the elements are the actual values of the input and output variables. Given a predicate $p \subseteq \Sigma$ we say that π terminates in p if and only if $(q_k, v_k) \in p$. Considering a hybrid state machine H and the language of the traces of input and output messages $L(H)$, given a predicate $p \subseteq \Sigma$ the language $L_p(H)$ is the set of traces that accept p in runs of H ending in p.

The hybrid state machines already introduced in FOCUS have different characteristics respect to the presented hybrid i/o state machines. In fact, in [21] they are described as generalization of Mealy machines, where the internal behaviour is specified only as state transition function. The set of states Σ is not defined. In this work we define explicit the guard functions, the reset map function and the vector field function, which determine the

conditions for the state change, the reset of the variables and the evolution of the output and internal variables in the states. Another difference is that the hybrid machines in [21] can also be executed in a discrete time, instead in our definition the variables and the states evolve in continuous time. We define the composition of the state space, the behaviour of internal and output variables explicit, because we want to specify a logical behaviour of hybrid components. In Chapter 6, we model a case study of a simplified train brake system using hybrid i/o machines, in order to show the capabilities of the logical model. Moreover, we define the nature of the evolution of the output and internal variables, the transition conditions to have a precise class of hybrid systems in order to apply two important aspects: formal verification and discrete sampling of continuous variables.

3.4.1 Composition of Hybrid I/O State Machines

Components communicate using input/output ports. So a network of components can be represented by a directed graph. In this graph, the components are nodes and the edges correspond to communication channels. In [22] it is mentioned that a network of components may be semantically defined by a single component. Applying the same principle, we can construct a hybrid component starting from a more elemental set of hybrid components. The parallel composition between two or more components forms a net of components. Our composition of hybrid i/o state machines is based on the same principle as the well-known composition of state machines.

Two hybrid state machine $H_1 = (\Sigma_1, Var_1, Init_1, I_1, O_1, Dom_1, E_1, f_1, G_1, R_1)$ and $H_2 = (\Sigma_2, Var_2, Init_2, I_2, O_2, Dom_2, E_2, f_2, G_2, R_2)$ are parallel composed $H_1 \parallel H_2$ to a resulting state machine $H = (\Sigma, Var, Init, I, O, Dom, E, f, G, R)$, where:

- $\Sigma = (Q \times V)$, where $Q = (Q_1 \times Q_2)$, $V = V_1 \times V_2$, $\Sigma_1 = (Q_1 \times V_1)$, and $\Sigma_2 = (Q_2 \times V_2)$,
- $Var = Var_1 \cup Var_2$,
- the initial states are $Init = (Q_{10} \times Q_{20}, V_{10} \times V_{20})$ where $Init_1 = (Q_{10}, V_{10})$ and $Init_2 = (Q_{20}, V_{20})$,
- $I = I_1 \cup I_2$,
- $O = O_1 \cup O_2$,
- $Dom(q_1, q_2) = \{(v_{q_1} \times v_{q_2}) \mid Dom_1(q_1) = v_{q_1},\ Dom_2(q_2) = v_{q_2}\}$,

- $E = \{(E_1 \times E_2) \mid E_1 \subseteq Q_1 \times Q_1,\ E_2 \subseteq Q_2 \times Q_2\}$,
- $f((q_1, q_2), (v_1, v_2)) = \{(w_1 \times w_2) \mid f(q_1, v_1) = w_1,\ f(q_2, v_2) = w_2\}$,
- $G((q_1, q_2), (q_1', q_2')) = \{(v_{q_1} \times v_{q_2}) \mid G_1(q_1, q_1') = v_{q_1},\ G_2(q_2, q_2') = v_{q_2}\}$
- $R(((q_1, q_2), (q_1', q_2')), (v_1, v_2)) = \{(w_1 \cup o_{1d}) \times (w_2 \cup o_{2d}) \mid R((q_1, q_1'), v_1) = (w_1 \cup o_{1d}),\ R((q_2, q_2'), v_2) = (w_2 \cup o_{2d})\}$

This parallel composition can be so generalised for $n \in \mathbb{N}_+$ hybrid state machines $H_1 \ldots H_n$ and is denotated as $H_1 \parallel \cdots \parallel H_n$. In [21] a parallel composition with feedback between two components is defined over their interface behaviours. Instead, we compose the hybrid state machines including also their logical behaviour.

3.4.2 Hybrid I/O State Machine Traces

Considering the syntactic interface in the Definition 12 there is a mapping between the inputs and the outputs variables of the component to the input and output channels. The i/o channels provide and represent the communication between components and with the environment. The evaluation of them in a fixed time $t \in \mathbb{R}_+$ represents the actual state of the system.

In a hybrid i/o state machine we denote with V a finite set of variables, $I \subseteq V$ a finite set of input ports and $O \subseteq V$ a finite set of output ports. A *state* of a hybrid i/o state machine is represented by the value interpretation of all variables in V. Σ is the set of states and a *trace* is a function from $\mathbb{R}_+$ to Σ, considering the evolutions of the component in the reached states, over continuous time.

3.4.3 Example

Damm et al. [48] modelled and verify a variant of the European Train Control System (ETCS) standard[1], which is an automatic train collision avoidance system. We use the artefacts of the ETCS presented in [48] to build a case study, which is presented in Chapter 6 in more detail. We model a train controller modelled with a hybrid i/o machine, based on a variant of the ETCS.

[1] `http://www.era.europa.eu`

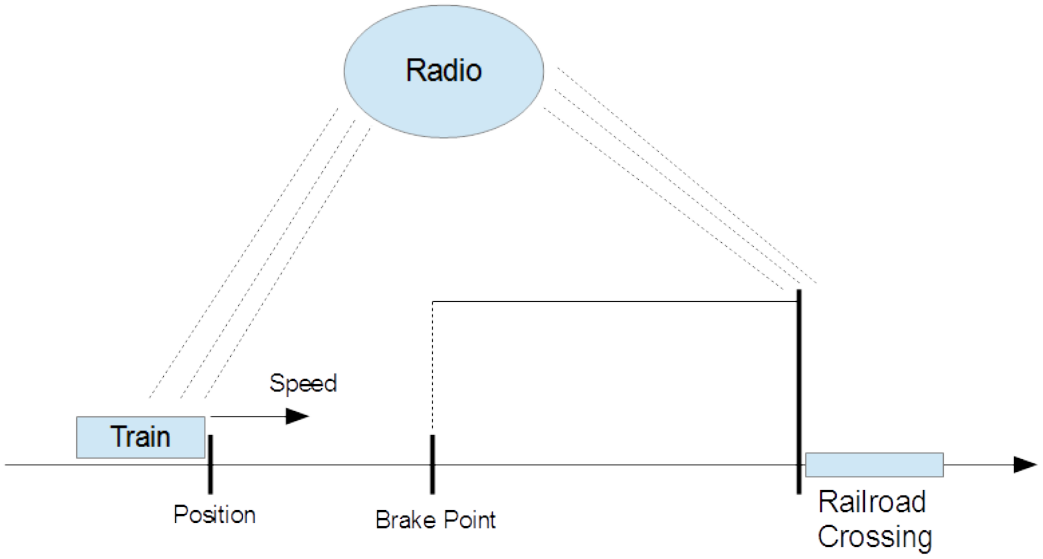

Figure 3.5: Simplified representation of the ETCS train controller system [48].

In this scenario (see Figure 3.5), there is a wayside radio system, which transmits a brake position to the train controller to indicate eventual problems on the railway. The position of the train is one-dimensional; the radio system sends the position at which the train should start to brake, in order to avoid an obstacle, as for instance the railroad crossing in Figure 3.5. If the brake point is equal to the special value -1, the train has no obstacles and so can travel without brake.

We model the control system in the train as a hybrid component with a hybrid i/o state machine as shown in Figure 3.6. The break point is represented by the input channel s, whereas the acceleration, the speed and the current position of the train are output streams. In the figure, continuous streams are depicted by dotted line, the others are discrete streams. The continuous variables z and v, which correspond to output streams, have their corresponding differential equations in each state. These equations are used to simulate the acceleration and the brake actions of the train. There are four transitions, which go from one discrete state to the next, with their conditions to be executed and eventually with the resulting reset assignments to the acceleration. If the train is in the brake point start to brake until the speed is zero or there is no obstacle any more.

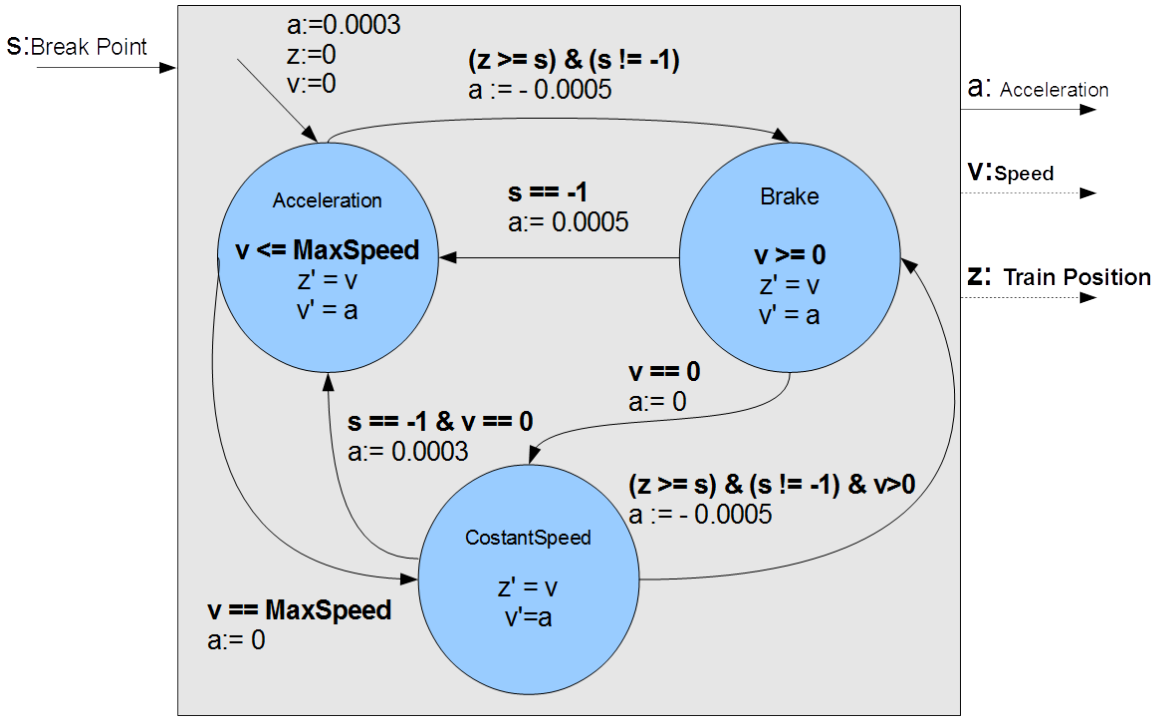

Figure 3.6: Hybrid component for the train brake controller.

In many cases, the data to be elaborated by CPSs can be described by differential equations and so by hybrid i/o state machines, which can be also composed and hierarchical structured through i/o streams. In the simulation and deployment phase, they have not necessarily be executed in continuous time, as explained in Chapter 4.

3.4.4 Comparison with Henzinger's Hybrid Automaton

We introduced hybrid i/o state machines inspired by Henzinger's hybrid automaton [68]. The characteristics of this formalism were most suitable for our purpose compared to other i/o state machines formalisms. We encapsulated the semantic structure of hybrid i/o state machines as logical implementation of our FOCUS components. There are some remarkable differences between hybrid automaton and hybrid i/o state machines: hybrid i/o state machines differentiate between i/o and internal variables, while there is no such differ-

entiation for variables in the Henzinger's automaton. We associate each i/o variable with a discrete or a continuous channel, thus defining the interface of the component. Moreover, in our component architecture, each component is not necessarily implemented by a hybrid automaton, but might have a different implementation as for instance functional specifications. We also aim to have a network of interconnected components representing our systems. In fact, hybrid state machines can be composed, through the communication of i/o channels, i.e. an output port of a component can be the input port of another component. Besides, we can construct systems with a hierarchical composition, where the components are composed inside other components. Therefore, it is possible to abstract the view considering components higher in the hierarchy. An implementation of a component as a hybrid i/o state machine can be provided only on the last level of the hierarchy.

3.4.5 Nonlinear Systems in System Modelling

Our model of hybrid state machines permits the definition of linear hybrid systems. As explained in the next Section (3.5) it is an important challenge to model and verify also the class of nonlinear hybrid systems. In these systems, the vector fields in the continuous state space are calculated by nonlinear ordinary differential equations and, for the discrete transitions, are regulated by nonlinear constraints. We believe that an extension of our state machines to nonlinear dynamic is possible (cf. 3.5.2), by using existing approaches listed in the next section to reduce nonlinear systems to linear ones, but as already motivated in Section 2.3.2 we restrict our topic in this thesis to linear systems.

3.5 Related Work

The increasing interest for embedded systems in automotive, medical, and avionic has determined the realization of many formalisms for hybrid systems. The combination between communication and computation results in distributed hybrid systems, where the system parts communicate with each other by discrete transitions.

3.5.1 Modelling

One of the most important approaches for modelling and analysis of hybrid systems is the Hybrid Automata framework [68], where a discrete transition

system is extended to a system with continuous infinite-state transitions. For these systems, most of the formal verification approaches are based on model checking, employing abstraction or approximations of the original hybrid automata [37]. Concurrent hybrid behaviour is captured in models as hybrid i/o automata [92], and hybrid modules [8]. Hierarchical hybrid systems can be modelled using the tools SHIFT [52], Charon [5] and PTOLEMY [54].

Despite Petri nets have been designed to model and analyse discrete event systems, David and Alla have introduced extensions called continuous and hybrid Petri nets [3]. A continuous Petri net describes the continuous dynamics and the number of tokens are real numbers instead of integers. In discrete Petri nets, the transitions are taken instantaneously, while in the continuous case the transitions are generated as a flow from an external input signal.

A well-known commercial tool is MATLAB Simulink/Stateflow (MSS)[2] and a specific language, supported by various implementations, is Modelica[3], which are used from many industries for the development of CPSs. The modelling language Modelica supports continuous and discrete components based on hybrid modelling techniques. For the modelling of various aspects, there are separate extensions, for instance for collision detection. A limitation of Modelica is a very high level of detail that is required for a physically accurate model. This includes for example, mass, inertia or friction coefficients, which are usually not known in early stages. Moreover, this makes it almost impossible to model different parts of a system with different degrees of abstraction, since the physics must always be respected. Simulink is a tool with a graphical interface for modelling, simulating and analysing of dynamical systems in a model-based environment. Stateflow is an extension of Simulink that introduces tools to model and simulate discrete-event systems within a hybrid environment. MSS is an industrial standard for embedded systems. The continuous dynamics are supported by Simulink blocks that models them as discrete states. Discrete state transition systems on hierarchical state machines are provided by Stateflow charts.

3.5.2 From Nonlinear to Linear Systems

Following the distinction explained in section 2.1, we focus on linear and nonlinear hybrid systems. Linear systems is a class of systems that have a better support for their modelling and verification, instead handling nonlinear systems is more complex. Therefore, are studied reductions of nonlinear

[2] www.mathworks.com
[3] www.modelica.org

systems to linear ones. There are methodologies to translate nonlinear hybrid systems to linear hybrid systems. For instance in [70] two methods are proposed to do such a transformation. The first substitutes the constraints on nonlinear variables with constraints on clock variables (see Appendix A). The second method partitionates the state space into linear regions for each discrete state, and overapproximates in each region the flow field using linear sets of flow vectors. Other authors handle the problem with a piecewise linear approximation of nonlinear functions, which permits to insert a nonlinear object into a hybrid model. For instance, the approach in [131] is based on orthogonal activation function based neural network.

3.5.3 Abstraction of Hybrid Systems

Another attempt to reduce the complexity of hybrid systems is the reduction to discrete systems. This type of transformation is called discrete abstraction. Some of these approaches are based on predicate abstraction [6, 128], others on invariants [98]. An approach based on relational abstractions is presented in [117]; it also supports model checking verification capabilities.

Chapter 4

Dynamic Discrete Sampling

The hybrid components, introduced in Chapter 3, need a real-time execution of their continuous variables. Moreover, we aim to implement them in a supporting modelling tool, where it can be complex to guarantee an error-free and computable simulation, especially in case of models with many variables.

In this chapter, we study the discretization of continuous variables through sampling, in order to optimise the performance and provide a discrete time execution of our models. In control theory sampling is a consolidated and widely used solution to discretize continuous signals, whereas numerical analysis provides the theory to formalize the sampling of variables in differential equations. These foundations are used to formalize the discretization and are discussed as related work in this chapter. We also need a discrete execution because the final produced artefacts, as embedded software or source code, will not be executed on a fully continuous hardware. The introduced sampling algorithms are *dynamic*, that is the sampling period is variable at elaboration time.

4.1 Introduction

In the last years, research was focused on conducting sampling in order to reduce the loads of the system, computing complex differential equations, and to simulate them in a digital execution environment. Two main theories are known in the literature: event-triggered control uses dedicated hardware, and sensors that send information to the controller when events occur. Self-triggered control, where there is a simulation without specific hardware. The system is regulated with sampling techniques made through a controller, usually represented by a finite state machine. We are focused on self-triggered

sampling algorithms for linear systems, in which discrete elaboration steps are as large as possible to guarantee a good level of performance of the digital simulation without specific hardware. In fact, in our algorithms, the discrete simulation step is dynamically adjusted according to the computational capabilities provided by the simulation tool.

We introduce in this chapter a sampled simulation for hybrid i/o state machines, presented in Section 3.4. We approximate the continuous elaboration in the differential equations through a discretization of the continuous variables. In Chapter 3, we compared hybrid i/o state machine with the hybrid state machines defined in [21]. These machines also support sampled executions of the input and output signals, which are implemented the so called *δ step timed state machines*. This state machine performs state transitions only at time equal to an integer multiple of $\delta \in \mathbb{R}^+$, when inputs are read and outputs are written, providing a discretization of the i/o streams. A difference with our sampled i/o state machine is that in our case the period δ is not constant. Instead, it is variable during the simulation, based on adaptive sampling concepts. Moreover, we defined the internal logical behaviour for the hybrid i/o machines. Therefore, our sampling solutions are designed considering this implementation. Our approach studies the numerical solution of the differential equations in the states and adapts the sampling considering the implementation. For instance, after a state transition, the continuous variables might be described by a different differential equation and this information is used in the dynamic sampling algorithms.

4.1.1 Definition

In communication/signal theory, the term *sampling* indicates the operation to approximate an analogue signal with a discrete signal. We use the following notation: an analogue signal is defined by a total function $x : \mathbb{R}_+ \to \mathbb{R}$, whereas for a discrete one is a function $z : I \to \mathbb{R}_+$ with $I \subseteq \mathbb{R}$. Figure 4.1 gives a representation of them. The time interval of the signal sampling is called sampling *period* T and defined as $T = t_{i+1} - t_i$. The sampling may be *periodic* if the sampling period is constant, denoted as Δ, and *variable* if the period is not constant, denoted by a function $\delta(n)$ with $n \in \mathbb{N}_+$.

4.1.2 Control Theory

Control theory is an interdisciplinary branch originating from engineering and mathematics, which handles the behaviour of dynamical systems, which

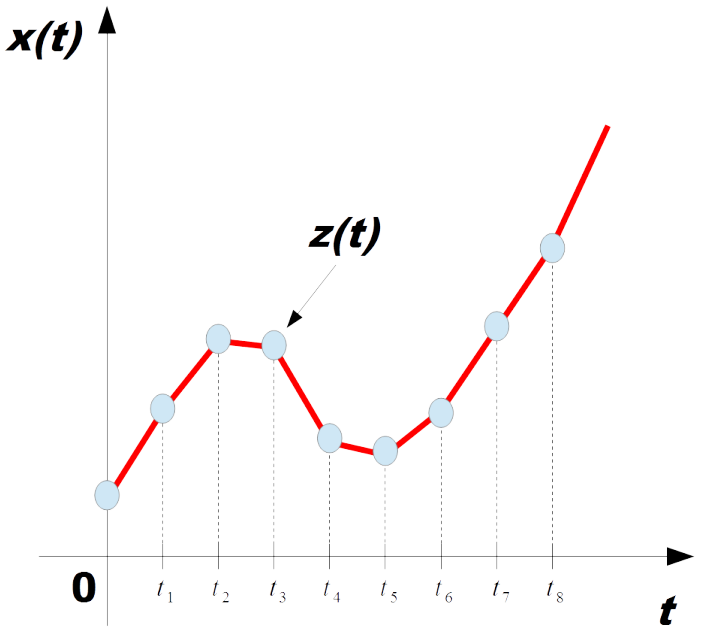

Figure 4.1: Signal Sampling.

can be discrete or continuous time systems. In control theory, the discretization of continuous signals has been widely studied and debated, because in practical applications control systems deal with continuous-time dynamics of linear or nonlinear systems. The increasing performance of computers permits to implement the discretization control algorithms and to evolve the concepts that are the basis of the control theory. In control theory, there are a wide range of approaches and solutions for the discretization of continuous signals. A common approach, called adaptive step size control, guarantees a desired accuracy of the solution while providing reasonable computational performance, which is the mentioned sampling with a variable period. The additional computation for the adaptation of the period is compensated by the overall advantages of the adaptive solutions.

The Shannon's sampling theorem formulates the approximation of a continuous signal in three steps: filtering, sampling, and interpolation. Shannon's work [120] is a first fundamental solution of information theory. In [81] a comprehensive overview on its extensions and applications is given. A survey work about sampling theory in the mathematical literature is presented in [71]. [129] extends the Shannon's approach, considering the signal approximation with consistent measurements. This consistency guarantees

that the output produced by the analysed system is the same if the signal approximation is re-injected into the system. Another extension of the classical Shannon sampling theorem combines this approach with wavelet theories [133]. The authors of [130] propose the approximation and interpolation of a signal using polynomial splines.

The signal processing is equivalent to Shannon's sampling theorem and is implemented in three steps: prefiltering, sampling, and postfiltering. An interesting extension is the generalization from one function to sample to more functions is made for instance in [61]. Another extension of sampling is based on the frame concept that is a generalization from a function to a set of functions [18, 101]. A further research domain is the irregular sampling, where the function is reconstructed from irregular sampled values [63]. In [51] is presented an irregular sampling approach based on nonuniform splines, this kind of solution has been successful used also for practical applications.

In the next sections, we will design a controller, for the sampled hybrid systems presented in the previous chapters, performing a discretised representation of the continuous-time systems. This control technique is referred as Sliding Model Control (SMC). Computer controlled systems utilize mainly these three different discretization methods: Euler, Zero-Order-Hold, and Runge-Kutta method [126, 99]. The Euler method elaborates a correct approximation of systems if the sample period is very small, anyway others methods generally provide better solutions, therefore the Euler method is recommended only if a very simple implementation is needed; The Zero-Order-Hold method provides an exact analytic solution if is used for linear systems; and the Runge-Kutta approach, based on making multiple Euler steps, works particularly well for piecewise smooth nonlinear systems. Euler method approximate solutions with a relatively low accuracy, therefore for real systems, especially for safety-critical ones, this method does not provide an adequate result. In order to have a better approximation, one needs the so-called higher-order approaches. The Zero-Order-Hold method has been used in industrial applications, especially in situations where the state information is discontinuous or intermittent. It uses the most recent information until new values are received, so it is a method more suitable for systems with time-delayed inputs.

The control theory community refers to dynamic sampling also as *adaptive time-stepping*. It is an approximation of ODEs, attempting to reduce the complexity together with the desired level of accuracy. One of the fundamental approaches was proposed by Widrow and Hoff [136]. In [89] a new version of the last mean square (LMS) algorithm with variable step size is presented. It is a gradient search algorithm that changes the time step

according to the size of the prediction error and predefined constant values. In [78] it is argued that adaptive approaches are generally better than non-adaptive ones considering restrictions due to error functions, moreover adaptivity is even necessary to numerically solve ODEs in a reasonably computational time. Also in [135] it has been shown that for linear problems adaptive methods are more efficient. The adaptive time-stepping approach, proposed in [80], determines the period according to an efficiency function related to the Runge-Kutta approximation, where a prescribed global error is their measure of efficiency. Similar approaches, which determine the period considering the accuracy of the approximation of differential equations by Runge-Kutta methods, are presented in [77, 126].

Considering the mentioned solutions and trends in control theory, and the characteristics of the hybrid i/o state machines, we study and elaborate dynamic sampling algorithms aiming to reach an efficient execution and simulation of the models in a supporting tool.

4.2 Dynamic Sampling of Focus Hybrid Components

The idea is to abstract the hybrid component defined in Section 3 in a discrete time system, where the sampling time determines the reachable states of the system. We presented preliminary results of dynamic sampling algorithms for FOCUS hybrid components in [27, 30]. In the continuous time model, the time approximation abstracts the state space in an infinite set, while the discretization of the time reduces it to a countable set.

In Figure 4.2 a simplified interface of a hybrid component is depicted, where $x_i \in I, i \leq n$ are input variables, $y_j \in O, j \leq m$ are output variables, $w_h \in Int \cup O_c, h \leq l$ indicates the internal and output variables, and $v_p \in V, p \leq k$ indicates all the variables. The differential equations are eq_{xy}, whereas res_x and esp_x are expressions associated to the variables. Our approach constructs the abstracted hybrid system in time and not in space, through a discretisation. We build an approach that extend the work in [103], where the authors apply a sampling approach to hybrid automata with a constant period. Our approach introduces a variable period. The variation of the period is determined from two different ideas, explained in the next sections. In the abstraction process of the hybrid component the *hybrid controller*, that is an elaborating object connected directly to the hybrid component plays an important role. It provides the sampled input and reads

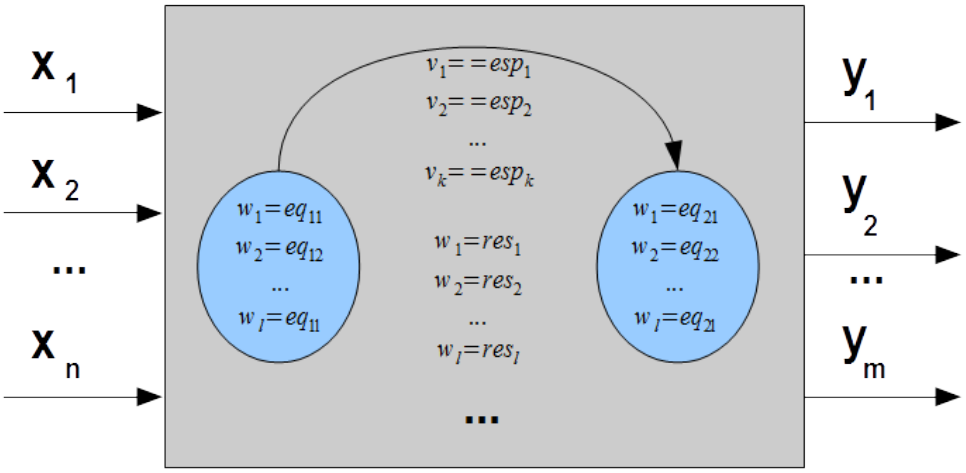

Figure 4.2: I/O Interface and internal elaboration of a hybrid component.

the output produced from the component in order to adjust consequently the period. Considering a system represented by a single component, where the continuous input channels can be physical parts, sensors or the environment, the controller provides these inputs at a calculated and varying time period so that the hybrid component receives input and produces output at discrete time. The controller then again reads the output.

4.2.1 Sampling Architecture

The communication between the components of our architecture is based on typed streams of messages. We introduced hybrid streams for the communication between hybrid components in Definition 10. Hybrid streams are total functions over a set of messages M_i and the streams are infinite sequences over real-time. We now define the architecture of our sampling approach, defining the hybrid component and the controller. We start defining sequences of sampled values from a continuous stream, which represent the input or output data produced in the discretised architecture.

Definition 18 (Sampled hybrid continuous stream). *Considering a finite set M where $\perp \notin M$ and a finite or infinite timed sequence s of elements of M:*

$$s = (m_1, t_1)(m_2, t_2) \dots (m_k, t_k) \dots$$

where $0 \leq t_1 < t_2 < \cdots < t_k < \ldots$, $m_{i+1} \in M$, $t_{i+1} \in \mathbb{R}_+$ *for* $i \in \mathbb{N}$, $i < |s|$. *We associate a hybrid continuous stream* α, *as specified in Definition 10, to a sampled sequence:*

$$\alpha : t \in \mathbb{R}_+ \mapsto \begin{cases} m_{i+1} \in M & \text{if } t = t_{i+1} \text{ for some } i \in \mathbb{N} \\ \perp & \text{otherwise} \end{cases} \tag{4.1}$$

Definition 19 (Sampled hybrid discrete stream). *Considering a finite set* M *where* $\perp \notin M$ *and a finite or infinite timed sequence* r *of elements of* $M \bigcup \perp$*:*

$$r = (m_1, t_1)(m_2, t_2) \ldots (m_k, t_k) \ldots$$

where $0 \leq t_1 < t_2 < \cdots < t_k < \ldots$, $m_{i+1} \in M^*$, $t_{i+1} \in \mathbb{R}_+$ *for* $i \in \mathbb{N}$, $i < |s|$. *We associate a hybrid discrete stream* β, *as specified in Definition 10, with a* $\Delta \in \mathbb{R}_+$ *discrete computation step. Elements* m_j *have to be equal to* $\perp$ *if* $t_j = \Delta * i$ *for some* $i \in \mathbb{N}$. *We obtain a sampled sequence:*

$$\beta : t \in \mathbb{R}_+ \mapsto \begin{cases} m_{i+1} \in M^* & \text{if } t = t_{i+1} \text{ for some } i \in \mathbb{N} \\ \perp & \text{otherwise} \end{cases} \tag{4.2}$$

These time-event functions α and β return at each time instance values of the message set M and $\perp$ in case of absence of messages. We denote with $\mathcal{S}_M$ the set of all such sampled streams.

The syntactic interface of our component (Definition 12) describes a component as a function with m inputs and n outputs. We define a behaviour for this hybrid component in the formalizations in Definition 15. The following formal notations, aim to build a sampling architecture of such a hybrid component H. The sampling architecture is composed of several connected elements that are explained in the following definitions. We start with a sampled hybrid i/o component using a time-event map for each of its i/o stream, where the N_i and M_i are respectively the input and output message sets for i/o channels of H and the $\mathcal{S}_X$ the sampled streams.

Definition 20 (Sampling architecture). *A sampling architecture of a hybrid i/o component* H *is composed by the following map for the sampled component*

$$v_H : \mathcal{S}_{N_1} \times \cdots \times \mathcal{S}_{N_n} \to \mathcal{S}_{M_1} \times \cdots \times \mathcal{S}_{M_m}$$

and a map for the hybrid controller

$$\mathcal{C} : \mathcal{S}_{M_1} \times \cdots \times \mathcal{S}_{M_m} \to \mathcal{S}_{N_1} \times \cdots \times \mathcal{S}_{N_n}$$

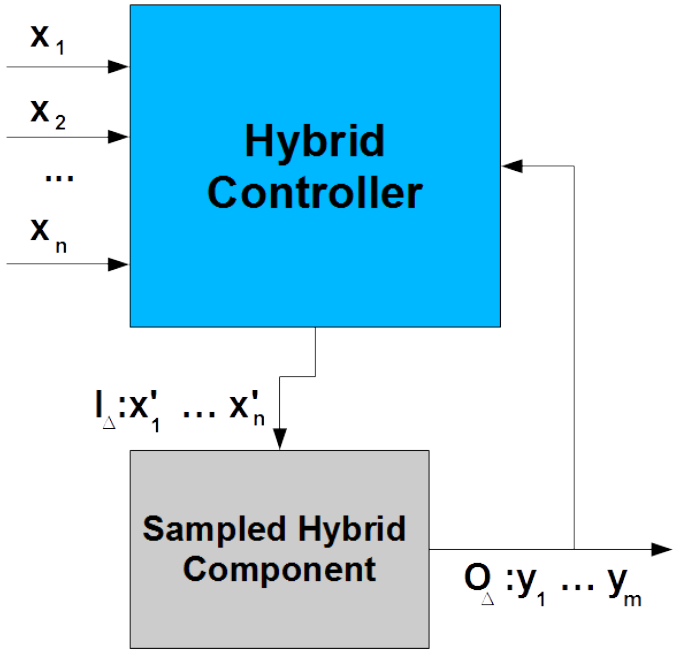

Figure 4.3: Sampling Architecture for hybrid components.

The set of output messages that arrive at the controller are received at sampling intervals, so the controller is activated at a sampling rate $\Delta > 0$ and provides the corresponding input for the next sampled period. This behaviour is specified in the following definition.

Definition 21 (Sampling component). *We associate a sampling controller $\mathcal{C}_H$ to a hybrid component H. The state of the continuous output streams at time $t \in \mathbb{R}_+$ is $o(t)$ (see Definition 17) and $o_\alpha(t)$ is the sampled output produced from the hybrid component and defined according to the Definition 18. The controller returns the sampled input at time t:*

$$\mathcal{C}_H(o_\alpha(t)) = \begin{cases} i(t) & \text{if } t = (i+1)\Delta \text{ for } i \in \mathbb{N} \\ \perp & \text{otherwise} \end{cases} \tag{4.3}$$

In this case, the controller returns the input at regular intervals, not considering the output received from the hybrid component.

We have a representation of the i/o sampled component and its controller in Figure 4.3, where the symbols I_Δ and O_Δ indicate the sampled i/o streams deduced from the streams specified in Definition 18.

4.2.2 Dynamic Periodic Sampling

In the previous section, we built an architecture with periodic sampling, which means, the period rate Δ during the execution remains constant. In this section, we introduce our approach based on a variable period. The basic idea is to adapt the length of the period, in the sampled architecture, to the precision necessary for the elaboration of the values produced from the differential equations and therefore for the continuous output of the component. In case the value of the variable associated with the differential equation changes its value more frequent, then a shorter period it may be needed in order to support the precision of the computation with output. Then no important value changes will be lost. On the other hand, when the values are stable in its evolution and match the actual precision, the system can maintain the actual period length or even make it longer. We define a period mapping $\delta : \mathbb{N} \to \mathbb{R}_+$ that returns the value of the period for the next elaboration step.

Definition 22 (Dynamic sampling controller). *We associate a dynamic sampling controller $\mathcal{C}_H$ with a hybrid component H. The state of the continuous output streams at time $t \in \mathbb{R}_+$ is $o(t)$, $o_\alpha(t)$ is the sampled output produced from the hybrid component and δ is the period mapping.*

$$\mathcal{C}_H(o_\alpha(t)) = \begin{cases} i(t) & \text{if } t = \sum_{i=1}^{n} \delta(i) \text{ for } i \in \mathbb{N} \\ \perp & \text{otherwise} \end{cases} \tag{4.4}$$

4.2.2.1 Approaches for an Adaptive Period

We aim to adjust the period of the hybrid component according to the values produced in the output from the differential equations in the continuous states. We want to apply two different approaches for a dynamic and adaptive period length. Both consider the values of each continuous variable associated with the differential equations defined in the automata states, which are the variables in the set W. Initially the period is set to an initial value d_0.

The first solution considers the slope of the evolution of the variable determined from the active differential equation, which is the equation in the actual discrete state q. The slope is calculated between the actual value of the variable and the value of the precedent elaboration step of the variable. If the slope is smaller or equal to the *acceptance value* then the period remains constant, otherwise is set to half until a minimum value. In the same

way if in a predefined *stabilization time* the slope remains smaller or equal to the acceptance value and the period is less than a maximum value, then the period is doubled, eventually until a maximum value is reached. Each component has a predefined acceptance value that have to be defined according to the variable differential equations, in order to have an effective dynamic sampling. These values can be manually or using a preliminary simulation or verification automatically determined.

The second approach considers some predefined *critical intervals* for each continuous internal or output variable. We have a minimum *acceptance period* for the period that is associated with each interval. If at execution time, the actual value of a variable in W enters in a critical interval and the period is longer than the acceptance period then the period is changed to the acceptance period.

4.2.2.2 Period Variation Based on the Slope

The first approach is based on the calculation of the slope for the functions defined by differential equations and a correspondent acceptance value for it. As shown in Figure 4.4, when the slope between the value at time t_2 and at the actual time t_3 is greater than the acceptance value, the period is halved. The acceptance value have to reflect the interval and the variance reached by the variable values. It has to be defined before the application of the sampling algorithm. At time t_9 since the slope was in the acceptance value for a period of time greater as the stabilization time the period is again doubled. We define formally this adaptive sampling for the architecture defined in Section 4.2.1, where the period Δ was assumed as constant. In this architecture, we associate a hybrid component H with the dynamic sampling controller $\mathcal{C}_H$. We apply our period modification defining the period function δ with the following statements.

Definition 23 (Period variation based on the slope). *Given a hybrid component H, a set of acceptance values L_i and a stabilization times S, with $0 \leq i < |W|$. We define the dynamic variation of the period as follows: we calculate the slope from the second step at time $t = 2d_0$ and for the following steps. The evolution of the system is guided from the differential equation over the variable in $w \in W$, that is $\dot{w} = f(q, v)$, where (q, v) is the actual state and $\delta(n-1)$ is the actual period of the component H for the n-th step of elaboration.*

At elaboration step n for each variable $w \in W$ we use the previous value

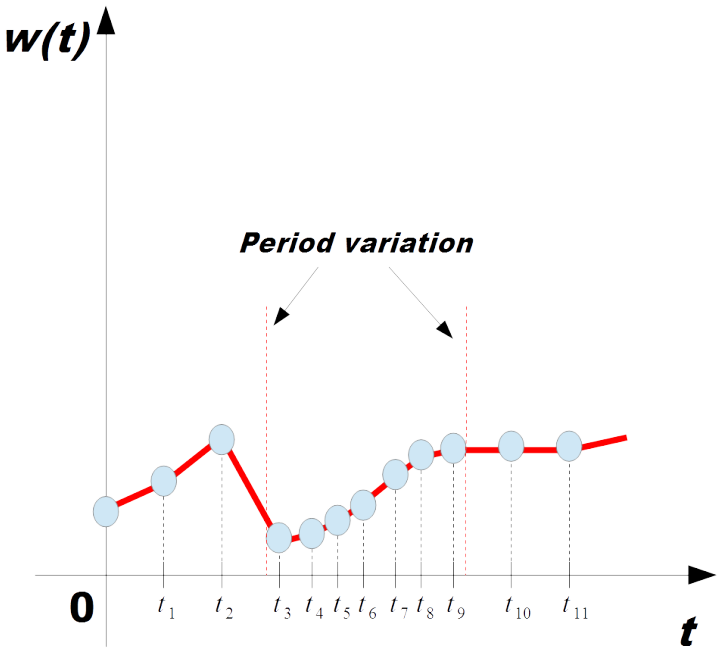

Figure 4.4: Period variation based on the slope.

of w at the computation step $n-1$ to compute the slope:

$$slope(w,n) = \frac{w(n) - w(n-1)}{\delta(n-1)}$$

We update the period for the next computation step only if the slope is greater than the acceptance value L_w; or until this elaboration step for a time interval equal or greater than the stabilization time S the period was constant and smaller than the maximum value d_{max}. This calculation is done for all variables $w \in W$. At the end, if at least one variable need to change the period its actual value is halved, until a minimal value d_{min}. The mapping δ is inductively defined: for an initial state (q_0, v_0) the mapping is set for the first step of elaboration to an initial value d_0:

$\delta(0) = d_0$

$$\delta(i{+}1) = \begin{cases} \delta(i)/2 \ (d_{min} \text{ if } \delta(i)/2 < d_{min}) & \text{if } \exists w \in W \text{ with } |slope(w,i)| > L_w \\ \delta(i) * 2 \ (d_{max} \text{ if } \delta(i) * 2 > d_{max}) & \text{if } (\forall w \in W \ |slope(w,i)| \leq L_w \wedge \\ & |slope(w,i-1)| \leq L_w \wedge \cdots \wedge \\ & |slope(w,i-k)| \leq L_w) \wedge (\delta(i) + \\ & \delta(i-1) + \cdots + \delta(i-k) \geq S) \wedge \text{with } k \leq i \\ \delta(i) & \text{otherwise} \end{cases} \tag{4.5}$$

The definition of acceptance and stabilization time values for the continuous variables has to be based on the range of values that the variables reach in the elaboration. The acceptance value of the slope for a variable is in fact related to the values assumed from the variable. If a variable has a relative small interval of values, the slope should also be not too big, in order to follow the fluctuations with a finer production of values.

We consider the variables independently from the actual differential equation associated with the actual state. In this way when a state transition is fired and the variable obtains a new different differential equation, this change can also determine a high slope with the precedent value. Therefore, when the discrete state is changed the slope determines a change in a smaller period, resulting in a higher production of values. A change in the state is normally a change in the modus of the system, so a better precision can guarantee a reactive simulation. In order to derive acceptance values and stabilization time, the hybrid component may be verified with a formal verification technique, where it is checked, through value invariants, the maximal variable value ranges.

4.2.2.3 Period Variation Based on Critical Intervals

The second idea for the period variation is conceived around predefined value intervals. During the simulation of the hybrid component, the period varies according to the current values of the variables in W, with respect to predefined critical intervals, as depicted in Figure 4.5. If the value of a variable is in a critical interval and the actual period is greater than the acceptance value associated to the interval, then the period is set to this value, for instance the steps at time t_4 and t_{10}. On the other hand, if the value of a variable is not in a critical interval any more, then the period is set again to the maximum value, as for instance at time t_7. The critical intervals are defined considering component requirements.

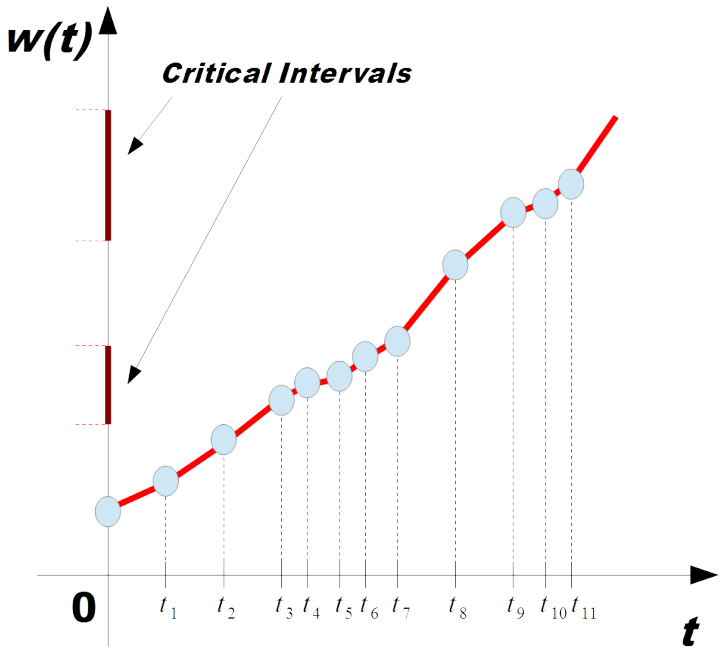

Figure 4.5: Period variation based on the critical intervals.

The sampling period for the next computation step is calculated for each output or internal variable, and finally set to the minimum period of the values calculated for each variable.

Definition 24 (Period variation based on critical intervals). *Given a hybrid component H, a set of critical intervals K, a set of period times P, with $|K| = |P| = |W|$ and $k \in K$, $p \in P$, $k, p \in \mathbb{R}_+$, we build the dynamic variation of the period as follows: for the initial states (q_0, v_0) the period is set to the maximum value d_0, at the second step of elaboration (at time $t = 2d_0$) the dynamic choice of the period starts. The evolution of the system is guided from the differential equation over the variable in $w \in W$, that is $\dot{w} = f(q, v)$, where (q, v) is the actual state.*

The modification of the actual period is made analysing the actual value of the variables in W, which is if a variable w is in a critical interval K_w, and then the corresponding P_w acceptance period is the necessary value of w for the period. It is finally chosen the minimum acceptance period between the variables that are in a critical interval. The period map δ is inductively over the elaboration steps as follow determined:

$$\delta(0) = d_0$$

$$w_{\delta_{i+1}} = \begin{cases} P_j & \text{if } w(i) \in K_j, \text{with } 0 \leq j < |W| \\ \delta(i) & \text{otherwise} \end{cases} \tag{4.6}$$

$$\delta(i+1) = min\{w_{\delta_{i+1}} \mid w \in W\}$$

Critical intervals have to be established before the execution of the sampled hybrid components and they should define appropriate sampling periods (acceptance periods). This information is tightly bound to the needs and real-time restrictions of the variables; therefore, they can be derived from the formal requirements of each variable of the systems or from the modelled hybrid component considering the implementation behaviours.

4.2.3 Approximation of Differential Equations

In this section, we provide a solution based on numerical mathematics for the execution of the dynamic sampling algorithms presented in the previous sections. This solution is necessary to discretise and approximate the differential equations defined in each discrete state of the hybrid i/o state machines. As depicted in Figure 4.2, in each state there are first order differential equations associated with internal or output continuous variables, according to the vector field function $f : Q \times V \to W$ (Definition 15). In linear hybrid i/o machines, differential equations define the behaviour of continuous variables $w \in W \cup O_d$ in each discrete state, resulting in a system of n differential equations with n real variables.

There are many classes of numerical methods for ODEs, a well known classification is in the following two classes: the *one-step methods* that use one initial value at each step and the *multistep methods* that use more values to compute the solution [64]. One-step solutions calculate the point y_{i+1} by using only the point y_i as initial value and do not use any previously computed solution points. Instead the multistep methods use the $k + 1$ previously computed solution values $y_i, y_i - 1, \ldots y_{i-k}$ to calculate the next solution point y_{i+1}. Generally, multistep methods are slower but produce more accurate solutions than one-step methods.

We presented hybrid i/o state machines as a general modelling paradigm for CPSs. Considering the heterogeneity of the modelled systems is it difficult to estimate the complexity of the typical differential equations, so we focus for a one-step method in this work.

In Section 4.1.2, we listed the most commonly used one-step approaches to calculate the discretization of differential equations. The Runge-Kutta

method is a higher order approach, compared to the other two. This approach works best with our i/o linear hybrid state machines. In fact, the limitations of the Euler method and the characteristics of the Zero-Order method are not well suited for our models. The Runge-Kutta method is a classical fourth-order method, which guarantees a perfect balance between accuracy and complexity. Moreover, this method has many scientific applications and can work very effectively as ODE integrator, especially when combined with adaptive stepsize algorithms [109].

Given a differential equation $\dot{w} = f(q, v)$ for a continuous variable $w \in W$, where without loss of generality $w(t) = v_j \in V$ in $v(t) = (v_0, \ldots, v_n)$, and a period defined by the function $\delta : \mathbb{N} \to \mathbb{R}_+$, then the value $w(\delta(i+1)) = f(q(\delta(i+1)), v(\delta(i+1))) = w_{i+1}$ is calculated with the classical Runge-Kutta method as follows:

$w_i = f(q(\delta(i)), v(\delta(i)))$
$h = \delta(i+1) - \delta(i)$

$W_1 = w_i$
$W_2 = w_i + h/2 f(q(\delta(i)), v^j(\delta(i), W_1))$
$W_3 = w_i + h/2 f(q(\delta(i) + h/2), v^j(\delta(i) + h/2, W_2))$
$W_4 = w_i + h f(q(\delta(i) + h/2), v^j(\delta(i) + h/2, W_3))$

$w_{i+1} = w_i + h/6(w_i + 2f(q(\delta(i) + h/2), v^j(\delta(i), W_2)) + 2f(q(\delta(i) + h/2), v^j(\delta(i), W_3)) + f(q(\delta(i+1)), v^j(\delta(i+1), W_4)))$

where $v^j(t, x) = (v_0, v_1, \ldots, v_j - 1, x, v_{j+1}, \ldots, v_n)$ with $v(t) = (v_0, \ldots, v_n)$ and V has cardinality n. A computer can execute this sequence of steps and there are already many implementations, as for instance in [109], where the algorithm has been implemented in C programming language. It requires more computational time than the Euler's method, but provides also more accurate results. An advantage of Runge-Kutta is that it can work effectively also with long sampling periods. The classical Runge-Kutta method presented above is an approximation for the single variable w, but can be used and generalised to find an approximate solution of a system of differential equations.

4.2.4 Sampled-Data Model vs Event-Triggered Model

The introduced hybrid components and hybrid i/o state machines in FOCUS (cf. Chapter 3) define in the FOCUS modelling theory a model of computa-

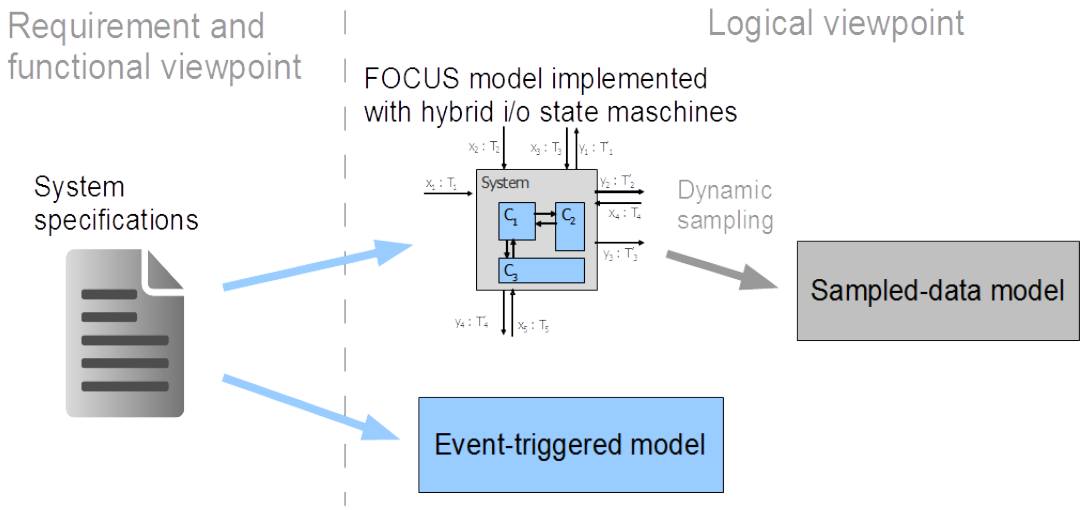

Figure 4.6: Two alternative implementations of the same system: event-triggered model and FOCUS model implemented with hybrid i/o state machines, which uses at run-time a sampled-data model.

tion in continuous time. The simulation of hybrid i/o state machines through dynamic sampling algorithms in a modelling tool, as for instance AutoFOCUS 3, results in a discrete time model of computation at run-time. In this section, we compare the sampled execution of a FOCUS continuous time model implemented with hybrid i/o state machines and a generic event-triggered model, both based on the same system specifications.

The compared models are shown in Figure 4.6, where the artefacts are classified according to the SPES matrix (cf. Section 2.4). As illustrated in Figure 4.6, a FOCUS model described by hybrid i/o state machines when is simulated by our sampling algorithms originates a sampled-data model. Generally continuous time embedded systems are implemented on digital hardware, which computes the signals at discrete time using sampling algorithms determining a sampled-data control. Event-triggered systems adapt the communication among components according to current needs and minimising the signal exchange in order to fulfil required performances. Numerous approaches have shown that event-triggered models scale down the computational time and the exchange of signals, with a little impact of the control quality [11, 134, 67]. Event-triggered models are an alternative to sampled-date ones when certain performance requirements have to be satisfied, as for instance the reduction of the computational or energy needs

[96]. A comparison from the control theory point of view is presented in [2]. This work concludes that often it depends of the kind of application which model is more suitable. Sampled-data models offer for distributed control, safety-critical and large-scale automotive systems in general more interesting performances. On the other hand, the design complexity and the inflexibility of sampled-data architectures can be not suitable for other application.

We consider a modelling example, that is, the control software of a train braking control system presented in Chapter 6. We analyse for both event-triggered and sampled-data models the value approximation of the continuous variables. We model the control software of the braking train system using the introduced hybrid i/o state machines in FOCUS. This system can be also modelled (with a probable loss of precision) using an event-triggered model. We evaluate if the sampled-data model has the same characteristics of the event-triggered model. Hybrid i/o state machines in FOCUS define the behaviour of the continuous variables with differential equations specified in each state. During the simulation, the introduced dynamic sampling algorithms approximate a value for the continuous variable at discrete time intervals. The period determines the rate at which the values are calculated and it varies dynamically during the simulation in order to improve the accuracy of the approximation. In the event-triggered model, the continuous variables described by differential equations are modelled with a sort of discreted differential functions. The values of the variables are calculated in discrete time simulation steps, with a predefined length. The simulation step for the event-triggered model can be chosen arbitrarily short, in order to reach the desired precision, and it remains constant at execution time. On the other way round the sampling algorithms for hybrid i/o state machines change the period dynamically. Therefore, the values calculated for the continuous variables by the sampled-data model have a variable simulation step. For this reason, the two models provide a different approximation of the continuous variables: the sampled discrete model has the advantage to change dynamically the simulation step, according to suitable criteria. The FOCUS model implemented by hybrid i/o state machines respect to the event-triggered model has advantage to change dynamically the precision of the approximation. When both the implementations are modelled with the FOCUS theory, they have i/o channels respectively in the I and O sets, which define a formal syntactic interface: $I \rhd^{+} O$. This formal typed syntactic interface offers support for validation techniques of requirements specified on the i/o channels. This way, traces of messages produced by sampled-data and event-triggered models can be compared and validated against formal requirements. It is also possible to verify the FOCUS models, implemented

by hybrid i/o state machines model, and event-triggered models with model checking techniques (see Section 5). Anyway, formal properties defined on variables guided by differential equations in hybrid i/o state machines cannot be verified, with the same semantics, by model checkers on event-triggered models. In fact, in the first case the variables guided by differential equations are verified in a continuous time procedure, meanwhile in the second case these equations are already discretised in the event-triggered model and therefore also their verification will be with a discrete time procedure.

A certain precision of the approximation of differential equations or the minimisation of the computational and energy efforts determine the decision for the developers of a continuous time system between a sampled-data or event-triggered implementation. A requirement formalization phase with artefact based on formal methods facilitate this decision. In fact, the mentioned technical requisites need to be specified and described in a precise and rigorous manner. Whether a time-triggered or event-triggered is appropriate for a continuous system depends on its scope and application domain.

4.3 Dynamic Sampling of Component Architectures

In FOCUS, the basic modelling element is a component, from which a hierarchical architecture can be constructed. This concept is a direct consequence of the application of the FOCUS modelling principles. Any system to be developed must be described as a top-level component and is then broken down into a hierarchy of subcomponents. A component has a syntactical interface described as a set of input and output ports (i/o-ports), as presented in Section 2.2. These ports are used to interconnect several components for communication. Ports are assigned data types to restrict the messages allowed to be transmitted. Besides this black-box view, we have two alternatives to describe the actual behaviour of a component. By applying decomposition, we describe the component by a network of subcomponents. In this network, the i/o-ports of the subcomponents and of the hosting component itself are interconnected. Alternatively, a component is defined by some operational behaviour description. For instance, depending on the kind of behaviour a description may be e. g., in terms of i/o-automata (for adaptive behaviour) or tabular i/o-specifications (for control-intensive behaviour), which are both implemented in the AutoFOCUS 3 tool.

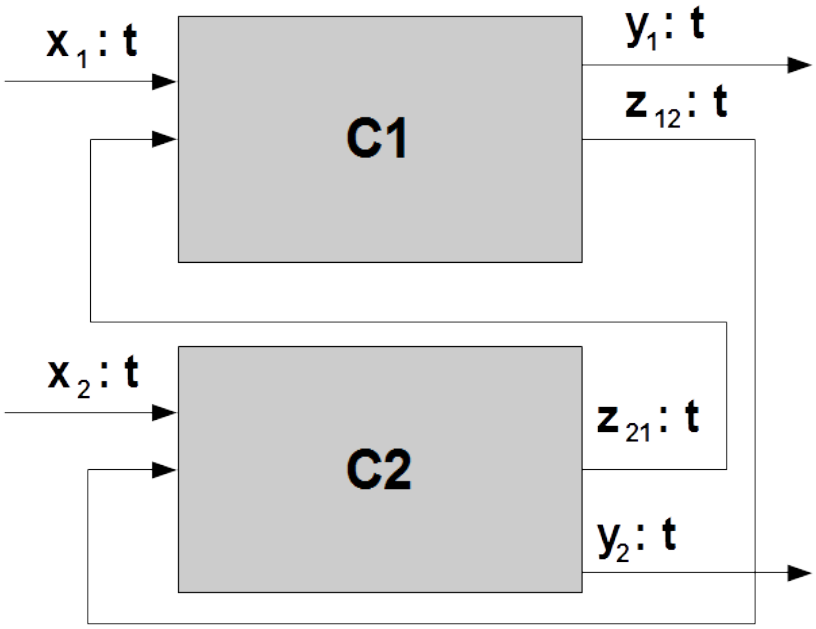

Figure 4.7: Mutual component composition.

4.3.1 Component Composition

So far we described our sampling architecture considering one component in our model. We now consider component composition, where the components are connected to each other, in order to form a network. In this network, each component has its sampling controller connected as described in the precedent section. In order to compose several behaviour functions or automata, we introduce the notion of typed channels. An output channel may be connected to an input channel such that the owning components communicate over this shared channel. This communication is unidirectional with exactly one sender and one or more receivers. Moreover, in our system model components can be nested in a hierarchical way, where the implementation of the nested component is defined on the lowest level. This way, the composition can be also abstracted, with the highest component, which represents the interface of the model.

A typical example of a composed model is the ”flip-flop” model, where the composition of two components is done through an input and an output port of a component that are respectively the output and input ports of the other component and vice versa, as depicted in Figure 4.7. Let us assume that

all i/o variables are continuous and that at the start of the elaboration the period is set to the maximum value d_0. During the elaboration, the variable y_1 requires a change of the period because the calculated slope of the variable is not in the acceptance range, so the period for the next step is set to $d_0/2$. Consequently, the component $C1$ needs a shorter period and therefore the component $C2$ has to adapt its period, because $C2$ provides the input with the variable z_{21} to $C1$. Analogously, if component $C2$ modifies the period to a shorter one, the component $C1$ has to adapt.

It follows that in our architecture a component that receives input from other components must be able to communicate needs for a shorter period to them. That is particularly important if the component is receiving inputs from other components at a longer period respect to its needs. After a computational step, we calculate the period for each component for the next step. Independently from the used method, each component has to check if the period of the components, which are connected to its input ports, are equal or shorter than its actual period. If a component has a longer period, it has to adapt it to the required period of the connected component. If the variable comes from the environment then it must be a continuous signal and the period variation straightforward. Otherwise, if the input variable comes from another component and its period is longer than the period required, the controller sends a request to the component that now has to consider and adapt its period not only based on its variables but also based on the required value.

4.3.2 Time Restrictions

We now consider two restrictions necessary for a working implementation of the sampling algorithm. We halved the period in the variation based on the slope and we considered specific period values for the variation based on critical intervals.

The first restriction is in both cases at run-time when the computational load for the simulation and for computing the periods is it too much to guarantee enough computational time at the CPU of the system. Therefore, during the simulation it is necessary to control whether the actual execution has enough resources to maintain a real-time elaboration. If the hardware cannot afford shorter periods for some components then the change will not be carried on and should be annotated.

The second time restriction comes from requirements for output ports. In a model, we can have some precise temporal requirements for the output

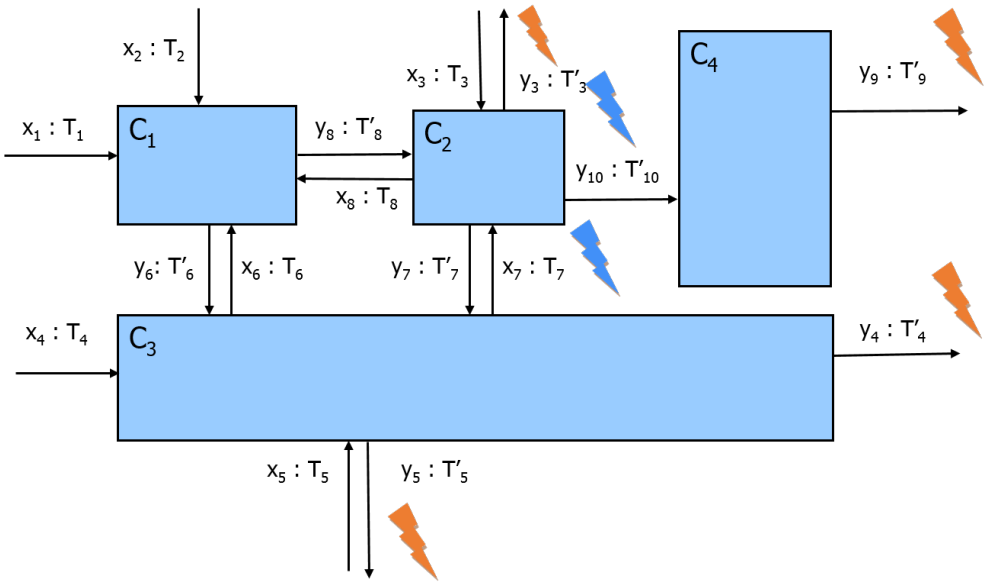

Figure 4.8: Time restriction at output ports of the hybrid component architecture.

ports of the system, that is, the components that provide the output ports to the environment as shown in Figure 4.8, for the output streams y_3, y_4, y_5, and y_9. If the requirements set a minimum rate for some output port, this has to be respected and therefore can affect the simulation for the connected components as explained in the first restriction.

4.4 Implementation

We present, in this section, our preliminary tests and considerations for a future implementation of our sampling algorithms in the modelling tool AutoFocus 3. We study the analysis and simulation capabilities of MATLAB Simulink/Stateflow (MSS) for hybrid systems. MSS is one of the most important industrial modelling tool for CPSs. We test the sampling environment of Simulink/Stateflow with a case study representing a Focus hybrid component. We implemented prototypically our sampling algorithms in MATLAB and we selected some models for a preliminary analysis of our implementations. We used MSS version R2014a for our experiments and implementations.

4.4.1 Dynamic Sampling in MATLAB Simulink/Stateflow

We already introduced in Section 3.5 the toolbox MSS for the design of dynamic and embedded systems. Simulink/Stateflow is a model-based data flow graphical tool integrated in the MATLAB environment, which can drive the elaboration with MATLAB or be scripted from it. Simulink/Stateflow support modelling and simulation of dynamic systems and is an industrial standard in control theory and signal sampling.

There are many dynamic and static sampling algorithms in the simulation environment of Simulink/Stateflow. We analysed widely used ODE solvers in the following case study. The selected ODE solvers are similar to our sampling solutions; in fact, some of them are also Runge-Kutta based methods. We used the toolbox MSS for two purposes: (1) a comparison between modelling and verification of FOCUS hybrid component in MSS and in AutoFOCUS 3 (further details about the verification are in Section 6.3); and (2) an initial test of our sampling solutions against the most used dynamic sampling algorithms in MSS.

We consider the following dynamic sampling solvers: *ode45* integreted in MSS, which provides fourth and fifth order formulas and is based on an algorithm of Dormand and Prince; *ode23* that is an implementation of a third-order Runge-Kutta method based on an algorithm of Larry Shampine and Przemyslaw Bogacki. In the MSS documentation, it is reported that the *ode45* solver should be applied as a first try for most problems. It also is worth a try of the *ode23* that can be even more efficient than the *ode45*. We made tests also with the following constant sampling algorithms: *ode4* that is the fourth-order Runge-Kutta method; *ode3* that is the fixed step version of the solver *ode23*.

4.4.1.1 Modelling and Simulation of Hybrid I/O State Machine in MSS

In Section 3.4.3, a simplified train brake controller is modelled using FOCUS in a hybrid i/o state machine. It is inspired to the European Train Control System (ETCS) introduced in Chapter 6. We modelled the hybrid component representing the train brake controller (see Figure 3.6 in Section 3.4.3) in a semantic equivalent Simulink/Stateflow model. The differential equations defined in the hybrid i/o state machines are converted in Simulink in *integration blocks*; for each differential equation there is an integration block. In the system, there are two differential equations: one for the speed and one for the

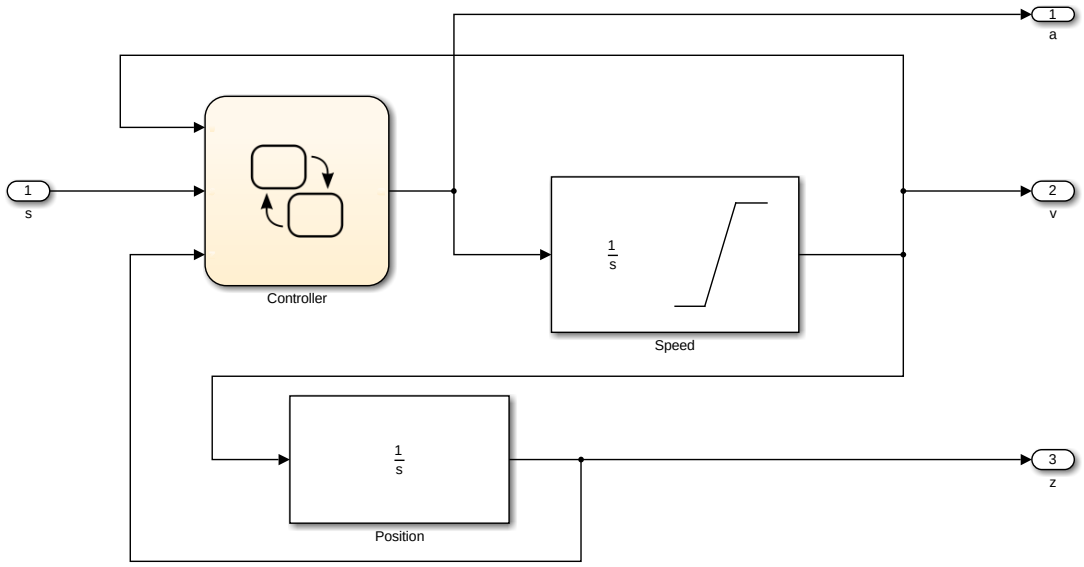

Figure 4.9: The Simulink model of the hybrid i/o state machine in FOCUS presented in Section 3.4.3.

position of the train. The i/o streams of the hybrid component are modelled in Simulink with i/o ports. The logical states of the hybrid state machine are modelled with a *stateflow diagram*. The resulting Simulink model is depicted in Figure 4.9, meanwhile the stateflow diagram (representing the logic of the component) is in Figure 4.10.

We simulate this model in a scenario where the breaking point is a constant value 30 (Km) with a simulation interval of 3600 seconds. We tested different ODE solvers with static and dynamic sampling. We cannot simulate the model in with a fixed (or variable) sampling period and a discrete solver for the differential equations, because the model contains continuous states. We executed the simulations with the following ODE solvers (with default parameters): *ode45*, *ode23*, *ode4*, and *ode3*. The results with the *ode23* are more accurate than the *ode45*. The results with the constant period solutions are very similar and the tool was unable to derive a fixed step size, because we do not specify any sampling period in our model. Therefore, the system chooses a period based on the length of the simulation, in our case for a length of 3600 seconds the period was of 72 seconds. In the simulation with *ode45* the train starts to brake at the km 34.30, despite it should begin to brake at the km 30. In the constant step simulation, the train start to brake

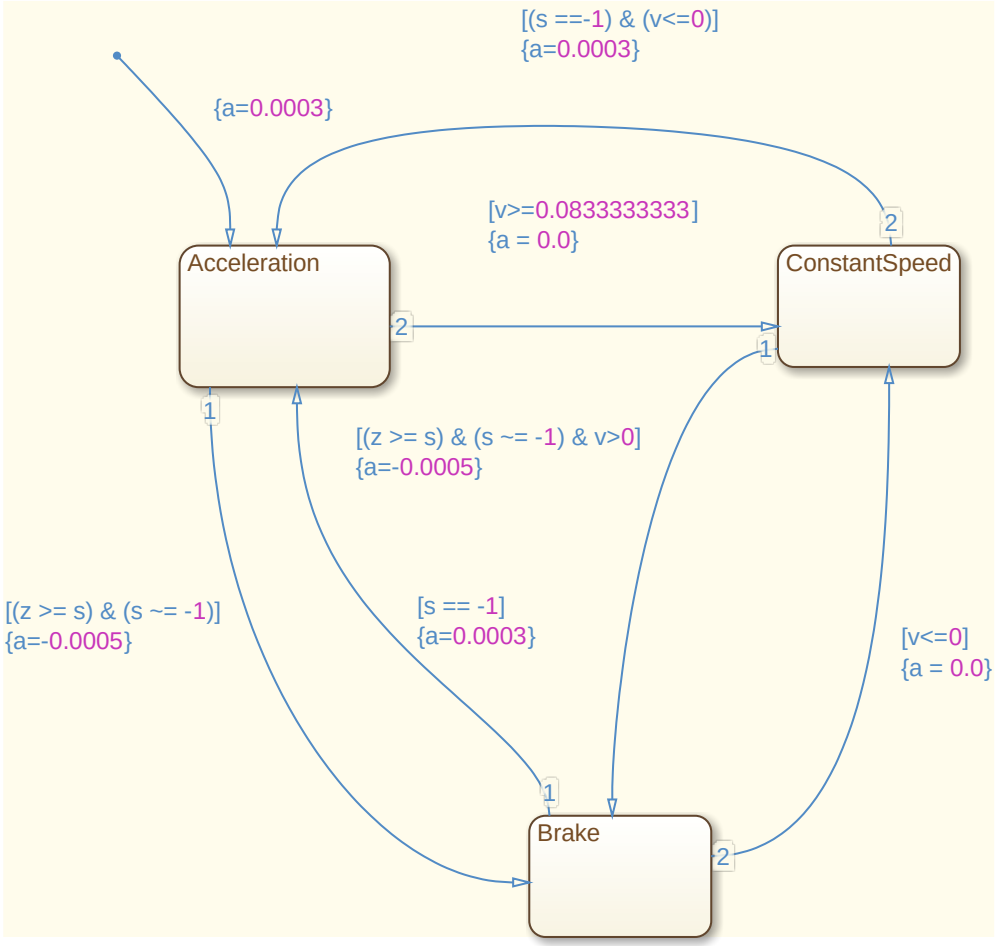

Figure 4.10: The stateflow diagram corresponding to the logical behaviour of the hybrid i/o state machine in FOCUS presented in Section 3.4.3.

at the km 31.10. If one would like to observe what happen at the braking position (30 km), the precision of the simulation need an adjustment of the sampling algorithms. We believe that this adjustments might be not intuitive as defining the following parameter for our solution based on critical intervals: a suitable critical interval for the train position variable, as for instance perform a sampling with a simulation step of 10ms when the variable is in the interval 25 (km) and 35 (km). Anyway, our algorithms and the ODE solvers integrated in MSS require a preliminary analysis of the models to be optimal configured.

4.4.1.2 Implementation of Dynamic Sampling Algorithms in MATLAB

We programmed in MATLAB our sampling algorithms as ODEs solver functions. As stated in Section 4.2.3 we use for the approximation of the continuous variables the fourth-order Runge-Kutta method. The resulting two MATLAB functions have different parameters and can be executed over continuous functions, as made by the MATLAB functions *ode23*, *ode45* etcetera. We call *ode4slope* the implementation of the algorithm based on the slope; and *ode4interval* the implementation of the algorithm based on critical intervals. Unfortunately, the implemented algorithms in MATLAB cannot be utilized as ODE solvers in the Simulink/Stateflow environment. We cannot therefore test our sampling solutions using Simulink models. Anyway, we can test them on differential equations defined in the MATLAB language.

We select a derived form of the Mathieu's differential equation for a preliminary evaluation of our solutions. It is a system of two linear differential equations representing an oscillating trigonometric function. Mathieu functions are special functions useful in mathematics for treating a variety of problems [137]. The MATLAB code of the differential equation we used for testing our algorithms is shown in Listing 4.1.

Listing 4.1: MATLAB Code for Mathieu's differential equation.

```
function r = test(t,x)
 % delta
 d = 0.1045;
 % epsilon
 e = 0.0048685;
 % result (2-dimensional)
 r = [x(2); (-d - e * cos(t)) * x(1) - 0.7 * d * abs(x(1))];
end
```

We simulated the derived form of Mathieu's differential equation, which was mentioned in the MSS community website MATLAB Central[1], in a time interval of one hundred thousand seconds, with the following ODE solvers: *ode45*, *ode4slope*, and *ode4interval*. We configured *ode45* with default parameters and our solutions with an initial period of 0.6, minimal period of 0.2 and maximal period of 2.0 seconds. We set for the *ode4slope* algorithm the following parameters: an acceptance value of 0.55 for all variables and a stabilization time of 1.0 second. Meanwhile, we set for the *ode4interval* algorithm the critical interval $[-1.0, 1.0]$ for the variable $x(1)$ and $[-0.5, 0.5]$ for the variable $x(2)$ with an acceptance period of 0.5 seconds for both variables. These parameters are chosen to have a similar number of simulation steps of the algorithm *ode45*. The results of the simulation in a time interval of 100.000 seconds of the differential equations described in the above MATLAB function *test*, with the mentioned parameter values, are summarized in Table 4.1. The values reached by the variables $x(1)$ (in blue) and $x(2)$ (in green) in the simulation with the different sampling algorithms are graphical represented in Figures 4.11, 4.12 and 4.13.

Sampling Algorithm	**Simulation Steps**	%	**Computation Time**	%
ode45	157.089	100	4.2665 seconds	100
ode4slope	196.768	125	4.0702 seconds	95
ode4interval	182.531	116	3.7980 seconds	89

Table 4.1: Simulation results in terms of absolute/relative simulation steps and computation time.

As shown in Table 4.1, our algorithms *ode4slope* and *ode4interval* calculate more simulation steps for the same simulation time interval (i.e. 16 - 25% more compared to *ode45*), while requiring less overall computation time (i.e. 5 - 11% less compared to the *ode45*). This performance improvement is due to the fact that we do not rely on estimating the approximation error. As stated before, to us the approximation error is less important because we aim at supporting different needs. Our solutions have the following advantages:

- The overall computational performance is better and generally more stable and predictable performance can be guaranteed with respect to existing approaches. Unfortunately, to the best of our knowledge no public benchmark sets are available for comparing adaptive sampling

[1] `http://www.mathworks.com/matlabcentral`

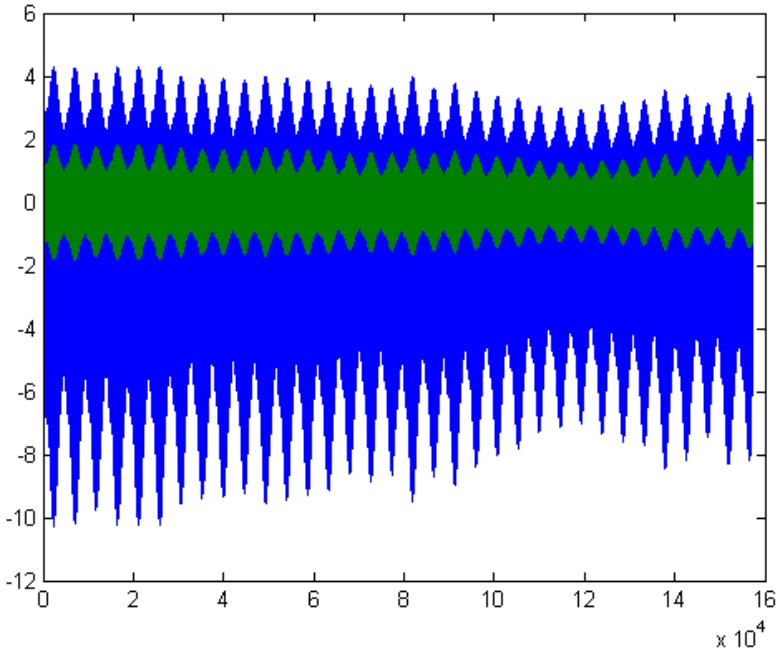

Figure 4.11: Simulation of the function *test* with the algorithm *ode45*.

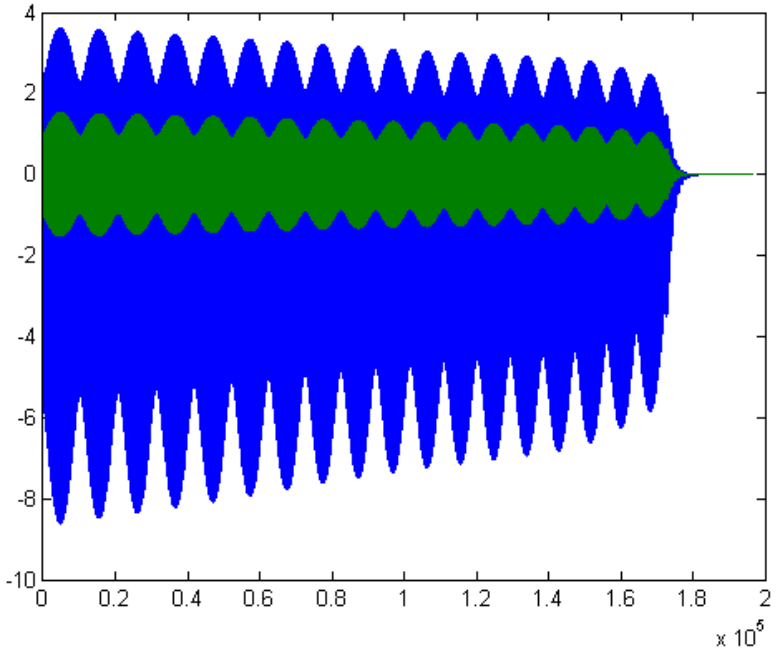

Figure 4.12: Simulation of the function *test* with the algorithm *ode4slope*.

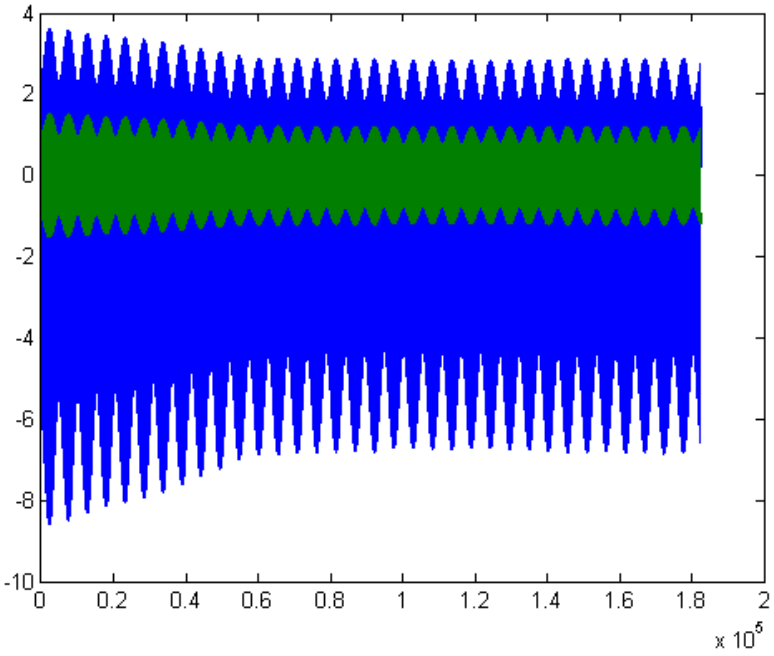

Figure 4.13: Simulation of the function *test* with the algorithm *ode4interval*.

algorithms with respect to performance. The initial performance evaluation with the presented example, which due to its simplicity can be considered a good representative benchmark, shows promising results for the scalability of our solution. In particular, scalability is interesting for systems with a large number of elements, where a simulation within reasonable amount of time is more important than the precision of the simulation itself. For instance, simulation performance might be more important than simulation precision in early development phases of systems with a very large number of components.

- An advantage of our approaches is the direct setting of period lengths in desired variable value intervals and the setting of acceptance values for the slope to determine the desired change of the period. Consequently, it is possible to initialize simulations with a desired precision. For instance, a possible scenario is the validation of a system, defining a shorter period when error states are reached. Parameters for the sampling algorithms could be directly derived by system requirements.

- In next chapter, we study an approach to formally verify hybrid i/o state machines with the model checker HyCOMP. We check formal as-

sertions over continuous variables modelled by differential equations. Formal assertions, as invariants, could be verified in an automatic or assisted way to guide the determination of parameters, which are necessary for our sampling algorithms. The verification process of HyCOMP consider an infinite time horizon. Therefore, our solutions can use information about the behaviour of the continuous variables in an infinite time interval, which is an advantage with respect to existing sampling algorithms as for instance in the MSS environment. In fact, they usually consider the variable behaviour in finite time intervals.

4.4.2 Dynamic Sampling in AutoFocus 3

The modelling framework provided by AutoFocus 3 permits to design systems that have a discrete elaboration, where each component synchronously performs a discrete step of computation. We defined models based on hybrid components and in this chapter sampling algorithms for their simulation. We plan to extend AutoFocus 3 models permitting the definition of hybrid automata as behaviour for the components. The discretised execution for the continuous variable will be supported with the dynamic sampling solutions introduced in this chapter. Preliminary results obtained from the prototypical implementation in MATLAB, which are presented in the precedent section, are promising. Therefore, an implementation of such dynamic sampling algorithms in AutoFocus 3 is desirable.

4.5 Related Work

The simulation of systems in a digital environment or tool, which involve and necessitate real-time continuous elaborations, is an important topic. In the last decades, computer science increased its importance in control theory. Also sampling algorithms are an integration of control, communications theory and theoretical computer science. A fundamental approach to control hybrid systems is based on finite-state approximation. For instance, the works [62] and [17] approximate the systems using observations. In the first approach, the continuous dynamics is restricted to the output events, with discrete-valued input and output signals based on observed threshold events. In the second work, a discrete abstraction for robot motion planning and control is presented. The finite-state abstraction is constructed by dividing the state-space into regions. In our approach, we abstract by sampling the system in time and not in the state space. This is similar as [103], but we

introduce a variable sampling period, based on the slope correspondent to the function associated to the continuous variables and on predefined critical interval values and periods for the continuous variables.

Chapter 5

Verification of Focus Components

During the last years, the functionality of safety-critical systems has grown dramatically. This includes software as well as hardware, and holds at the same time for reliability requirements. Therefore, we are now facing an enormous system complexity. Due, to the growing complexity, the possibility of unintentionally added defects, as for instance forgotten control conditions, grows as well. In consequence, system failures may lead to a considerable loss of money due to warranty costs or even—in the worst case—endanger human lives. Thus, the need for well-defined development theories, languages, and tools naturally follows. Their comprehensive use permits the construction of more reliable systems even though system complexity, is supposed to grow in the future.

In this chapter we introduce the support for analysis techniques in the model-based tool AutoFocus 3 [31], also supporting the hybrid nature of the models introduced in the Chapter 3. The method implemented for checking the systems is based on a formal verification approach.

5.1 Analysis Techniques

Nowadays, computers are widespread and used to control various safety-critical systems like automobiles, aircrafts, satellites, medical devices, etc. Software and hardware operating within such systems is complex. It consists of many modules or programs, each in the magnitude of thousands to millions lines of code. The existence of errors may have severe consequences. For instance, consider the famous fault occurred in 1994: the “Pentium bug”,

which caused Intel to recall faulty chips and take a loss of about $475 million [45]. In June 1996 another important program failure was in the Ariane 5 rocket: a computation exception occurred during the conversion of a 64-bit floating point number into a 16-bit value. The rocket exploded, less than forty seconds after its launch. The reason for this catastrophe was identified to be a software failure in a system responsible for calculating the rocket's horizontal velocity. Since these exemplary events, software and hardware analysis techniques mostly model checkers but also theorem provers have been more widely used [43].

Formal verification based on formal methods is a technique to prove this correctness in an exhaustive manner, that is, all the possible executions of the system are checked. Nevertheless, the acceptance and utilization of formal verification methods is not always easy to integrate in the design of systems. There are important issues that reduce its applicability, as not optimal integration in modelling tools, skills that are required to use it, enough time or memory resources for some verifications cannot be provided. This calls for better integrated development environments, where design and verification tasks are strictly linked together with faster and more usable methods. We also have to consider that using analysis techniques yield a more efficient specification and thus verification of the undesired behaviours. Therefore, it is possible to reduce the time of the whole development process. For a detailed description of analysis techniques, see [26].

5.1.1 Definition

In order to have a characterisation of the term *analysis techniques*, we refer to the definition of *Software Verification and Validation* from the IEEE [1]:

"Software verification and validation (V&V) is a technical discipline of systems engineering. The purpose of software V&V is to help the development organization build quality into the software during the software life cycle. V&V processes provide an objective assessment of software products and processes throughout the software life cycle. This assessment demonstrates whether the software requirements and system requirements (i.e., those allocated to software) are correct, complete, accurate, consistent, and testable. The software V&V processes determine whether the development products of a given activity conform to the requirements of that activity and whether the software satisfies its intended use and user needs. The determination includes assessment, analysis, evaluation, review, inspection, and testing of software products and processes. Software V&V is performed in parallel with software development, not at the conclusion of the development effort."

That definition is applicable to software systems only. We state an analogous one for hardware verification [85]. The desired functionality and the overall architecture of hardware design is typically defined with a high-level specification, as for instance block diagrams, tables, requirements, and informal text. In order to obtain a final design, a combination of top-down and bottom-up design techniques are applied. The specification is checked in validation and verification activities to verify that it does indeed meet physical and implemented design. Based on these definitions we consider domains and the purposes of each analysis technique.

5.1.2 Testing and Simulation

Nowadays, testing, simulation, and code review methods are available for software written in almost all programming languages. In practice, these techniques detect a high number of coding bugs and thus are considered very effective. Software Productivity Research[1] built up an extensive database by collecting software defects and the efficiency of their corresponding defect removal techniques [82]. Considering these results, only a 35% of the total amount of defects in the final product were identified to be bugs inserted while coding. In the results, about 20% of the faults were traced back to requirements, 30% to design and the rest to other types of errors. Hence, in this research, the majority of all errors made during the overall software development process arise during the requirements and the design phase whereas less during coding.

Typically, it is not possible to cover all possible system behaviours using testing and simulation techniques. However, there are also some support tools for exhaustive testing sessions, through test case generation, which use in the verification process systematic verification techniques [75, 76]. All possible system behaviours can be characterised as evolution of internal data or variable values during the execution of a program stimulated with all possible input values. Hence, data and variable values together define a well-defined system state. All those reachable states together define the system's state space. Considering such state definition, even small programs have a large number of states, directly dependent on the possible values that variables or data can have. For instance, a program which has a variable defined in the natural set $\mathbb{N}$, which is usually represented in a program with an integer variable, it can take as many values as allowed by the memory available, each value representing a different state. Without considering the obvious mem-

[1] `http://www.spr.com`

ory limitations, the program can be considered to have an infinite number of states. Since testing and simulation are applied by explicit definition of values to variables, it is impossible or very tedious to examine all possible program behaviours. Therefore, critical behaviours might not be considered during testing and possible defects are not detected.

5.1.3 Inspections, Reviews, and Walkthroughs

We refer to the following definition: "software inspections, reviews, and walkthroughs are human-based methods for analysing documents based on informal or semiformal techniques for the purpose of early and effective defect detection during development in order to improve product quality and reduce development rework" [36].

In order to have a comprehensive overview, we consider the survey from the International Software Engineering Research Network[2] and the Fraunhofer Institute for Experimental Software Engineering (IESE) initiated in 2002, and its evaluations for the state of software review practice [113]. The results obtained give evidence that there are some important benefits from the use of software reviews: improved communication among developers, evaluation of project status, and enforcing standards. Considering the survey results, software reviews are not systematically performed and planed in the industry, in fact about 40% of participants accomplish reviews on requirements or design documents, and about 30% on source code. At the same time, the preparation or defect detection step is often not systematically executed in the industry. Even though empirical studies have demonstrated that a preparation phase for reviews is crucial for effective reviews [132], about 60% of respondents stated that they did not regularly manage it. Considering companies conducting a preparation phase, half of them use a checklist as support; about 10% use more advanced reading techniques, and finally 40% use reading technics, which do not offer systematic support for defect detection. Finally, in a systematic evaluation and improvement program, companies seldom embed reviews. In fact, 40% of the companies do not collect data at all, and 18% collect data but do not analyse them. Considering the 42% who collect and analyse data, only 23% try to optimise the review process. Even though it has been proved that optimisation is crucial to accomplishing effective reviews. Today, inspections, reviews, or walkthroughs are integral parts of most industrial software development models, procedures, and standards [113].

[2] `http://isern.iese.de`

5.1.4 Runtime Verification

Runtime verification [46] is a verification technique that combines formal verification (Section 5.1.5) and program execution. The examination process for finding the defects is made by passively observing the input/output behaviour during the normal program execution. Typically, requirements are specified as temporal constraints, or automata and state charts, and can represent liveness, fairness and safety properties. Different to model checking techniques (Section 5.1.5.1), where a model of the target system is built and verified; runtime verification is instead applied while the target system is in execution. Formal verification methods (Section 5.1.5), as model checking for instance have to deal with infinite traces, whereas runtime verification only considers finite traces. A runtime monitor for detecting requirement violations that monitors on a potentially never-ending system is an exception to this rule.

5.1.5 Formal Verification

Formal methods are the theoretical basis for formal verification and are used for specifying and verifying systems through mathematically based languages, techniques, and support tools [43]. System specification means expressing the system requirements in a mathematical or formal language. System verification is applied after the specification and formally proves that the system meets its requirements. Several tools are available for specifying and verifying system, providing the developers suitable notations, formalisms, and algorithms. There are two fundamental verification techniques: model checking and theorem proving, which we discuss with more detail in the next sections. Compared to (informal) testing, formal verification has the advantage that the verification is made in a complete, semi-automatic or fully automatic exhaustive way, where all the possible executions of the system are considered. For these techniques: first, a mathematical model of the system, and second, formally specified requirements, which the model should satisfy, have to be provided. Formal verification has also some drawbacks: it requires high skills to be used in specific problems, it is time and memory consuming, and it is only as reliable as the used formal models.

Formal verification is applied to complement standard methods such as testing and simulation, in order to increase the reliability of the system under development. Formal verification ensures that the system model meets its functional requirements, before the final hardware is deployed, when usually error corrections are more expensive and difficult. Despite its complex-

ity, formal verification tools have been introduced in industrial development projects: from security and safety related projects, over hardware circuit verification to software driver verification. In the hardware sector, formal verification techniques have been applied successfully and thus have also encouraged the software sector to consider whether similar benefits could be accomplished in program verification [100]. Some progress has been made, despite many challenges remain [84], as properties of hardware and software system are extremely different, namely the strict structure of hardware, the inherently finite state of hardware, and the limited size of hardware [121]. The economic aspect also plays an important role when applying these techniques in a development process. In some domains, there is a strict time between the design and the launch of products in the market, which make difficult to find in time the experts and to train them to use conveniently formal verification techniques. Instead, in other domains, especially for safety-critical systems, these techniques are a consolidated role in the development process.

5.1.5.1 Model Checking

Model checking is a technique that builds a formal model of a system and verifies whether it satisfies a desired property. Generally, the check is performed as an exhaustive state space search on the model. The main challenge is to design and improve algorithms and data structures that are capable to search in large state spaces. Model checkers, i.e. tools, which perform model checking, take two inputs: a finite state model of the system and a formally specified property. A model checker checks whether the system satisfies a property, and provides a "yes" or "no" answer. If the system does not satisfy a property, the answer is "no" and a counterexample is provided, i.e., a trace (system run) violating the property. Counterexamples are useful in order to find bugs in the system design as they guide developers to the unintended behaviour. Figure 5.1 describes the model checking process.

In today's practice, two general approaches are used: the first, *temporal model checking* is a technique developed in the 1980s independently by Clarke and Emerson [40] and by Queille and Sifakis [110]. There, the specifications φ are expressed in a temporal logic [107] and the model $\mathcal{M}$ as a finite transition system. Verification is performed through an efficient search procedure, which checks if the finite state transition system is a model for the specification. In the second approach, the system and the specification are modelled as automata, and the system is compared to the specification to determine whether its behaviour conforms to the specification. Different notions of conformance have been proposed, as language inclusion [66, 88],

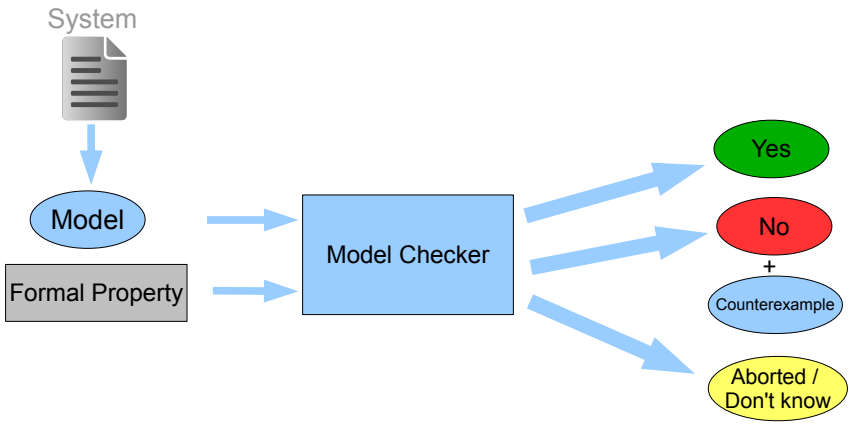

Figure 5.1: Overview of the model checking procedure.

refinement orderings [44, 114], and observational equivalence [44, 58, 115].

Model checking is completely automatic, in contrast to theorem proving, and in some cases produces an answer in a relative short time, especially if the property is not satisfied and a counterexample is provided, which usually points to subtle design errors and model checking can be used to improve the debug phase. The ”state space explosion” problem is the main disadvantage of model checking. In 1987, McMillan proposed Bryant's ordered binary decision diagrams (BDDs) [24] to represent the state transition systems in a more efficient way. BDDs can be used to check CTL formulas. In order to ease the state space explosion problem, other approaches have been suggested, as for instance the exploitation of partial order information [102], localisation reduction [88, 87], and semantic minimisation [83], which eliminates unnecessary information from a system model. Today, model checkers are expected to work with systems having between 100 and 200 state variables. Systems with 10^{120} reachable states have been checked by model checkers [25], instead systems with an essentially unlimited number of states has to use abstraction techniques [41].

5.1.5.2 Bounded Model Checking

In the semiconductor industry bounded model checking (BMC) is one of the most commonly applied formal verification techniques. The success of this technique comes from the brilliant performance of propositional SAT solvers.

Biere and others have introduced BMC in 1999, as a complementary method to BDD-based and unbounded model checkers [19]. BMC visits all states within a bounded number of steps, say k. Using BMC the model is built in the following way: A propositional formula is built from the system until k steps in conjunction with a property. Next, the obtained formula is passed to an efficient SAT solver. If and only if there is a trace of length k that contradicts the property the formula is satisfiable. If the propositional formula is unsatisfiable, we cannot be certain about the correctness of the verification, because there may be counterexamples longer than k steps. BMC provides a full counterexample trace in case of the property is not satisfied, and it is one of the best techniques to identify shallow bugs.

5.1.5.3 Theorem Proving

Theorem proving uses formulas in some mathematical logic or theory, to express the system and its properties [43]. A mathematical logic specifies the *formal system* and defines a set of axioms and a set of inference rules. In practice, theorem proving finds a proof of a property from the axioms of the system. The proof is composed of steps, which invoke the axioms and rules, and derive definitions and intermediate lemmas if possible. Theorem proving approaches perform verification using calculi that are based on two principal works. The first is by C.A.R. Hoare, which presents a calculus to reason about program correctness in terms of pre and postconditions [74]. The second is by E.G. Dijkstra, who extended Hoare's ideas in the notion of "predicate transformers" which begins with a postcondition and uses the program code to determine the precondition, instead of starting with a precondition.

Nowadays, theorem proving is used more and more in the verification of safety-critical properties of hardware and software systems. We can coarsely categorise theorem provers into the classes (i) highly automated, (ii) general-purpose programs, and (iii) interactive systems with special-purpose capabilities. Approaches based on automatic verification have been useful as search procedures and were successfully applied in solving combinatorial problems. Interactive systems are more appropriate for automating formal methods and the formal development of mathematics [43]. In order to prove properties on infinite domains, theorem proving relies on techniques such as structural induction. The interactive theorem proving process is slow and often error-prone, because, as the name suggests, it requires the selection of possible theorems by a human. However, the user often gains useful insight into the system or the property being verified, during the process of finding the proof. Theorem proving can deal with systems having an infinite state space,

whereas model checking usually cannot.

5.2 Model Checking of Discrete Components

To develop correct hardware and software systems, two fundamental phases are validation and verification. Formal verification methods as model checking are not easy and often require skills to handle the temporal logic formulas, and to understand and debug the counterexamples, but they provide a formal and comprehensive proof of correctness of the system. Considering the available analysis techniques for model-based development, we decided to implement the support for model checking in the modelling tool AutoFocus 3. Some disadvantages of this technique can be reduced with a model-based and user-friendly integration, to overcome the intrinsic complexity of model checking.

We demonstrate applicability of verification mainly in four areas. Firstly, we integrate specification and verification tightly into the model-based development process. This means that verification properties are linked to model elements and can be verified locally easily during the development. In addition, the support for different specification languages is essential. Therefore, we implemented property templates for simpler but recurrent cases and high-level languages for the more complex properties. Properties disproved by verification must be analysed and either the system model or the properties have to be corrected. In the case of model checking, disproof is given by counterexamples, which we present to the user either as a simulation of the system model or as a message sequence chart [125]. For formal verification to be actually practical, it must perform efficiently and may not impose long delays on the user. Therefore, we evaluate various optimization techniques implemented in the model checkers, TVARC [23] (only prototypical) and the Cadence Symbolic Model Verifier (SMV) [93].

TVARC We study model checking based on three-valued logic system instead of common two-valued logic, which extensively applies model abstraction and notions of model refinement. Three-valued Abstraction Refinement (TVAR) extends the existing work on the three-valued logic model checking [23]. The three-valued logic results *true*, *false* and *don't know*, are used in the model checking algorithm for refining an abstraction in a similar manner to the Counterexample guided abstraction refinement (CEGAR) approach [39]. TVARC is our prototypical implementation of the model checker integrated in AutoFocus 3 that accept as input C programs and μ-calculus formulas to

be verified. In this implementation, an AutoFocus 3 model is transformed to corresponding C code by a code generator, more precisely a generator for C0 code a C language subset.

In C0, some features are forbidden, including pointer arithmetic and the non-nested use of function calls. In any case, the code generated from AutoFocus 3 models does not need these features and other characteristics still available in C0 like dynamic memory allocation, pointers or arrays. We further verified the C0 code representation of the implemented model with model checking techniques. However, the implementation of TVARC is at a prototypical stage. Since TVARC abstracts the model to being verified, it also reduces the state space dimension in the verification phase. We worked to extend the foundation of our three-valued logic model to a multi-valued logic model. A formula in a multi-valued logic evaluates no longer to just true or false but to one of many truth values, therefore we can express to which extent a property is considered satisfied by a model. The main motivation to introduce multi-truth values is a support for software product lines or product families, so our system models are explicitly expressed differences and shared characteristics of the different product versions. Thus, the question of which products of the product line satisfy a certain property corresponds to the truth value of the formula encoding the property with respect to the multi-valued system. The model checking verification is based on an extension of the refinement and abstraction principle for the three-valued logic. A complete explanation of this approach can be found in [29].

Cadence SMV The choice of Cadence SMV as model checker is mainly due to its semantics. In fact, in AutoFocus 3 the interconnected components execute an elaboration step synchronously, in the same manner as the modules in an SMV model. Furthermore, the symbolic model checking provided by SMV works well with hardware-like systems, which fit well with the embedded systems modelled in AutoFocus 3. SMV guarantees one of the best performances for the formal verification of such systems available. In order to perform the verification, the original models are mapped into the SMV language, where each component is mapped to an SMV module. Then the resulting SMV program is executed for verification, with the integrated model checker. The results are displayed directly in the tool environment.

5.2.1 Theory: From AutoFocus Components to SMV Programs

In the literature, the usage of SMV language to model finite-state Symbolic Transition Systems (STSs) is widely documented. We repeat here the fundamentals concepts of the SMV language from [93]. This language is the input language of two widely used model checkers: Cadence SMV and NuSMV[3]. Cadence SMV is a well-known model checker branched from the original SMV and developed by Cadence Berkeley Labs. Instead of storing states and transitions explicitly, in STSs the states and transitions are described with symbolic formulas. Thus, STSs specify the semantics of SMV programs. Formally an STS S is a tuple $\langle V, W, I, T, Z\rangle$ with:

- V is the set of symbolic variables that represent the states,
- W is the set of symbolic variables that represent the events,
- $I(V)$ is the initial formula,
- $T(V, W, V')$ is the transition formula,
- $Z(V)$ is the invariant formula.

In the transition formula, V' is the next state reached from the state variables.

A set of symbolic variables represent the state of a system and are partitioned into *modules*. In an STS, the same module can be instantiated several times, to define them in a hierarchical way. In an SMV model, a component is an instantiation of a module. In a composition of components, also hierarchical, each elaboration step is synchronous and is equivalent to the conjunction of a step in each component. A program in SMV is composed of a set of modules and each module is described by a tuple $\langle PARAM, VAR, IVAR, INIT, TRANS, INVAR\rangle$ where:

- $PARAM$ is a set P of formal parameters,
- VAR defines a set of variables V, with a type $\tau(v)$ for each variable v and is a set of variable declaration,
- $IVAR$ defines a set W of variables, with a type $\tau(w)$ for each variable w and is a set of input variables,

[3] `http://nusmv.fbk.eu`

- $INIT$ defines a formula I using the variables in $V \cup P$ and is a set of initial conditions,
- $TRANS$ defines a formula T using the variables in $V \cup P \cup W \cup V' \cup P'$,
- $INVAR$ defines a formula Z using the variables in $V \cup P$.

Module instantiations may be used in the VAR declaration, in the form $\langle M, \beta \rangle$, where M is a module and each formal parameter is associated with an actual parameter β. The actual parameters must have the same type of the parameters, so for each $p \in P$, $\beta(p)$ must evaluate to the same type.

The structure of an SMV program is hierarchical with module instantiations, where the root node is a so-called *main* module. The STSs, previously presented, define the semantics of an SMV program. An STS corresponds to a Labelled Transition System (LTS). Assuming that there are no circular dependencies in the module hierarchy, the semantics of a module is defined recursively on the hierarchical structure; a detailed explanation is in [93]. AutoFocus 3 models are converted to an SMV program, mapping each component into an SMV module, where the root component is the module main of the SMV program. The semantics of SMV is similar to the synchronous execution model of AutoFocus 3, where other model checkers have an asynchronous execution model, as for instance the tool SPIN[4]. In fact, an SMV program with more modules is executed in parallel, so that the transition relations of each module are conjoined and the initial predicates are instantiated together (cf. [31]). In Section 5.3.2, we specify formally the transformation from hybrid Focus components to HyDI programs; therefore, we omit here the similar formal transformation from Focus components to SMV modules.

An SMV module is specified as a state machine, where the variable values, boolean, integer or integer interval, which determine the state space. At the beginning, the initial values of variables determine the initial state of the model. SMV specifies the assignments to the state variables in the following step, with a transition relation. We translate directly the data types and functions in the data dictionary in AutoFocus 3 to SMV data types and functions. As discussed above we restricted to finite data types and not recursive functions, because SMV cannot handle infinite state systems. In AutoFocus 3, the user can define functions through pattern matching, which we converted in SMV functions. In the AutoFocus 3 models, the components exchange synchronously at most one data value in each elaboration step, so we map the channels to SMV variables, with an extra variable to handle

[4] `http://spinroot.com`

the absence of a value over the channel. AutoFocus 3 supports hierarchical definition for the system components, so we translated each component to an SMV module. We support two kinds of implementations for the components: state machines and functional definitions, both component implementations are translated to a single SMV module. For state machines, we support local variables and use a state variable, an enumerator, to simulate the states. If the transition relations of the state machines are non-deterministic, also the SMV module has the same non-deterministic behaviour in the verification. Each input and output port of the component is mapped to a parameter of the SMV module. Consequently, the original hierarchy and structure of the AutoFocus 3 model is replicated through module instantiations. A functional specification is translated to an SMV module. The SMV main module corresponds to the root component in the component hierarchy defined by an AutoFocus 3 model.

5.2.2 Implementation in AutoFocus 3

For the usability of verification, it is essential to reduce the complexity by specifying verification properties and the handling of the verification process. The user should have a user-friendly environment to adapt the properties and to trace its semantical connection to the system model. We evaluate several model checkers that implement different optimization strategies, in order to practical test their characteristics. We apply the TVARC model checker to the C0 code generated from the model, (to be precise its C intermediate language representation) and SMV directly to the model design, as depicted in Fig. 5.2.

We generated two artefacts from the system model, namely a C0 code implementation, and an SMV representation. The correctness of the transformation to C0 has been shown in [72]. For the execution of the SMV model checker, AutoFocus 3 automatically translates the selected component with all its subcomponents into an SMV instance, where an SMV module represents each state machine. Each module contains its subcomponents as nested modules. During this translation, we have restrictions for recursive constructs. In general, AutoFocus 3 provides recursive functions and data types. However, the SMV generator and the C0 generator do not support these features. This limitation is due to the finiteness of the state space (required by SMV and often applicable in embedded systems design) and the fact that recursive data types induce a potentially infinite state space (of course, only a finite part may be reached). In such cases, we return an error message to the user.

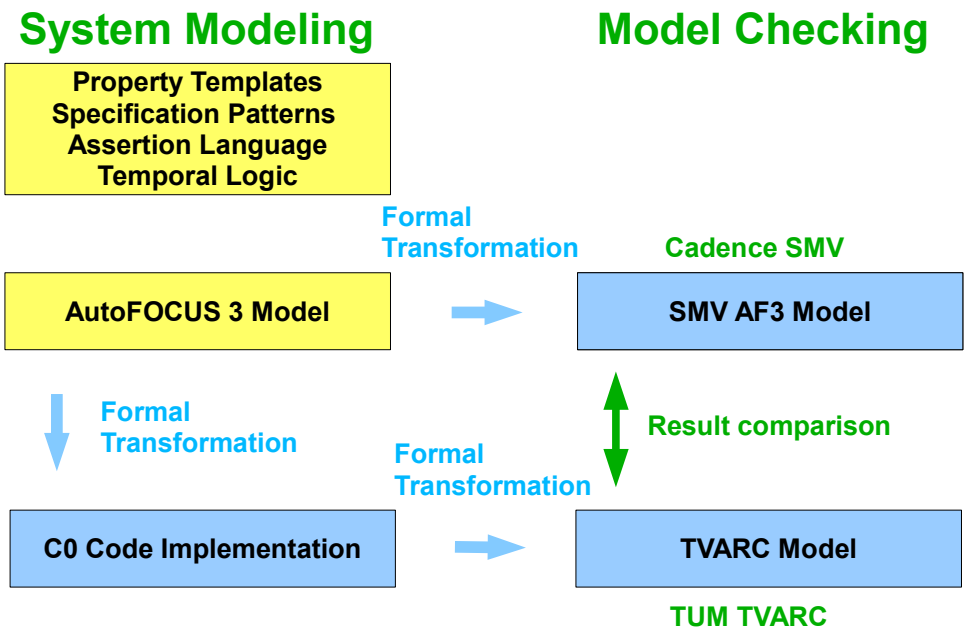

Figure 5.2: Verification artefacts and activities in AutoFocus 3.

Another limitation is given by large state variables like integer variables. Excessive use of these quickly leads to the so-called state space explosion. Even in small projects, we noticed that the verification process required too much time and memory. Therefore, in the model checker view the user can restrict the value range of integer variables for the whole model, and define a specific range for certain variables (Fig. 5.3). It is important that the user specifies the smallest possible integer interval to avoid the state explosion problem. With static analysis, we could automatically determine these intervals. The application of abstraction techniques may help to reduce these problems. Further, SMV has already been tested in a case study from the automotive domain [56], whereas the testing phase of TVARC's integration is not yet published.

5.2.2.1 Properties Verification

Our approach allows the designer to attach verification properties to components in addition to their decomposition or their behaviour description. The scope of a verification property is fixed by the component it is attached to. Thus, the property refers to the i/o ports of the component. If available, the property can additionally refer to all i/o ports of the immediate sub-

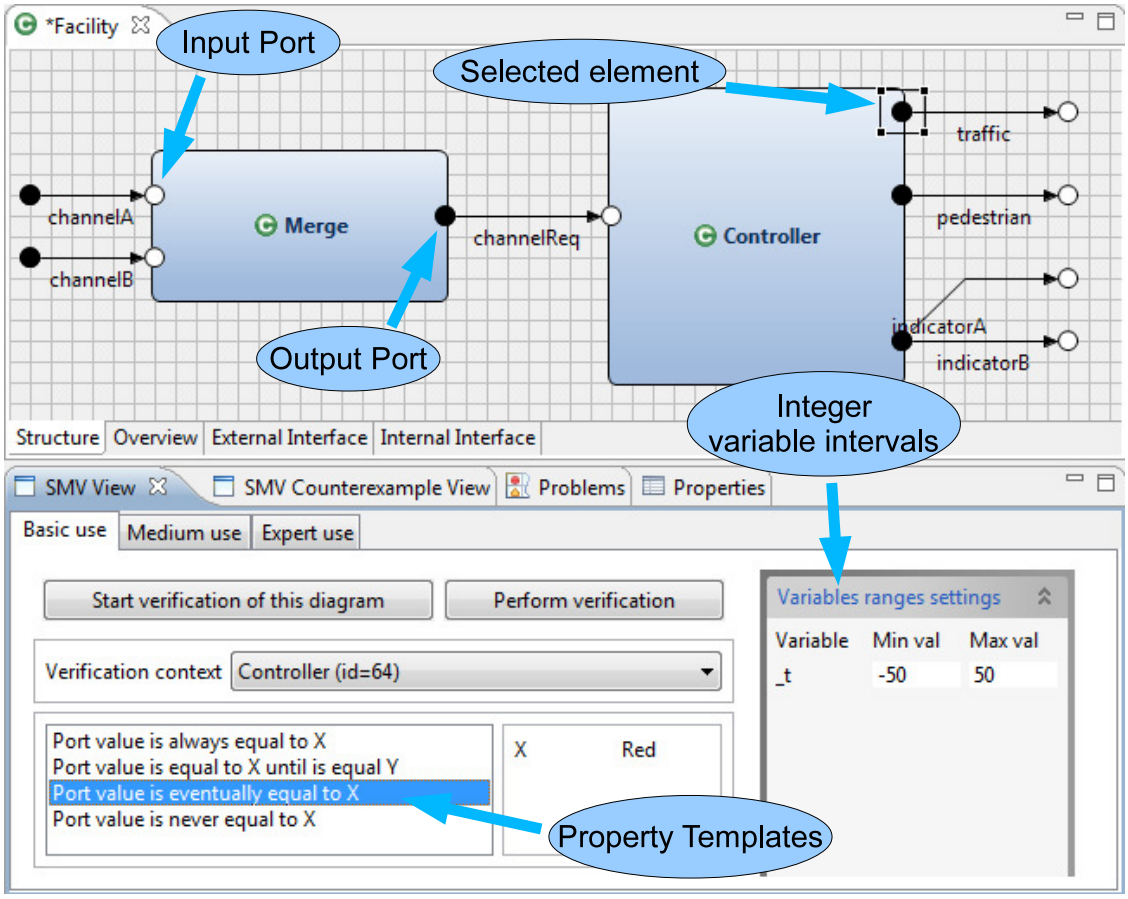

Figure 5.3: Property template options for model checking in AutoFocus 3.

components. If an automaton defines the component, the automaton's state variables are also in scope. After augmenting the model with verification properties, we are able to choose the component to be verified. If we verify a component, this component together with all its subcomponents are considered. The chosen component is verified against all possible inputs allowed by the types of its input ports, thus the components environment (consisting of the non-descendant components) is completely ignored. Contrary, if we would like to verify a component together with its directed connected components, we would have to verify the parent component containing all these components. Thus, the chosen component fixes the verification *context*.

For illustration, Fig. 5.3 shows a small component network in AutoFocus 3. In this view, the user selects a model element in the upper part and a predefined property template from the list below. The templates contain free

variables that the user has to define. Therefore, the user can refer to model elements like i/o-ports or state variables. The context of the verification is selectable from a drop-down list containing all parent components. Finally, the verification can be initiated. A model checker performs the verification. If the model checker is able to prove the property, the user is presented a confirmation. If the property is disproved, a counterexample is generated for the user's review. Counterexamples can be simulated stepwise or illustrated as MSCs, see Fig. 5.4 and 5.5.

Benefits As previously explained, the user can specify a verification condition in the scope of a certain component. This allows the tool to facilitate the formulation of verification properties. For example, the property's formula is type checked with respect to the types of the ports and state variables. Other supportive functionality is discussed in more detail in Section 5.2.2.2. This shows that the tight coupling of properties to the system model improves the process of verification and validation. Our approach also allows to locally verify properties of a component. By verifying components and their subcomponents independently from the surrounding environment, the developer can apply verification continuously throughout the development process. Thus, implementation errors can be found earlier and bug fixing costs are reduced.

Contrary to operational behaviour descriptions, the developer can formulate verification properties in a declarative way. In many cases, a declarative description is considerably shorter and more comprehensible than the operational description. Furthermore, each property is specific to a certain aspect of the component's functionality allowing to abstract from the complexity of the component's behaviour. With the presented tool integration, the developer can easily formulate such properties during the component's development. In the same manner as with unit tests, which should be defined together with the implementation, our model checking integration allows the same approach but using formal verification. This way, as the system development proceeds, several verification properties are collected and contribute to the confidence in the system's implementation.

5.2.2.2 Specification of Properties

The formal properties to be verified on the system model are specified in temporal logic [107]. To specify the properties, SMV supports the following temporal logics: Computational Tree Logic (CTL) [42] and Linear Time Logic (LTL) [107], meanwhile TVARC supports μ-calculus [55]. Temporal

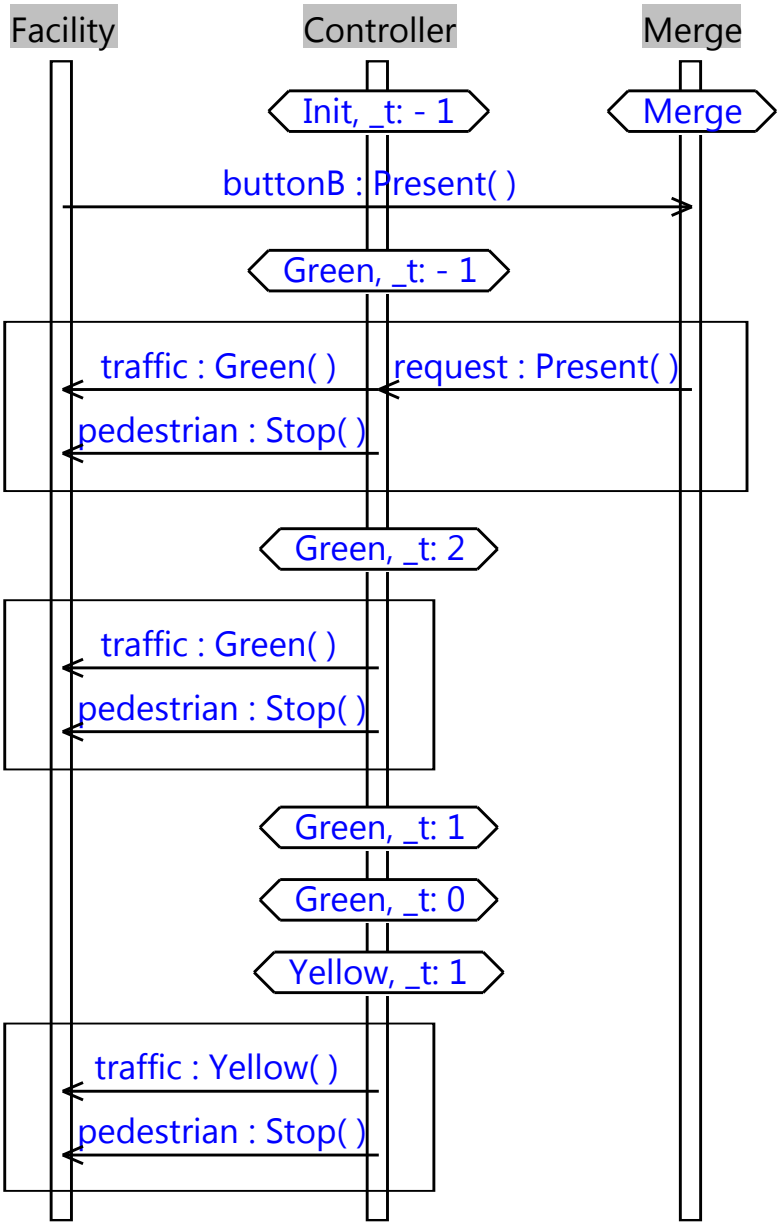

Figure 5.4: Counterexample visualized as MSC.

logic formulas can be subdivided in *safety*, *liveness* and *fairness* properties. If one wants that something, like an event or a state of the system, must not happen then is a safety property. For instance, in an ATM a safety property is the check that it does not release the money if the login of the user was not successful. Instead, if something must happen then the property is called a liveness property. In an ATM, a liveness property is that if the user inserts the card then must be returned to the user. A fairness property for an exchange protocol between a banking server and an ATM argues, when the communication has been terminated, either the server has received an ATM message and the ATM has received a server message, or none of them has lost their messages.

The integration of SMV/NuSMV in AutoFocus 3 models supports both CTL and LTL, which can express all the mentioned property kinds. Their formulas are inductively defined with logical and temporal operators. To understand exemplary safety and liveness properties expressed in LTL, we explain two temporal operators: *G*, for globally, where the formula affected by the operator has always to hold; *F*, for future, where the formula affected by the operator has to hold eventually. An example of a liveness property is *if "input x received" holds then "output y provided" eventually*:

$$G(input\ x \Rightarrow F\ output\ y)$$

A safety property is *the error state z is never reached*:

$$G(\neg actual\ state\ ==\ z)$$

In a formula specified with AutoFocus 3 is possible to refer to i/o streams, internal variables and actual states of the component state machines.

Practical application of such temporal logic requires advanced skills in formal verification, since it is difficult to express properties in these formalisms. The acquisition of this level of knowledge may be an obstacle to the adoption of automated verification tools. In order to improve the usability and expressiveness in the specification of the properties, we propose three higher level approaches. The most intuitive one is based on predefined *Property Templates*, the second uses a *pattern-based approach*, and the third is based on the *Structured Assertion Language for Temporal Logic* (SALT) [14].

The first approach, as highlighted in Fig. 5.3, provides templates that can be directly selected, configured and executed from the graphical interface. We represent common properties to be verified, which are integrated in the model checker view and can be saved along with the model itself. The user selects an element in the model view, and then the appropriate templates

are shown. Examples of property templates for a port and a component are, respectively:

- Port value is eventually equal to X
- After input port A has value X, output port B has eventually value Y

Templates contain variables, e. g., "X" in the precedent template. Each variable is bound to the content of an associated text field. Using these text fields, the user defines logical parts of the formula. Variables correspond to states or port values or, if they are indicated as "events", to general formulas. A type and name check is made for controlling the correctness of the values inserted by the user.

The second approach is a pattern-based approach for the presentation, codification and reuse of property specifications for finite-state verification. We refer to the specification pattern definition given by [53]: "*A property specification pattern is a generalized description of a commonly occurring requirement on the permissible state/event sequences in a finite-state model of a system. A property specification pattern describes the essential structure of some aspect of a system's behaviour and provides expressions of this behaviour in a range of common formalisms*". The patterns are based on a survey of available specifications, collecting over 500 examples of property specifications. Most of these specifications were instances of the patterns. A specification pattern is composed by the following elements: name, description of its behaviour, mappings into temporal logic formalisms, examples of uses and relationships with other patterns. For instance, the *absence* pattern specifies a state/event that must not occur; and the *response* pattern specifies an initial state/event, which, when it occurs, must always be followed by another condition/state/event.

It is necessary to associate a scope with a pattern, which specifies the extend of the program execution, in which the pattern must hold. The authors of the specification patterns found five scopes [53]: *global*, when the pattern must hold for the whole program execution; *before* when the execution must hold ahead of a state/event; *after* when the execution must hold after a state/event; *between* when the execution must hold between two given states/events; and *after-until* when the execution must hold between two given states/events and also after the first state/event occurs and the second state/event can eventually not occur.

We added most of the specification patterns in the model checker view of AutoFocus 3, where the user has to select from a list a scope and a pattern [53]. After that step, the correspondent formula parts are presented and

should be filled with the desired elements by the user, as already explained for property templates. The user can fill the expressions with logical operators and elements of the model. In the selected pattern, the role of each text field is indicated with a short description.

The third approach is based on the specification language SALT, which provides a higher level of abstraction compared to other temporal logics formalisms. This is based on ideas of existing approaches, such as specification patterns but also provides nested scopes, exceptions, support for regular expressions, real-time, and employs some constructs similar to a programming language. As a result, its specifications are more intuitive to utilize and to understand than temporal logic. We have added support for these three concepts in the front end view of AutoFocus 3 of the model checker.

The formalization of system properties and assertions have been conceived for Focus components and for the hybrid systems introduced in [21]. In this work, we define interface assertions over the components channels of the syntactic interface. They are preferred to the verification through temporal logic. The main motivation is better expressiveness of the assertion based approach, to specify formulas over the behaviour of i/o channels, through predicate logic and basic operators over streams. In our approach, we selected an existing model checking solution for Focus components, which guarantees a seamless integration, counterexample handling, suitable semantics, and reasonable performances. We implemented the integration of the model checker Cadence SMV in AutoFocus 3 and improved its capabilities to express properties to support additional high-level specification languages.

5.2.2.3 Handling of Counterexamples

In case a verification fails, a counterexample generated by SMV is reported in a tool view. The counterexample represents a path in the SMV model. This path leads from an initial state to a state where the property is violated or it contains a loop, where the desired behaviour is never reached. The path is visualized as a sequence of states, together with the eventual loop information. In this sequence, each state describes a current state of the state machines and a valuation of all variables and i/o ports that compose the model.

In order to improve the representation and understanding of such data, we visualize the counterexample as a message sequence chart. In the generated diagram, the actors are the different components of the model and the messages between them represent the value assignment in the model from an

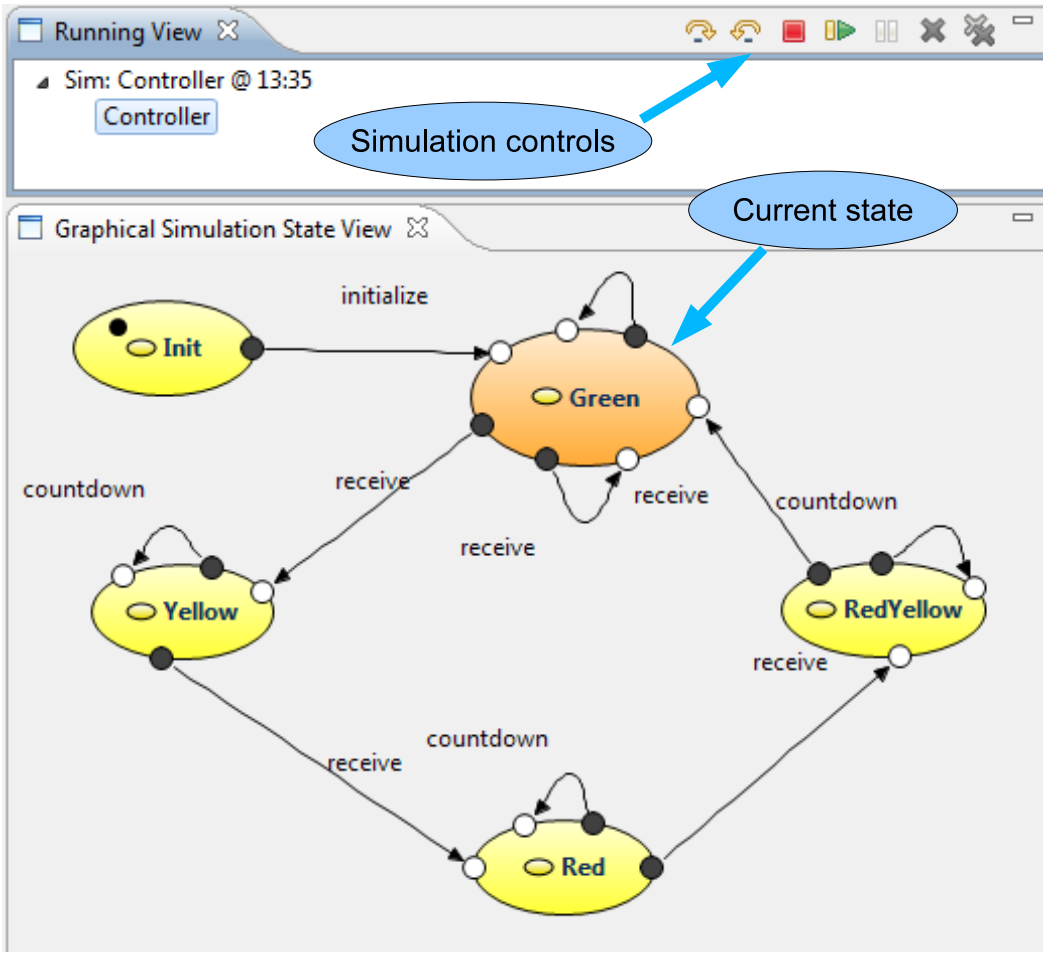

Figure 5.5: User interface for simulating counterexamples.

output port of the sender component to the input port of the receiver component. It is also possible to simulate the counterexample in AutoFocus 3. The view shown in Fig. 5.5 contains controls to execute forward and backward steps as well as continuous playback of the simulation. In other views, the user can examine the current values of i/o ports and a state machine's transition system with the highlighted current state. If the model contains more than one component, it is possible to open more component views. A step in the simulation corresponds to the passage from a state to the following one in the SMV counterexample. In case a property is disproved during the verification, these views help to debug the model. The described debugging phase is solely based on the structures previously modelled by the user. Thus, the intermediate model generated for the model checker is not shown. We consider the integration of the verification and counterexample analysis in the tool a big advantage as the user is not bothered with the underlying structures and the model checker anymore.

5.2.3 Related Work

To the best of our knowledge, an integrated and user-friendly verification tool environment as presented here has not been widely supported in the modelling tools. Standard industry tools integrate similar formal verification capabilities. *Simulink Design Verifier* and *SCADE Design Verifier* are extensions for Simulink[5] and the SCADE tool[6], which use formal analysis techniques provided by Prover Plug-In. *Rhapsody in C++* [118] has a verification environment for UML models, the specifications to be verified can be formulated using temporal patterns or a graphical specification formalism called Life Sequence Charts [47]. The verification is applied to UML models composed by UML state charts and UML class diagrams. They are transformed in a format for VIS model checker [20]: a finite state machine for the model and a temporal logic CTL formula for the property to verify. In Hugo/RT [12] UML state chart diagrams are associated with a UML class diagram to specify the model to verify and communication diagrams describe how the objects of a model may interact. The tool generates input for two model checkers: SPIN and UPPAAL [16]. The model checkers can verify whether the interactions expressed by a UML communication diagram are realized by the state machines. Considering the specification of the model checking properties, none of these approaches integrates user-friendly templates with correctness checks, specification patterns, assertion language and

[5] `http://www.mathworks.com`

[6] `http://www.esterel-technologies.com/products/scade-suite`

temporal logic. Additionally, none of these tools has an integrated counterexample handling and simulation.

5.3 Formal Verification of Hybrid Components

We implemented a first holistic and user-friendly verification environment with a focus on the model checker integration. In order to enhance the usability of model checking in model-based development, we integrate different property specification concepts and back-translation of counterexamples to the model level in the modelling tool (e.g., counterexamples can be examined step by step in the AutoFocus 3 Simulator). In the official release of AutoFocus 3, NuSMV model checker a reimplementation and extension of SMV is supported. The features of NuSMV are almost the same but NuSMV provides an open architecture for model checking, which offer more extensions possibilities respect to Cadence SMV. Both implementations with NuSMV/Cadence SMV are made for discrete time systems, so they do not support the continuous dynamic of hybrid components. In this section, we present our ideas for an extension of formal verification approaches also to hybrid systems.

5.3.1 Hybrid System with Discrete Interaction (HyDI)

The authors of NuSMV introduced support for formal verification of hybrid systems in a new version of their model checker that is publicly available: HyCOMP[7]. This is implemented by translating system models into the HyDI (Hybrid system with Discrete Interaction) [35] language. It is possible to verify invariants over the real variables, using MathSAT[8] an SMT (Satisfiability Modulo Theories) solver integrated in HyCOMP. HyDI is a language for modelling systems with discrete and continuous dynamics on real-valued variables. The intention of the authors was to overcome the limitation of model checking in the verification of infinite-state systems. In fact, HyCOMP supports complete symbolic model checking with verification of LTL properties.

Its foundation is the standard symbolic language SMV that is extended with continuous variables, flow conditions, and synchronization between system modules. The SMV models are hierarchical and the variables evolve in

[7] `http://es-static.fbk.eu/tools/hycomp`
[8] `http://mathsat.fbk.eu`

each module synchronously. On top-level the modules in HyDI are asynchronous and use events for the synchronization. A discrete-time infinite transition system is the target model of a HyDI instance. The instantiation of HyDI modules are called HyDI processes.

The language HyDI supports four different semantics, combining global and local time and asynchronicity that can be interleaving or step. There are two verification modes: the first translates the model with MathSAT applies SMT-based techniques; the second mode work on the structure of the network. There are sets of benchmarks available[9] and it is possible to translate models of MATLAB Simulink/Stateflow and Altarica to HyDI programs. The SMT based-verification is applied to hybrid automata networks [35], that is a parallel composition of two or more hybrid automata [68]. The time progression is local to each automaton, so there is a local-time semantics where the shared events are labelled with time-stamps and there are specific events to synchronize the components.

5.3.1.1 HyDI Programs

In Section 5.2.1, we explained the fundamental structure of the SMV modules. In HyDI a set of modules, which extends the SMV modules, are given. These modules introduce three features: events for the synchronization between processes, continuous variables that can evolve in continuous time and flow conditions that define the continuous evolutions of the variables also referring to the derivative of the continuous variables. In a HyDI program it is possible to instantiate other modules in the *main* module, which can be asynchronous or synchronous using shared events. There is the same mechanism as in SMV program to share variables through the module parameters, but the network is not hierarchical.

5.3.2 Transformation of Focus Hybrid Components to HyDI Programs

Considering the experience gained with the discrete components and the importance of checking CPSs, we worked towards a formal verification support also for hybrid components. We integrate the theoretical foundation for the transformation of hybrid systems into HyDI programs, in order to apply its verification capabilities. We would like to use the verification in future to

[9] `http://es.fbk.eu/people/mover/hydi`

test invariants for the variable values, to gain information that could be used in the sampling algorithms.

5.3.2.1 System Representation

HyDI programs describe the system with a symbolic representation using state variables, in an analogous way as for SMV programs (cf. Section 5.2.1), but they extend the SMV module introducing continuous data types and elaborations. Symbolic Hybrid Systems (SHSs) define the semantics of HyDI modules, recursively on the instantiation of the modules as for SMV modules. The tuple $\langle D, X, Ev, IN, T, Z, F\rangle$ represents an SHS S:

- D represents the states and is the set of symbolic discrete variables,
- X represents the states and is the set of symbolic continuous variables,
- Ev represents the events and is the set of symbolic variables,
- $IN(D, X)$ represents the initial formula,
- $T(D, X, Ev, D', X')$ represents the transition formula,
- $Z(D, X)$ represents the invariant formula,
- $F(D, \dot{X})$ represents the flow formula.

A HyDI module is described by the following tuple, similarly to an SMV module (see Section 5.2.1):
$\langle PARAM, VAR, IVAR, INIT, TRANS, INVAR, FLOW\rangle$ with the following differences respect to an SMV module:

- VAR is now composed from discrete variables in the set D and continuous variables in the set X,
- $FLOW$ defines a formula F using the variables in $D \cup Ev \cup \dot{X}$ that is a set of flow conditions. Where $\dot{X}$ defines the differential equations associated to the continuous variables.

5.3.2.2 HyDI Semantics for Focus Hybrid Components

The semantics for HyDI programs is defined in [35] for asynchronous and parallel composed processes, whereas FOCUS hybrid components are synchronous and hierarchical defined. We do not consider the synchronization events of the original HyDI programs, because the introduced models do not need such events. The hybrid automata used and implemented in HyDI programs are linear as the automata used in hybrid components. This means that the predicates in $FLOW$, which contain the operator *der*, must be a linear combination of the derivative operators. In this section, we define the semantics of HyDI programs translated from FOCUS hybrid components, based on the global time semantics presented for HyCOMP.

It is straightforward to define a semantic correspondence between an SHS $S = \langle D, X, Ev, IN, T, Z, F\rangle$ and a FOCUS hybrid component $H = (\Sigma, Var, Init, I, O, Dom, E, f, G, R)$, which is specified in the Definition 15, with $\Sigma = (Q \times V)$, $I \subseteq V$, $O \subseteq V$, $Int \subseteq V$, $Int_d \subseteq V$, $O_d \subseteq O$, $O_c \subseteq O$ and $W = Int \cup O_c$:

- Q is represented by the set of assignments D;
- V is represented by the assignments X;
- W is represented by the set of assignments $Y \subseteq X$, where Y are the internal and output continuous variables;
- $G(q_1, q_2) = \{a \mid q_1, x_1, a, q_2, x_2 \vDash T\}$ the guard function G corresponds to the events of the transition formula;
- $Init = \{(q, x) |\ q \in Q, (q, x) \vDash I\ for\ some\ assignment\ x\ to\ X\}$;
- $Dom(q) = Z|_{q(D)}$ that is the domain function in the state q is given by the invariant formula Z restricted to the state q with all the value of V that satisfies the formula;
- $E = \{(q_1, q_2) \mid q_1, q_2 \in Q,\ q_1, x_1, a, q_2', x_2' \vDash T \wedge a \in G(q_1, q_2')\ for\ some\ assignment\ x_1\ to\ X\ and\ x_2'\ to\ X'\}$ the set of edges E is derived from the transition formula;
- $f(q, v) = F|_{q(D),v(X)}$ the vector field function for the state q and actual value of the variables v is given by the formula F restricted to the state q and assignments v, with all values from D and X that satisfy the formula;

- $R((q_1, q_2)), v) = T|_{q_1(D),v(X),q_2(D')}$. The reset function is associated to a transition between states q_1 and q_2, where the actual state v is defined by the formula T restricted to the states q_1, q_2, and assignments v to variables with all values from D, X, and X' that satisfy the formula.

In our transformation, we do not have events in Ev, because we embed the conditions without using the events, this set will be empty in the transformation. In fact, we have a synchronous execution meanwhile the HyCOMP uses an asynchronous execution with events for synchronize interconnected processes.

In the following statements, the module declarations $INIT$, $TRANS$, $INVAR$ and $FLOW$ are restricted, according to the semantics for the HyDI modules presented in [35]. The formulas IN, T, Z, and F are built as boolean combinations of assignments to the discrete variables, whereas linear constraints are defined over the continuous variables. The constrains are of the form $\sum_j c_j x_j \bowtie c$, with $\bowtie \in \{\leq, \geq, <, >, =\}$, c_j, c are constants and x_j are variables, variables in the next elaboration step or derivative. Moreover, Z and F formulas limit the assignments s to variables in D as conjunctions of linear constraints. However, it will be necessary to overcome some of such restrictions in the definition of the constraints, in order to have a complete transformation of FOCUS hybrid components in HyDI programs and their verification.

A HyDI program is defined by a set of modules similar to a hybrid component with its subcomponents. The top component in the hierarchy corresponds to the *main* module that is the root module of a hierarchy build by the recursive module instantiations in each module. A given HyDI program P_H, generated from a FOCUS hybrid component H, consists of a hierarchical module structure, with the root module
$main = \langle PARAM, VAR, IVAR, INIT, TRANS, INVAR, FLOW \rangle$ such that:

- $PARAM$ represents the set of parameters (that is an empty set);
- VAR represents the set of declarations that defines the variables in the set D and X, where the variables in X are continuous and the variables $d \in D$ have type $\tau(d)$, it is also added an additional argument δ for each process instantiation;
- $IVAR$ represents the set of declarations that define the set of input variables Ev. We do not need to define variables for the synchronization of processes in our translation, but we add a declaration for assign the continuous type to δ;

- $INIT$ represents the set of initial conditions declarations that define a formula IN over $D \cup X \cup P$;
- $TRANS$ is constructed to include also the formula F that constitutes the $FLOW$ declarations. It is defined by the formula $T \wedge F'$, where T is a formula over the variables $D \cup X \cup P \cup EV \cup D' \cup X'$ and F' is a formula constructed from F, in which the predicates of the form $\sum_j \dot{x_j} \bowtie c$ are substituted by $\sum_j x'_j - x_j \bowtie c \times \delta$;
- $INVAR$ represents the set of invariant condition declarations that defines a formula Z over $D \cup X \cup P$.
- $FLOW$ defines a formula F using variables in $D \cup Ev \cup \dot{X}$ that is a set of flow conditions.

After the definition of the root module, we define the other modules in the hierarchy. Each module M_i is a tuple:
$\langle PARAM_i, VAR_i, IVAR_i, INIT_i, TRANS_i, INVAR_i, FLOW_i \rangle$, with:

- $PARAM_i$ represents the set of parameters and an extra parameter δ;
- VAR_i, $IVAR_i$, $INIT_i$, $TRANS_i$, $INVAR_i$, $FLOW_i$ have the same definition respectively as VAR, $IVAR$, $INIT$, $TRANS$, $INVAR$, $FLOW$;

The declarations VAR and VAR_i may also contain module instantiations, which are variables with type as a tuple $\langle M, \beta \rangle$, where M is a module and β is used to associate the formal parameters. The extra parameter δ is for the continuous elaboration of the predicates in the vector field.

The semantics of a module is defined recursively on the structure, without circular dependency in the instantiations. In case the root module M has no module instantiations the semantics of a HyDI program is defined by the SHS $\langle D \cup P_d, X \cup P_c, Ev, IN, T, Z, F \rangle$, where $P = P_d \cup P_c$ is the set of input parameters of the module. Otherwise if there are module instantiations $\mathcal{I}$, so that for each instance $i \in \mathcal{I}$ there is a module instantiation $\langle M_i, \beta_i \rangle$, we consider recursively the SHS $S_{M_i} = \langle D_{M_i}, X_{M_i}, Ev_{M_i}, IN_{M_i}, T_{M_i}, Z_{M_i}, F_{M_i} \rangle$ specified for M_i. For an instance i with type $\langle M_i, \beta \rangle$ we define the SHS S_i as $\langle D_i, X_i, Ev_i, IN_i, T_i, Z_i, F_i \rangle$ with:

- D_i is defined taking the set D_{M_i} without the formal parameters of M_i and renaming the other variables $d \in D_{M_i}$ in $i.d$;

- X_i is defined renaming all the variables $x \in X_{M_i}$ as $i.x$;
- Ev_i is defined taking the set Ev_{M_i} and renaming all the variables $e \in Ev_{M_i}$ as $i.e$;
- IN_i is defined taking the set IN_{M_i}, renaming all the variables v as $i.v$ and replacing the parameters p of M_i by $\beta(p)$;
- T_i is defined taking the set T_{M_i}, renaming all the variables v as $i.v$ and replacing the parameters p of M_i by $\beta(p)$;
- Z_i is defined taking the set Z_{M_i}, renaming all the variables v as $i.v$ and replacing the parameters p of M_i by $\beta(p)$;
- F_i is defined taking the set F_{M_i}, renaming all the variables v as $i.v$ and replacing the parameters p of M_i by $\beta(p)$.

We define the SHS S of M by the tuple $\langle D \setminus \mathcal{I} \cup \{i.d \mid i \in \mathcal{I}, d \in D_i\} \cup P_d, X \setminus \mathcal{I} \cup \{i.x \mid i \in \mathcal{I}, x \in X_i\} \cup P_c, Ev, IN \wedge \bigwedge_{i\in\mathcal{I}} IN_i, T \wedge \bigwedge_{i\in\mathcal{I}} T_i, Z \wedge \bigwedge_{i\in\mathcal{I}} Z_i, F \wedge \bigwedge_{i\in\mathcal{I}} F_i\rangle$. The SHS relative to the root *main* module gives the semantics of a HyDI program.

5.3.3 Specification of Properties

During execution of HyDI programs constraints are verified, which are defined by the $INVAR$ declaration. In this declaration can be referred both discrete and continuous variables in a state expression, this constraint must be satisfied in all the states of the system. We can restrict invariant that must hold only in a location, specifying it in the expression.

For example, consider a property for the brake train controller, if we define the states of the hybrid i/o state machine with an enumerator variable as $\{acceleration,\ brake, constant\_speed\}$, then an invariant for the state *brake* can be:

```
INVAR
 state = brake -> accel <= 0
```

That is, if the state brake is active then the acceleration of the train must be negative. In Chapter 6, we provide the whole HyDI program for the train controller example.

5.3.4 Related Work

Modelling languages as SHIFT and R-Charon [86] for distributed hybrid systems are focused on simulation and compilation and they have no verification support. For some modelling approaches, simulation is provided, but a limited verification or only on fragments. As for instance in UPPAAL [16], where the properties do not support the nesting of formula. The verification with the tool PHAVer [15] is computationally expensive and suffers from the system scalability issues.

Davoren und Nerode [50] applied a Hilbert-style calculus with an extension of μ-calculus to hybrid systems. In this approach, the system behaviour cannot always be represented through the used propositional modal logic. Solutions based on real-time temporal logics, as for instance in [119], are not able to represent the full dynamics of hybrid systems, especially the part of the behaviour represented by the differential equations.

Model checking techniques based on invariant techniques permit to perform the verification without an explicit reachability search; moreover, they do not need solutions of the differential equations [119, 112]. In [49] the authors define the fundament for proving temporal specifications of discrete-time hybrid systems. Checking if a hybrid system from a set of initial states reaches a predefined set of states is called (in verification) symbolic reachability analysis. The first model checker to implement this analysis, restricted to linear hybrid systems is HYTECH [9]. To facilitate this analysis the tool CHECKMATE [34] computed an overapproximation of the reachable sets through polyhedral representations. SpaceEX is a tool that implements the reachability analysis with a scalable approach, capable to analyse a complex helicopter controller [60]. Flow* is a tool for modelling and verification of hybrid systems defined by nonlinear ODEs [33]. The tool computes Taylor model flowpipes, which are over-approximations of the reachable states in a step, supporting an adaptive or fixed step size.

One of the first comprehensive formal verification approaches, based on deductive verification, has been proposed by Platzer [105]. The verification combines first-order structures, compositional verification logic and proof calculus. The specifications are expressed through first-order logic formulas. Hybrid systems are modelled as hybrid programs, where discrete and continuous dynamic interact. The discrete part is constructed with first-order discrete jump formulas, while the continuous evolutions are first-order differential algebraic formulas. The author defines a differential dynamic logic for the specification and verification of hybrid systems, not limited to linear systems. The automated and interactive theorem prover KeYmaera [106] is

based on differential dynamic logic and its verification calculus. This approach can offer a complete verification of the systems; however, it is not a fully automatic procedure, as for instance the model checking with NuSMV, and may require certainly specific skills to be used. It would be also difficult to implement specification patterns or other high-level property specifications for limiting its overall usability.

Abstraction of the system model is a successful method to reduce the inherent complexity of the verification. A technique also used for hybrid systems is predicate abstraction [7, 38], which constructs finite state models from potentially infinite and complex state systems. The authors of [79] apply an interval arithmetic for over-approximation of nonlinear hybrid systems. It is also presented a supporting tool called hydlogic, which introduces Satisfiability Modulo Theories (SMT) for the bounded model checking of nonlinear systems. With this tool is it possible to verify systems with nonlinear ODEs and nonlinear guard constraints.

Chapter 6

Modelling Case Study

In order to show the modelling characteristics of our hybrid components and their simulation, we consider a case study based on the European Train Control System (ETCS) standard[1]. We show with this case study the necessity to extend the Focus theory from discrete time models to continuous ones. The ETCS works also as collision avoidance protocol, applied with a fully automated and coordinated movement of trains. The system is influenced from data provided by trackside infrastructure called Radio Block Centers (RBC). The train routes are subdivided in tracks, where there is an RBC for each of it. The position of all trains in a track are monitored by the correspondent RBC, which authorizes the trains to go freely until a stop point, called End-of-Authority point (EoA).

To avoid that trains go beyond the actual EoA point, each train has an Automatic Train Protection system (ATP) on-board to monitor such risk. If the ATP detects a collision with the EoA, it will take control of the speed system enforcing a break action in order to stop the train ahead of the EoA. The ETCS has different levels of application; in level three, the EoAs are moved forward, as soon as a safe distance between the train ahead and the successor train is reached. The traditional method for the collision avoidance implements interlocking principles, which are based on static partitions of the tracks into blocks, where the trains can move following interlocking protocols. Instead the ETCS is based on a "moving block principle"; where the RBCs protects each train along the route. In case, that the EoA is a railroad crossing the train has to lock the railroad crossing before the train will reach this point.

We build our model using a variant of the ETCS level three protocol, pre-

[1] `http://www.era.europa.eu`

sented by Damm et al. [48] for introducing their formal verification methodology. They extended the protocol protecting track segments in case a train moves into a segment before getting the permission. The system dynamics is modelled in MATLAB Simulink. The model of the dynamics is subdivided in three parts: mechanical transition, outer conditions, and the control part of the ETCS protocol, all modelled in Stateflow.

We build our case study based on these mentioned models. We construct the system with FOCUS hybrid components. A state machine defines the internal behaviour of each component and i/o channels handle the intra-component communication. Both the automaton and the streams may have discrete or continuous behaviour, showing the *hybrid* nature of the case study. In this chapter we want to achieve the following goals: (i) provide a sufficiently complex example, where is it possible to show a complete architecture of hybrid components and their interfaces; (ii) show the necessity and opportunity to have discrete and continuous streams with different scopes; (iii) provide a running example of our dynamic sampling solutions; and (iv) build a logical description of systems that can be understood by experts of different disciplines.

6.1 Modelling of the ETCS

In ETCS level 3, trains move in safe blocks through an automatic and decentralized interlocking scheme, where each RBC controls its block with a radio based control system. The train movements in a territory, subdivided into a fixed number of track segments, are supervised and controlled by an RBC. Trains can move freely until the end of movement authority (EoA) through the authorisation of the RBC. At each point of time, there exists an EoA, which may be the end of a track segment, an unsafe point, or the end of a train that is ahead. This way a safety block surrounding the train is defined. In practice, this moving system dynamically resets the track behind the train. In Figure 6.1, there is a graphical representation of the ETCS protocol. The variables for the distances are explained in the following models. The picture shows a scenario where the EoA ahead of the train is a railroad crossing. We model the control part of the ETCS using four FOCUS components (see Figure 6.2), some of them requiring a continuous update of variables thus being are hybrid components. In the following figures, the discrete channels are coloured in black, while the continuous are in blue. The component whose automata have states with continuous dynamic are in blue, while the other components are in grey.

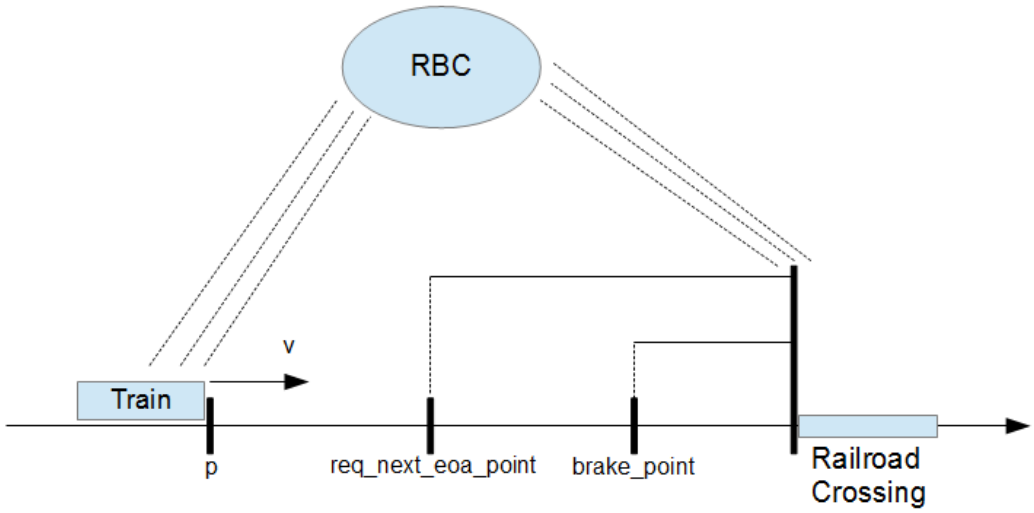

Figure 6.1: Schematic representation of the ETCS System.

6.1.1 RBC Request

The first component represents the RBC Request for communication between the train and the RBC and is presented in Figure 6.3. The internal state machine has an initial state that is only left if on the input channel *active* an activation message is sent. This means that this ETCS supervises the train. When this transition is fired, the output channels *train_pos* and *train_id* are set respectively to the current train position p and the id of the controlled train. At the same time a request for a new EoA is sent to the RBC, a timer is reset and the state machine evolves in the *rbc_send* state. In this state through the loop transition, a request is transmitted again after the time interval of a transmit action but before the maximal time has been reached. The *rbc_wait* state is reached and a brake service message with the output message *brake_req := Present* is sent, if no message is received before the maximal delay period. The message will be read from the *move* component, described in the following sections. If an acknowledge is received before the upper bound the other transition to the *rbc_wait* state is explored, and a request message through the output channel *calc_req* is sent. In the *rbc_wait* state there is only one transition that is taken, after the active message from the input channel *next_eoa_req* and at the same time in the output port an activation message *eoa_req* is received. With this transition, the system produces a request for travelling to the next segment, which will be sent to

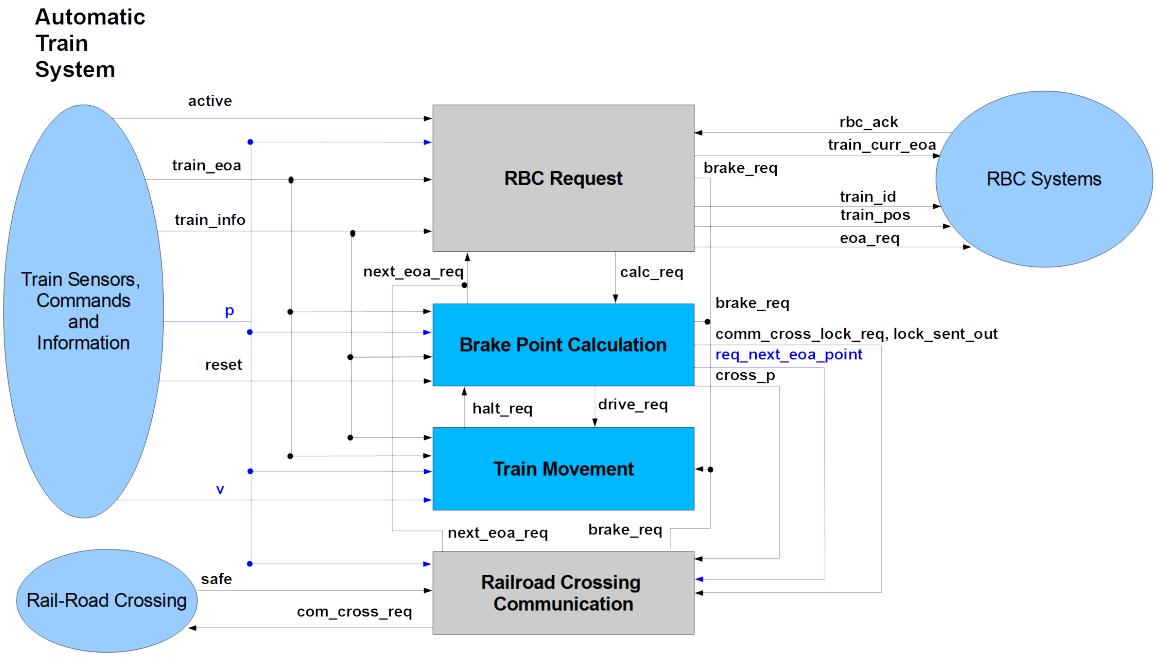

Figure 6.2: ETCS system in the train modelled in FOCUS.

the RBC in the output port *train_curr_eoa*. The request to the RBC is made with the following output messages as parameters: the current position of the train *train_pos*, the current EoA *train_curr_eoa* and the train id *train_id*. The new EoA will be calculated by the RBC considering the current train position, the current EoA and the positions of possible other trains ahead.

Continuous Dynamic The only element elaborated in continuous time is the input port p for the actual train position.

6.1.2 Railroad Crossing Communication

The second component represents the communication between the train and a railroad crossing. The component and the behavioural state machine are in Figure 6.4. From the initial state the transition to the state *x_com_point* is fired, if there is an activation message in the input channel *x_cross_lock*. This means that there is the necessity to activate the railroad crossing and a lock message is sent to the output port *com_cross_req*. In the *x_com_point* state, the transition to the safe state is fired if the train has not reached

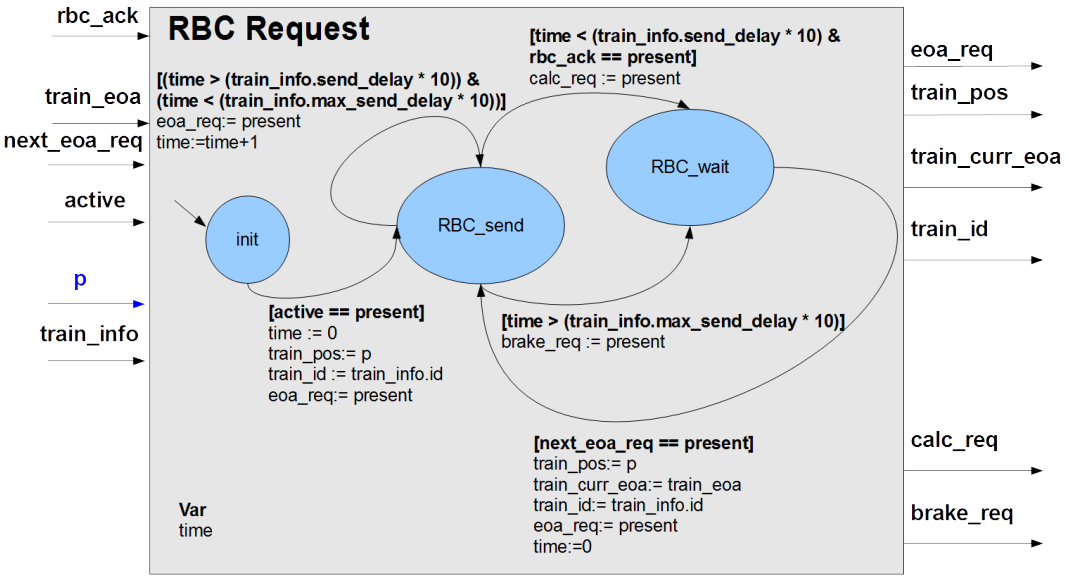

Figure 6.3: Component for the communication between train and RBC unit.

the start of communication point to rbc, *req_next_eoa_point*, and the level crossing has communicated the safe message through the input channel *safe*. With this transition, a new EoA from the RBC is requested and the local variable *lock_sent* is set to one indicating that the train has already sent a lock message to the railroad crossing. If in the *x_com_point* state the train is **behind** the start of communication to RBC point and there is an unsafe message in the input channel *safe*, then the transition to the unsafe state is fired and a brake message is transmitted to the move component. There is only one possibility to leave the *unsafe* state, that is, by switching off the automatic mode of the train, therefore this state is a blocking state in our model. The transition to leave the *safe* state is fired if the train position is behind the position of the railroad crossing. With this transition an unlock message to the railroad crossing is transmitted. This way the *init* state is reached again, being ready for the next request.

Continuous Dynamic There are two input channels with a continuous dynamic, which represent the train position and the point to initialize the request for a new EoA.

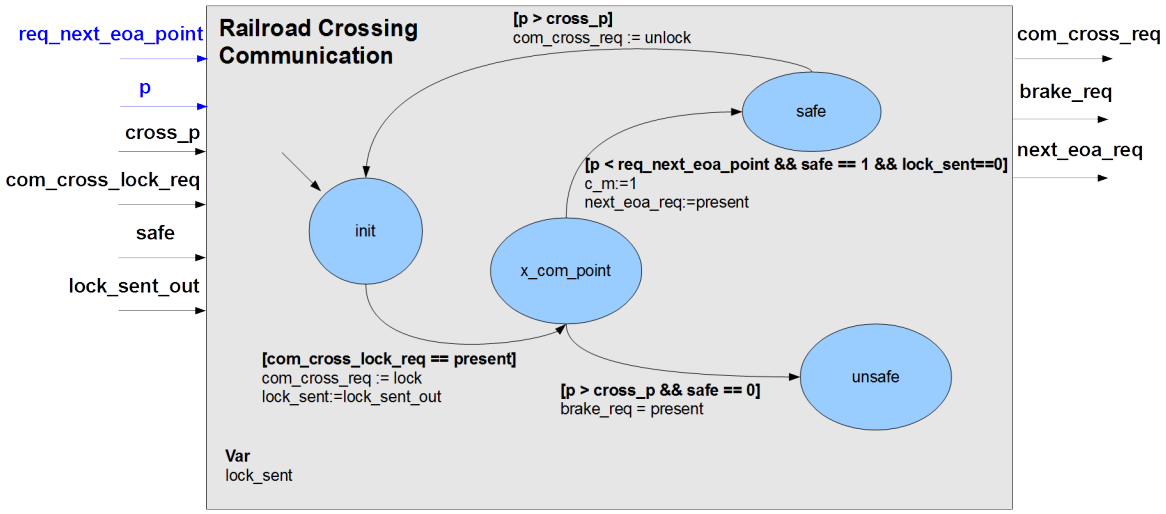

Figure 6.4: Component for the communication between the train and railroad crossing component.

6.1.3 Brake Point Calculation

This component is crucial for the safety of the train movement, as performs the calculation of the brake point. The component is depicted in Figure 6.5. There is only one continuous state *brake_point*, where the variables *brake_point* and *lock_req_point*, and the output channel *req_next_eoa_point* are elaborated, through linear differential equations and updated every time the state is active. The variable *brake_point* is the point in which the service brake is initialized. This point has to be updated constantly in order to ensure safe breaking, and guarantee that the train will stop before the EoA point. The current speed of the train and the deceleration produced by the brake determine the safe distance. At every time interval the point in which the brake has to be activated is calculated, as modelled in [48], by the equation:

$$brake\_point = EoA - \frac{1.1 \cdot v^2}{2 \cdot b}$$

in the formula v is the current speed of the train and b is the deceleration of the service brake. To guarantee a 10% of safety margin it is multiplied by the factor 1.1. In the scenario, the train will not brake at every EoA, because it asks the RBC for a new EoA before entering the breaking mode. In order to obtain this situation, the request should be sent in advance enough to avoiding the breaking initialization at point *brake_point*.

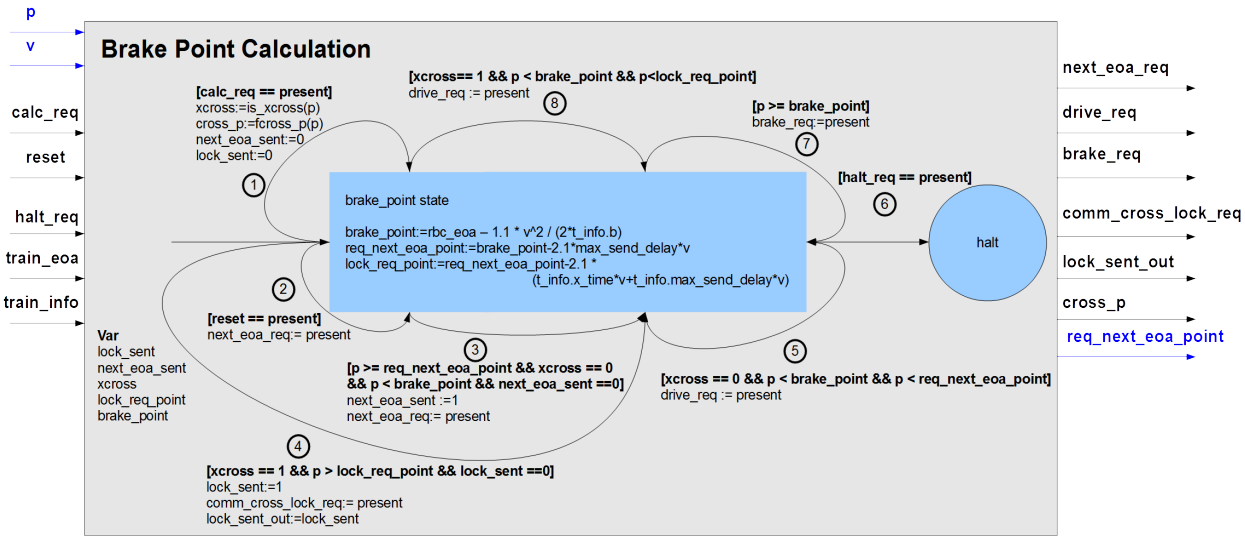

Figure 6.5: Component for the calculation of the train brake point.

The time for sending a new request is calculated considering the maximal send delay to the RBC and the response time to a request. So the point is calculated from the point to initialize the brake minus two times the delay for the velocity of the train:

$$req\_next\_eoa\_point = brake\_point - 2.1 \cdot max\_send\_delay \cdot v$$

The variable *lock_req_point* is the point to initialise a lock request to a railroad crossing:

$$lock\_req\_point = req\_next\_eoa\_point - 2.1 \cdot (train\_info.x\_time + max\_send\_delay) \cdot v$$

The field *x_time* from the input port *train_info* represents the time to set up the railroad crossing in a safe state. In fact if the EoA is a railroad crossing the train has to lock the crossing point in time, so it has to send a lock request and receive an acknowledge. This scenario is shown in Figure 6.1. The acknowledge from the Communication Railroad Crossing component is received in the *safe* input channel. Afterwards a new EoA is requested.

Transition number one is fired after an EoA message is received from the RBC. In this transition two local variables for counting the messages sent to the RBC and to the railroad crossing, respectively *next_eoa_sent* and *lock_sent* are initialized. The information if a railroad crossing is ahead of the train position p is initialized in the local variable *xcross*. Likewise, the

position of the crossing is copied to the local variable *cross_p* from a function defined in the data dictionary. Transition number two is taken if there is an input message on the channel *reset* and a new EoA point is requested. The same request is sent in the transition number three, which is fired if the train has passed the point for an EoA-request and its position is before the point to start the service brake. If the train position is beyond to initialize a lock request (*lock_req_point*) and a railroad crossing is ahead the train, then transition number four is executed, where a request for a lock is sent to the railroad crossing. Transition number five sends a drive message to the component *train movement* to change the driving mode of the train. This transition is fired if no railroad is ahead of the train, the train has passed the position for starting the service brake, and the train has not yet reached the point for committing an EoA-request. Through transition number six, the state *break_point* is left and state *halt* will be enabled. This transition occurs if there is an emergency signal in the input port *halt_req* forcing a brake of the train. Transition number seven is fired if the train has passed the point to initialize the break point and therefore a request for the service brake is generated. Finally, transition number eight is executed if there is a rail crossing ahead of the train and a request for changing the drive mode to the component train movement is sent.

Continuous Dynamic This component has two continuous input channels for the position and velocity of the train, and one output channel that is the position to initialize a new EoA-request. In this component, we introduce *continuous states* that perform a set of linear differential equations in continuous local variables or output streams if they are active. Otherwise, the correspondent output channels produce *no value* and the local continuous variables are not modified. As explained in the previous section, the state *break_point* has three differential equations to calculate the three positions used from the system. These points are important and critical for the safety of the train and its passengers, because they are used, to determine the right point of initialization of the service brake.

6.1.4 Train Movement

The train movement component is responsible to control for the train velocity (see Figure 6.6). At start of the execution, variable *old_v_d* is set to zero. This variable holds the previous value of the desired speed. Transition number one is fired if an activation message is received at input port *drive_req*. Then the local variable *drive_control*, used for the drive control, is set to ”on”, the

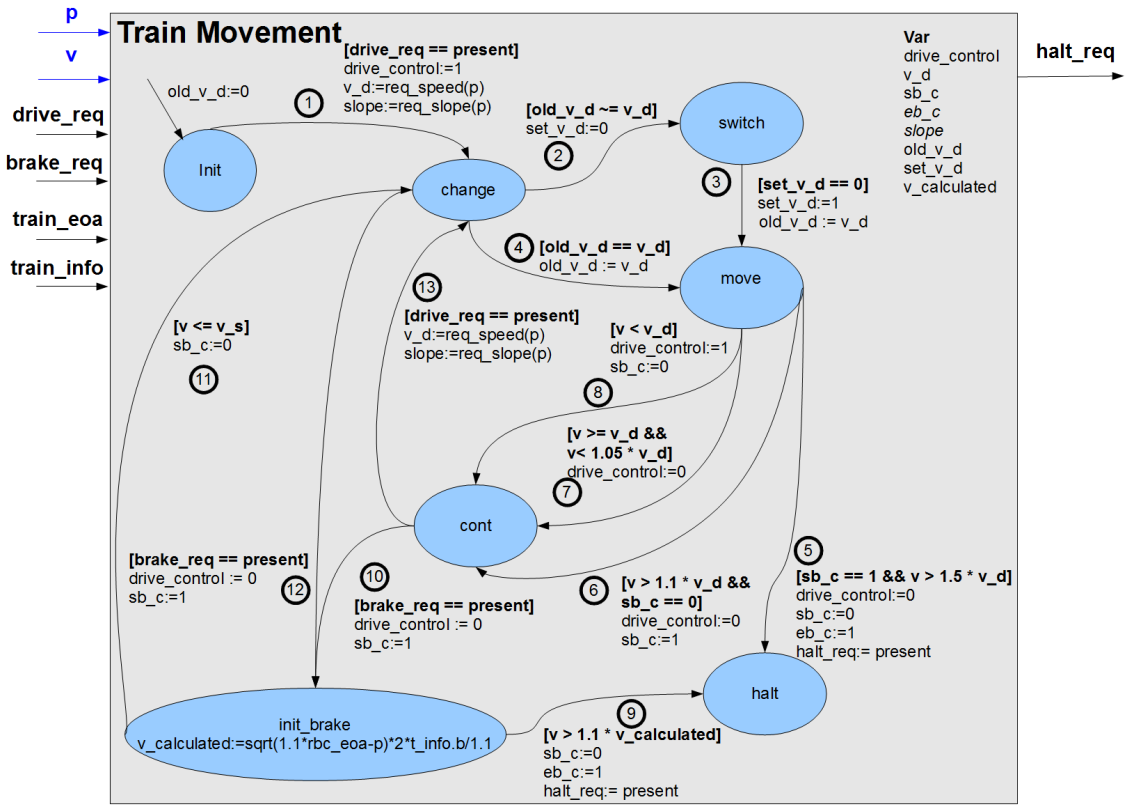

Figure 6.6: Component for the control of the train movement.

desired speed at the actual position is set in *v_d*, and the gradient of this track segment is set in *slope*. Both values are returned from the data dictionary. From this transition the automata evolves to the *change* state. In this state if the current desired speed is not equal to the previous value the transition number two to state *switch* is fired. This situation is marked with a reset of the variable *set_v_d*, which implies transition number three to state *move*, where the new value of the current desired speed is defined. Transition four to state *move* is fired if there is no variation of the desired speed.

In the *move* state, the train speed is handled. In fact, transition number eight is enabled if the current speed is below the desired speed, so the drive mode is switched on and the service brake is disabled. If the speed is equal or greater but not more than 5% over the desired speed, the drive mode is disabled and transition seven is taken. Transition six is enabled if the velocity is greater than 110% of the desired speed and if the breaking mode is not

enabled. Then with the activation of this transition, the driving mode is disabled and the service brake mode is enabled.

These last three transitions lead to the *cont* state. Transition thirteen from this state to state *change* is activated, if from the channel *drive_req* an activation message is received, also the slope and the desired speed are updated. With a *brake_req* message transition ten is taken. This way is set the breaking mode and *init_break* state is reached.

When the velocity is greater than 150% of the desired speed and the emergency brake mode is enabled then the transition five is fired. Thus, the state machine changes from state *move* to *halt* state and a halt message to the component brake point calculation is sent.

From the *change* state, transition twelve to the state *init_brake* is activated if the component receives an activation message in the input channel *brake_req*. Then the drive mode, storing zero in the variable *drive_control*, is disabled and the service brake mode switching on the variable *sb_c* is enabled.

The *init_break* state is a continuous state, which has a differential equation associated to the variable *v_calculated*. This variable contains the calculated speed to guarantee the breaking curve. If *v_calculated* is greater or equal than the current speed transition eleven to the *change* state is fired and the service brake mode is disabled. Otherwise if *v_calculated* is lower than the current speed transition nine is active, then the emergency brake mode is activated and the *halt* state is reached. At the same time, the emergency brake situation is communicated to the calculation brake point component.

Continuous Dynamic This component has two continuous input channels for the train position and speed, as other components. In the state *init_break*, the speed for controlling the breaking action of the service brake is calculated with a differential equation and in case the train speed is still too high, the emergency brake is activated.

6.2 Dynamic Sampling of the ETCS

In Chapter 4, we presented dynamic sampling solutions to simulate hybrid components in AutoFocus 3. To show a practical application of such sampling techniques, we consider the component Train Movement (explained in Section 6.1.4). The state *init_brake* calculates the continuous internal variable *v_calculated*. When the effective train speed is greater than this calculated speed plus 10%, the train will be slowed down with the emergency break. We

consider a simulation of this component in this state based on the concepts presented in Chapter 4. The discretization is implemented by approximating the continuous variable with the Runge-Kutta method. The period varies according to the two introduced methods.

The variable *v_calculated* is differentiated using the actual measured speed of the train and we consider how the dynamic sampling might approximate its values. We take the period variation based on the gradient to consideration and we show how can help the precision of the discretization. Between two simulation steps, the speed of the train may increase considerably or the speed sensor may have a delay to update its increasing value. In this case, a slope variation greater than the acceptance value determines the reduction of the sampling period; therefore, the component calculates more values of the variable *v_calculated* in the same time interval. As consequence, the guard of transition 9 is controlled more often and the component can signalise promptly an emergency going to the state *halt*. This adaptivity is an advantage of our dynamic sampling simulation over a static one. In addition, the sampling approach based on critical intervals can also guarantee an adaptable precision of the simulation. In this case, critical intervals are modelled. When the variable reaches a critical interval, corresponding for instance to a high not usual speed, then the period is set to a predefined value. Again, this value guarantees a fast response of the controller to a dangerous situation, activating in time the emergency brake.

6.3 Formal Verification of the ETCS

We presented a simple train controller in Section 3.4.3, which is a simplified version of the case study presented in this chapter. We translate the depicted hybrid component in Figure 3.6 of such a controller in to a HyDI program and we verify it with the first public release of the HyCOMP model checker (version number 1.0). The behaviour of this hybrid i/o state machine results in a compact HyDI program. As explained in the precedent chapter we convert each hybrid component to a HyDI module. Since our example has only one component, we have the *main* module and the *brake_controller_type* module. We report in the Listing 6.1 the resulting HyDI program.

Listing 6.1: HyDI program corresponding to the hybrid i/o state machine presented by the train brake controller case study (cf. Section 3.4.3).

```
MODULE main

 VAR breakPoint : -1..5000;
 VAR brake_controller: brake_controller_type(breakPoint);

 INVARSPEC NAME BRAKE_CONTROL := !(brake_controller.state !=
        acceleration & brake_controller.a > 0);

 MODULE brake_controller_type(s)

  VAR state: {acceleration, brake, constant_speed};
  VAR a: real;                  -- Acceleration is in m/s^2
  VAR z: continuous;
  VAR v: continuous;

  INIT
  state=acceleration & a = 0.0003 & z = 0 & v = 0

  DEFINE
  MaxSpeed := 0.0833333333; -- MaxSpeed is in m/s (300 km/h)

  INVAR
  (v >= 0) & (v <= MaxSpeed);

  TRANS
  state=acceleration & ((z >= s) & (s != -1)) ->
        (next(state) = brake) & (next(a) = -0.0005) &
        (next(v) = v) & (next(z) = z);
  TRANS
  state=acceleration & (v >= MaxSpeed) ->
        (next(state) = constant_speed) & (next(a) = 0.0) &
        (next(v) = v) & (next(z) = z);
  TRANS
  state=acceleration & (!((z >= s) & (s != -1)) &
        !(v = MaxSpeed)) -> (next(state) = state) &
        (next(a) = a) & (next(v) = v) & (next(z) = z);
```

```
TRANS
state=constant_speed & ((z >= s) & (s != -1) & (v > 0)) ->
      (next(state) = brake) & (next(a) = -0.0005) &
      (next(v) = v) & (next(z) = z);
TRANS
state=constant_speed & ((v = 0) & (s = -1)) ->
      (next(state) = acceleration) & (next(a) = 0.0003)
      & (next(v) = v) & (next(z) = z);
TRANS
state=constant_speed & (!((z >= s) & (s != -1) & (v > 0))
      & !((v = 0) & (s = -1))) -> (next(state) = state) &
      (next(a) = a) & (next(v) = v) & (next(z) = z);
TRANS
state=brake & (s = -1) -> (next(state) = acceleration) &
      (next(a) = 0.0003) & (next(v) = v) & (next(z) = z);
TRANS
state=brake & (v = 0) -> (next(state) = constant_speed) &
      (next(a) = 0.0) & (next(v) = v) & (next(z) = z);
TRANS
state=brake & (!(v = 0) & !(s = -1)) ->
      (next(state) = state) & (next(a) = a) &
      (next(v) = v) & (next(z) = z);

FLOW
der(v) = a & der(z) = v;
```

The *main* module encapsulates the *brake_controller* module in order to contain and instantiate input variables of the modules at the highest hierarchy, which represent the input channels of the components in the original FOCUS model. In fact, the *brake_controller* module has a parameter that corresponds to the input channel s, the break point constantly communicated from the railway control system. We instantiate in the *main* module the integer variable *breakPoint*, with an interval of values between -1 and 5000. In the verification phase, HyCOMP will execute the component with all the values in this interval. The output variables a, z, and v, and the states of the automaton are defined as variables with the keyword VAR and the initial state is defined with the expression after the keyword $INIT$. We define with $INVAR$ the invariants to express the physical limits of the train speed and that are the state invariants in the i/o FOCUS hybrid state machine. The transitions of the automaton and the update of the state are defined with the keyword $TRANS$. The evolutions of the continuous variables are specified

with $FLOW$, where the operator *der* is the derivative of the variable passed as parameter. We specify an invariant in the *main* module with the keyword $INVARSPEC$, which imposes that the acceleration variable "a" can have a positive value only if the actual state of the hybrid i/o state machine is the *acceleration* state. We verified the invariant of this HyDI program with HyCOMP 1.0 in circa eight minutes using a computer with an Intel Core i7-2720QM processor and 8 GB ram memory. Unfortunately, the actual version of HyCOMP was not stable enough to verify LTL formulas.

In Section 4.4.1.1, we modelled this simple train controller in a semantic equivalent model with MSS to evaluate our sampling algorithms. MSS integrates a formal verification environment, called Simulink Design Verifier (SDV), which offers exhaustive formal analysis of Simulink models with respect to specified assertions. We tried to verify the Simulink model of the brake controller with SDV executing a predefined compatibility test. We got an error message, because the model is not verifiable, since SDV does not support models that contain continuous-time integrator blocks. We used them to model the differential equations contained in the state machine. It seems that the verification process is dependent of the chosen sampling algorithm. Instead, HyCOMP proves HyDI programs representing i/o FOCUS hybrid state machines independently from its sampling process only considering the model execution in continuous time.

6.4 Conclusions

Continuous Dynamic In this case study we have introduced continuous streams and states, as new modelling elements for the discrete models defined in FOCUS. The theoretical foundation of such extensions are explained in Chapter 3. If we would have modelled only discrete the presented ETCS system fragment, it would have the following problems.

- Without continuous streams: it would be necessary to have a suitable value in milliseconds for the execution of a step action. This time must be enough long to avoid a dangerous delay in the handling of safety control, for instance the control for the activation of the emergency brake. In this way if there will be other requirements the discrete execution step should be again adjusted. However, that could lead to problems of different versions of the same model with different length of the execution step. With a very short step duration, it may be that other components do not need this simulation precision.

- Without continuous states: the elaboration through differential equations are replaced by a sort of discrete differential equations to be performed when the state is active. The elaboration of these equations will be also dependent on the discrete execution step, which has the same limitations explained in the precedent paragraph.

If we consider these problems, an extended modelling theory with continuous states and streams is more versatile and convenient, in a clear representation, modification and execution of the systems. We consider these extensions in the FOCUS modelling theory as a first step towards an integrated interdisciplinary design of CPSs.

Discrete Abstraction An advantage of our modelling concept is that we could abstract the model constructed without showing the continuous details. We can define a complete discrete view of the system, as for instance the model in Figure 6.2, just considering the components and the channels as discrete units. This way, the users can have a representation of the system through the components and their port connections, which represent the behaviour without showing the continuous state and variables inside the state machines and considering the streams as discrete. It is also possible to define requirements on the i/o streams, without considering the internal continuous dynamic.

Dynamic Sampling and Verification In Chapter 4, we studied sampling algorithms to adapt the granularity of the step to the behaviour of the continuous equations dynamically. We propose a dynamic sampling solution with approximation of differential equations that is thought for the modelling theory FOCUS and its implementation in the tool AutoFOCUS 3. We showed in this case study an example of hybrid component architecture and we made some considerations about its simulation in Section 6.2. The results that could provide our sampling algorithms encourages an integration of such solutions in AutoFOCUS 3.

Formal verification is another important aspect of the system design. In Section 6.3, we presented the automatic verification of a simplified component of the ETCS model. We could verify invariants, meanwhile the same model cannot be checked in MSS. The presented verification of hybrid components can also be easily implemented in AutoFOCUS 3, since we already provided a user-friendly integration of SMV/NuSMV for discrete time models and HyCOMP is part of the same family of model checkers.

Next Steps We specified the theory in Section 3 behind the continuous time and date extension for FOCUS, we showed in this section a case study with this principle applied, in the modelling, simulation and verification phases. These concepts are conceived to be put into practice in an implementation in the tool AutoFOCUS 3. The introduction in the modelling environment of the tool is an implementation issue; meanwhile more challenging topics are simulation and verification of the models.

Chapter 7

Conclusion

We presented an approach that comprises modelling, verification and simulation aspects for embedded and hybrid systems. We established our concepts over the modelling theory FOCUS, providing an extension that aims improving the support for modelling, simulation and verification of CPSs. We are aware of the intrinsic difficulties in a synthesis of a suitable modelling theory with formal verification capabilities for a complex domain as for CPSs. In fact, the design of these systems involves more disciplines and is strictly linked to the technical level and the hardware implementation.

The FOCUS theory has been conceived mainly for representing a system in formal models that are part of its logical viewpoint. In fact we refer to the SPES framework, where the development of systems is subdivided in different views or representations, namely the requirements, functional, logical and technical (Section 2.4). Our work focuses on the logical level, trying to propose a model representation that could be used by different experts involved in the design, such as computer scientists, electrical engineers, and mechanical engineers. The modelled system in the logical representation contains already behavioural characteristics, a system structure with typed interfaces, both being derived directly from the requirements and functional views. In some sense, we abstract in the logical view technical details and therefore the complex integration between the involved disciplines. The logical models have an important role in system engineering, since they permit the simulation of the behaviour of the system as well as its formal verification and validation. It is an important advantage that we can verify requirements and properties of the system, without the necessity to have already the final hardware implementation. This way the development life cycle is more effective and bugs can be detected earlier. In fact, changes in the logical models can be implemented more easily and cheaper than when it is discovered at the

technical representation. For these reasons, our work focuses on improving the artefacts of the logical view.

The presentation of our results starts with theory fundamentals, where we presented a brief classification of the systems, their development life cycle, and our methodology for the design of embedded systems. The FOCUS theory was designed originally for discrete components that operate in discrete time. We explained the basic elements that constitute the FOCUS theory, established in an embedded systems formalism. In the cyber-physical domain the involved physical components, the environment and the time precision may not have an optimal representation with discrete components and may require a continuous time and data type precision. Some systems can be represented with discrete FOCUS components because we can abstract the continuous dynamic of the system in discrete time. However, there are systems, especially safety-critical systems, which require continuous modelling artefacts and capabilities for their logical representation. For instance, the case study about a train system controller discussed in Chapter 6 requires differential equations to simulate and verify the reaction of the train with sufficient accuracy. Therefore, we extended the behaviour of the FOCUS components introducing differential equations and continuous variables. In computer science, there is a well-known formalism, which is used for defining systems with discrete and continuous dynamics. Based on these modelling structures we developed our notion of hybrid components. We defined hybrid i/o state machines as logical behaviour for FOCUS hybrid components. We extended the FOCUS models to obtain a more precise and better representation of CPSs, while using the streams theory, the abstraction and modularity of components available in FOCUS. We implement structure, interface and behaviour of the developed systems making it possible to simulate them. We studied the possibilities for a suitable simulation framework, and used AutoFOCUS 3 as a tool for demonstration. Since a full continuous simulation is too complex, we have studied sampling algorithms for the variables used in hybrid components. We believe that the precision of the sampling algorithm should be adapted to the necessities at run-time, therefore we introduced dynamic sampling solutions. We implemented and performed some preliminary tests of our sampling algorithms. Another important aspect of our models is the possibility of a formal verification based on model checking. Based on concepts already implemented for model checking of discrete systems, we worked towards an extension of the same concepts for the hybrid components.

7.1 Contributions

Modelling. The first step of our work is the definition of the foundations to be used to model formally CPSs. As introduced with our methodology (cf. Section 2.4), there are different representations of the same system under development. The FOCUS models of our approach are used for a logical description of systems, which is usually preceded by requirements and functional representations, and is followed by an implementation or technical description. The contribution of this work is to extend the FOCUS modelling theory to support CPSs. We modify the well known hybrid automata paradigm as implementation for hybrid FOCUS components, in order to support continuous time and data. We defined hybrid components with the aim to provide a logical representation of the system that could be used and understood not only by computer scientists, but also by electrical and mechanical engineers. Since CPSs involve different fields of expertise, a goal of system engineering is to provide a logical representation of the system, which defines behaviour, structure, and clear interfaces as soon as possible in the development life cycle. The introduced models can be simulated and verified without the necessity of final implementation details. This way, we want to limit problems that may occur at the integration phase, when the artefacts produced by different domain experts are integrated to form a complete system. We provide a sort of free logical language that can be used for important aspects in the development such as design, simulation, and formal verification of system properties.

Real-time simulation and sampling. We studied the execution in continuous time of variables guided by differential equations and their simulation in a modelling tool like AutoFOCUS 3. There are difficulties to guarantee a full continuous simulation of hybrid systems, due to the intrinsic complexity to handle differential equations over continuous variables in a reasonable time. In fact, with a high number of variables, the complexity of the differential equations increases and it becomes difficult to provide a real-time precise simulation, since most computers do not have enough computation resources. Furthermore, a system modelled with hybrid components might for instance represent control software for embedded systems, so the final real-time execution will be performed again in a discrete time environment.

A well-known solution in literature are sampling algorithms. In practice, a variable that evolves its values in continuous time is sampled assuming a value only in discrete time. The time between a sampled variable value and the next value is called period. If the period remains constant during the

sampling, we speak of static sampling; otherwise, if the period can vary we speak of dynamic sampling. We studied a dynamic sampling approach for hybrid components, because we believe that the sampling algorithm has to increase or decrease the period length, to increase the precision of the simulation according to runtime needs. Therefore, we developed two different solutions for adjusting the change of the period length: one based on the gradient calculated between the actual and the precedent value of the variables; and another based on some predefined critical intervals for the values of the variables, that is, when a variable is in a critical interval then the period must be equal or smaller to a correspondent predefined acceptance period. With both solutions, a change of the period can increment or decrement its length.

The first solution tries to follow the sudden changes of the values of the variables. They might happen for two reasons: the new differential equation associated to the variable with respect to the old equation at the precedent elaboration step, when the automaton changes its discrete state; or a rapid change of the values of the variable with the same differential equation. In both cases, we provide a shorter period in order to have more sampled values and therefore more precision. When the gradient is stable again, the period can be incremented. In the second solution, when the value of a variable is in a predefined critical interval, the period is set to a correspondent value associated to each critical interval. When the variable is not in the critical interval any more, the period might be set again to a longer value. The dynamic sampling is applied to each variable of the component and the shortest required period is taken. We analysed the impact of the dynamic sampling in a model with an architecture of interconnected components. In this case, a component can propagate the need for a shorter period to other components connected to its input ports. In fact, these components have to set their period in order to guarantee, for the variables associated with the involved output ports, a sampling period that provides enough input values for the period of the receiving component. We implemented prototypically our sampling solutions using MATLAB and we executed preliminary tests. We compared our solutions with sampling algorithms in the industrial standard for model-based design of embedded systems: MATLAB Simulink/Stateflow (MSS). The tests show that our solutions have better time performance and a more stable execution, because the adaptation of the period length is not based on the error minimization of differential equations. In some scenarios, where time performance is more important than the precision of the simulation, our solutions can be preferred. More important is that in our solutions it is possible to specify the desired precision of the simulation in re-

gions of interest. Unfortunately, to the best of our knowledge we did not find benchmark sets specific for dynamic sampling algorithms. Furthermore is not possible to execute a customized sampling algorithm in the Simulink/Stateflow environment, but only in the MATLAB environment. Therefore, it was not possible to test the algorithms with Simulink/Stateflow models. In our approaches, setting the period length and its adaptation mechanisms can be done directly through specifying desired simulation situations. For instance, a system validation can be done specifying the desired precision of the simulation for the error states or events of the system. These parameters could be directly derived from system requirements.

Integration and study of formal verification techniques. In the development life cycle of hardware and software systems, a fundamental phase is verification. It is particularly important for CPSs, since they are often used in safety-critical domains. As explained in Chapter 5 there is a wide spectrum of different analysis techniques conceived for specific development phases or system domains. In this work, we considered and studied model checking techniques to achieve full automatic and exhaustive verification. Model checking is a formal technique based on rigorous mathematical foundations, which is not easy for the non-expert to handle. Anyway, for an optimal development environment, we need an automatic verification technique for seamless integration in the modelling tools and without complicated usage.

In order to overcome this problem we implemented a holistic and user-friendly verification environment with focus on the integration in the modelling tool AutoFocus 3. We integrated the model checker SMV/NuSMV in the modelling tool together with different property specification concepts, to help the user expressing the system properties. We support specification patterns, an assertion language, and property specification based on templates adapted from the model artefacts. Moreover, we include back-translation of counterexamples to the model level, e.g., counterexamples can be examined step by step in the AutoFocus 3 simulator. Our goal was to render model checking usable not only by skilled developers. In addition, we believe that a strong integration of formal verification in model-based development is an advantage. In our modelling tool AutoFocus 3, we implemented this approach for models that are executed in discrete time steps. Then we studied and tested a similar support of formal verification for the introduced hybrid components. The synchronous semantics of the SMV/NuSMV modules is an excellent representation for the AutoFocus components. Therefore, we considered extensions of NuSMV for hybrid systems with the model checker

HyCOMP based on the HyDI language. HyDI is a language that adds support for continuous variables and variable derivatives in the next state assignments, which represents a suitable model for hybrid components. We adapted the semantics of HyDI to hybrid systems, because it originally did not support hierarchical module definitions and executed the processes in one program asynchronously. The verification in HyDI is enabled through an SMT Solver that verifies invariants over the variables and states of the system. With the semantic correspondence between HyDI programs and hybrid components, we wanted to offer the same verification capabilities. We verified with HyCOMP formal assertions over continuous variables in a HyDI program derived from a hybrid i/o state machine in FOCUS. The same system modelled in Simulink/Stateflow was not verifiable by Simulink Design Verifier, because this integrated verifier does not support models that contain continuous-time integrator blocks.

7.2 Outlook and Future Work

This work establishes a foundation for comprehensive modelling and verification of CPSs. We aimed at solving criticalities with the development of these systems, essentially introducing and adapting model-based design concepts. One crucial point is to introduce these concepts not only for computer scientists, but also for other experts, as for instance electronics and mechanical engineers. This interdisciplinary nature of CPSs guarantees a wide spectrum of future evolutions for our contribution.

Modelling and Dynamic Sampling. The proposed extension of the FOCUS models can be considered as a first step for an integrated interdisciplinary design of CPSs. This logical representation is mainly conceived for control software components of CPSs. Further evolutions of the presented formal modelling theory are for instance the introduction of the support for product lines or instantiation of predefined modules/components. To improve the interdisciplinary of the models, the support for specific energy or material flows between the model parts/components is of particular interest. Through case studies and feedback from different discipline experts, it should to be examined whether the introduced models are valid logical representations or whether extra characteristics are needed, which are not considered yet.

A problem to be addressed in the final implementation of our sampling algorithms is the set of adequate periods at run-time in a network of components, the lack of benchmark sets as well as the difficulties to evaluate

them. Industrial case studies need to be performed to prove our approaches and define eventually better heuristics. It would also be advantageous to execute these cases not only in MATLAB but also in the AutoFocus 3 tool. Therefore, the realization of an implementation of our dynamic sampling algorithms in AutoFocus 3 will be a fundamental topic.

Formal Verification. We provided a semantical correspondence between HyDI programs and Focus hybrid components, and executed preliminary tests to verify assertions through SMT solving. A modification of HyCOMP to support hybrid components according to the provided semantics, the resolution of some limitations of the actual implementation, and an integration in a supporting tool as e.g. AutoFocus 3 are beneficial. These steps would be necessary to obtain an integrated modelling and verification environment.

(Semi-)Automatic Determination of Dynamic Sampling Parameters. It will be interesting also to use the HyCOMP verification process to determine acceptance values for the sampling based on gradients. For instance in order to find automatically the most convenient acceptance value for a variable, we can with HyCOMP systematically verify which maximal values are reached by the variable. Other automatic or assisted ways to guide the determination of parameters of the simulation could be studied. In this way, our algorithms can use information about the behaviour of continuous variables in an infinite time interval using formal verification. Meanwhile, existing approaches as for instance the sampling in the MSS environment usually check the variables behaviour with a finite time horizon.

Appendix A

Linear hybrid systems

A *linear term* α over a set of variables V is a linear combination of the variables in V with rational coefficients. A *linear formula* ϕ over V is a boolean combination of inequalities between linear terms over V. The following definition, which specifies linear hybrid systems, is reported from [4].

Definition 25 (Linear hybrid system). *A hybrid system $A = (V_D, Q, \mu_1, \mu_2, \mu_3)$ is linear if its activities, exceptions, and transition relations can be defined by linear expressions over the set V_D of data variables:*

1. *For all locations $\ell \in Q$, the possible activities are linear functions defined by a set of differential equations of the form $x' = k_x$, one for each data variable $x \in V_D$, where k_x is a rational constant: $f \in \mu_1(\ell)$ iff all $t \in \mathbb{R}_+$ and $x \in V_D$, $f(t)(x) = f(0)(x) + k_x \cdot t$. It is written $\mu_1(\ell, x) = k_x$ to define the activities of the linear hybrid system A.*

2. *For all locations $\ell \in Q$, the exception is defined by a linear formula ϕ over V_D : $\sigma \in \mu_2(l)$ iff $\sigma(\phi)$.*

3. *For all location pairs $e \in Q \times Q$, the transition relation $\mu_3(e)$ is defined by a guarded set of assignments*

$$\phi \rightarrow \{x := \alpha_x | x \in V_D\}$$

where ϕ is a linear formula over V_D and each α_x is either a linear term over V_D or "?": $(\sigma, \sigma') \in \mu_3(e)$ iff $\sigma(\phi)$ and for all $x \in V_D$, either $\alpha =?$ or $\sigma'(x) = \sigma(\alpha_x)$.

To indicate that the value of the variable x is changed nondeterministically to an arbitrary value is denoted by $x :=?$.

Are some special cases of linear hybrid systems of particular interest are:

- x is a *discrete variable* if $\mu - 1(l, x) = 0$ for each location $\ell \in Q$. Therefore, a discrete variable changes only when the location of control changes. We can conclude that a *discrete* system is a linear hybrid system where all data variables are propositions and clocks.

- A discrete variable x is a *proposition* if $\mu_3(c, x) \in \{0, 1\}$ for all pairs $c \in Q \times Q$. If all the data variables are proposition then a linear hybrid automaton is same as a finite-state system whose states are labelled with propositions.

- If $\mu_1(\ell, x) = 1$ for each location ℓ and $\mu_3(e, x) \in \{0, x\}$ for each pair $e \in Q \times Q$, then x is a *clock*. Therefore, the value of a clock increases with time uniformly; a transition of the automaton either resets it to 0, or leaves it unchanged. A (finite-time) *timed* system is a linear hybrid system where all data variables are propositions and clocks.

- If there is a constant $k \in \mathbb{R}$ such that $\mu_1(\ell, x) = k$ for each location ℓ and $\mu_3(e, x) \in \{0, x\}$ for each pair $e \in Q \times Q$, then x is a *skewed clock*. Thus, a skewed clock is similar to a clock variable except that it changes with time at some (fixed) rate different from 1. A *multirate timed* system is a linear hybrid system where all data variables are propositions and skewed clocks. An n-rate timed system is a multirate timed system whose skewed clocks proceed at n different rates.

- If $\mu_1(\ell, x) \in \{0, 1\}$ for each location ℓ and $\mu_3(e, x) \in \{0, x\}$ for each pair $e \in Q \times Q$, then x is an *integrator*. Hence an integrator is like a clock that can be stopped and restarted, and can measure accumulated durations. An integrator system is a linear hybrid system where all data variables are propositions and integrators.

- A discrete variable is a *parameter* if $\mu_3(e, x) = x$ for all pairs $e \in Q \times Q$. Thus, a parameter is a symbolic constant, which can be used, for instance, in the guards of the transitions. For different special types of linear hybrid automata defined above, we can also define its parameterized data variables version. A *parameterized timed* system is a linear hybrid system where all data variables are propositions, parameters, and clocks.

In [4] a verification approach to check invariant properties applicable only to linear hybrid automata is presented.

Proof Lists

List of Figures

Bibliography

[1] IEEE Std 1012 - 2004 IEEE Standard for Software Verification and Validation. Technical report, 2005.

[2] Amos Albert. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. *Embedded World*, 2004.

[3] Hassane Alla and René David. Continuous and Hybrid Petri Nets. *Journal of Circuits, Systems, and Computers*, 8(1):159–188, 1998.

[4] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. pages 209–229. Springer-Verlag, 1992.

[5] Rajeev Alur, Thao Dang, Joel Esposito, Yerang Hur, Franjo Ivancic, Vijay Kumar, Insup Lee, Pradyumna Mishra, George J. Pappas, and Oleg Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems, 2003.

[6] Rajeev Alur, Thao Dang, and Franjo Ivančić. Counterexample-guided predicate abstraction of hybrid systems. *Theor. Comput. Sci.*, 354(2):250–271, March 2006.

[7] Rajeev Alur, Thao Dang, and Franjo Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *ACM Trans. Embed. Comput. Syst.*, 5(1):152–199, February 2006.

[8] Rajeev Alur and Thomas A. Henzinger. Modularity for Timed and Hybrid Systems. pages 74–88. Springer-Verlag, 1997.

[9] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[10] Rajeev Alur, Tom Henzinger, Gerardo Lafferriere, George, and George J. Pappas. Discrete Abstractions of Hybrid Systems. In *Proceedings of the IEEE*, pages 971–984, 2000.

[11] A. Anta and P. Tabuada. To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems. *Automatic Control, IEEE Transactions on*, 55(9):2030 –2042, sept. 2010.

[12] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *6th International Conference on Formal Engineering Methods (ICFEM'04, Proceedings)*, volume 3308 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2004.

[13] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[14] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT - Structured Assertion Language for Temporal Logic. In *ICFEM*, pages 757–775, 2006.

[15] 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. In *Journal of Logic and Algebraic Programming*, pages 129–210, 2006.

[16] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio 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'04, Proceedings)*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer, September 2004.

[17] Calin Belta, Volkan Isler, and George J. Pappas. Discrete Abstractions for Robot Motion Planning and Control in Polygonal Environments. *IEEE Transactions on Robotics*, 21:864–874, 2004.

[18] John J. Benedetto. Wavelets: A Tutorial in Theory and Applications. chapter Irregular Sampling and Frames, pages 445–507. Academic Press Professional, Inc., San Diego, CA, USA, 1992.

[19] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer.

[20] Robert K. Brayton et al. VIS: A System for Verification and Synthesis. In *8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.

[21] Manfred Broy. System Behaviour Models with Discrete and Dense Time. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, pages 3–25. Springer, 2012.

[22] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-I9202, Technische Univerität München, 1992.

[23] Glenn Bruns and Patrice Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *11th International Conference on Computer Aided Verification (CAV'99: Proceedings)*, pages 274–287, London, UK, 1999. Springer.

[24] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[25] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. Mcmillan, and David L. Dill. Symbolic Model Checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:401–424, 1994.

[26] Alarico Campetelli. Analysis techniques: State of the art in industry and research. Tech. rep., TU München, 2010.

[27] Alarico Campetelli. Dynamic Sampling for FOCUS Hybrid Components. In Peter Csaba Ölveczky and Cyrille Artho, editors, *Proceedings of the 3rd International Conference on Circuits, System and Simulation (ICCSS'13)*, volume 3(5), pages 402–406. International Journal of Modeling and Optimization, 2013.

[28] Alarico Campetelli, María Victoria Cengarle, Irina Gaponova, Alexander Harhurin, Daniel Ratiu, and Judith Thyssen. Specification techniques. Tech. rep., TU München, 2010.

[29] Alarico Campetelli, Alexander Gruler, Martin Leucker, and Daniel Thoma. *Don't know* for multi-valued systems. In Zhiming Liu and Anders P. Ravn, editors, *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, volume 5799, pages 289–305. Springer, 2009.

[30] Alarico Campetelli and Georg Hackenberg. Performance Analysis of Adaptive Runge-Kutta Methods in Region of Interest. In *2nd International IFIP Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC '15)*, 2015.

[31] Alarico Campetelli, Florian Hölzl, and Philipp Neubeck. User-friendly Model Checking Integration in Model-based Development. In *24th International Conference on Computer Applications in Industry and Engineering*, 2011.

[32] Alarico Campetelli and Maria Spichkova. Towards system development methodologies: From software to cyber-physical domain. In *Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems*, volume 105 of *Electronic Proceedings in Theoretical Computer Science*. Open Publishing Association, 2012.

[33] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An Analyzer for Non-linear Hybrid Systems. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 2013.

[34] Alongkrit Chutinan and Bruce H. Krogh. Verification of Polyhedral-Invariant Hybrid Automata Using Polygonal Flow Pipe Approximations. pages 76–90. Springer, 1999.

[35] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. HYDI: a language for symbolic hybrid systems with discrete interaction. In *EUROMICRO-SEAA*, 2011.

[36] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. Software Reviews: The State of the Practice. *IEEE Software*, 20(6):46–51, 2003.

[37] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. Abstraction and

Counterexample-Guided Refinement in Model Checking of Hybrid Systems, 2003.

[38] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Olaf Stursberg, and Michael Theobald. Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 192–207. Springer, 2003.

[39] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50(5):752–794, September 2003.

[40] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer.

[41] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[42] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 1999.

[43] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.

[44] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15, 1994.

[45] Tim Coe, Terje Mathisen, Cleve Moler, and Vaughan Pratt. Computational Aspects of the Pentium Affair. *Computing in Science and Engineering*, 2(1):18–31, 1995.

[46] Séverine Colin and Leonardo Mariani. Run-Time Verification. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2004.

[47] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. 19(1):45–80, 2001.

[48] Werner Damm, Alfred Mikschl, Jens Oehlerking, Ernst-Rüdiger Olderog, Jun Pang, André Platzer, Marc Segelken, and Boris Wirtz. Automating verification of cooperation, control, and design in traffic applications. In *Formal Methods and Hybrid Real-Time Systems. Volume 4700 of Lecture Notes in Computer Science*, pages 115–169. Springer, 2007.

[49] Werner Damm, Guilherme Pinto, and Stefan Ratschan. Guaranteed Termination in the Verification of LTL Properties of Non-linear Robust Hybrid. pages 1–13.

[50] J.M. Davoren and a. Nerode. Logics for Hybrid Systems. *Proceedings of the IEEE*, 88(7):985–1010, July 2000.

[51] Carl De Boor. *A practical guide to splines*. Appl. Math. Sci. Springer, New York, NY, 1978.

[52] Akash Deshpande, Aleks Göllü, Aleks Gollu, and Pravin Varaiya. Shift: A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata, 1997.

[53] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[54] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming Heterogeneity - The Ptolemy Approach. In *Proceedings of the IEEE*, pages 127–144, 2003.

[55] E. Allen Emerson. Model Checking and the Mu-calculus. In *Descriptive Complexity and Finite Models*, pages 185–214, 1996.

[56] M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, S. Rittmann, K. Scheidemann, M. Spichkova, and D. Trachtenherz. A Top-Down Methodology for the Development of Automotive Software. Technical Report TUM-I0902, 2009.

[57] M. Feilkas, F. Hölzl, C. Pfaller, S. Rittmann, B. Schätz, W. Schwitzer, W. Sitou, M. Spichkova, and D. Trachtenherz. A Refined Top-Down

Methodology for the Development of Automotive Software Systems - The KeylessEntry System Case Study. Technical Report TUM-I1103, 2011.

[58] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP - A Protocol Validation and Verification Toolbox. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, London, UK, 1996. Springer.

[59] A. Fleischmann. *Model-based formalization of requirements of embedded automotive systems.* PhD thesis, TU München, 2008.

[60] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable Verification of Hybrid Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

[61] Xiang gen Xia. A new prefilter design for discrete multiwavelet transforms. *IEEE Trans. Signal Processing*, pages 1558–1570, 1998.

[62] José M. E. González, Antonio Eduardo Carrilho da Cunha, José E. R. Cury, and Bruce H. Krogh. Supervision of Event-Driven Hybrid Systems: Modeling and Synthesis. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *HSCC*, volume 2034 of *Lecture Notes in Computer Science*, pages 247–260. Springer, 2001.

[63] Karlheinz Gröchenig. Reconstruction algorithms in irregular sampling. *Mathematics of computation*, 59(199):181–194, 1992.

[64] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I (2nd Revised. Ed.): Nonstiff Problems.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[65] D. Harel and P. S. Thiagarajan. Message Sequence Charts. In L. Lavagno, G. Martin, and B. Selic, editors, *UML for Real: Design of Embedded Real-Time Systems*, pages 77–105. 2003.

[66] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. *AT&T Bell Laboratories Technical Journal*, 69(1):45–59, 1990.

[67] W. P. M. H. Heemels, J. H. Sandee, and P. P. J. Van Den Bosch. Analysis of event-driven controllers for linear systems. *International Journal of Control*, 81(4):571–590, 2008.

[68] T. A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 278–, Washington, DC, USA, 1996. IEEE Computer Society.

[69] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 545–558. Springer, 1992.

[70] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Algorithmic Analysis of Nonlinear Hybrid Systems. *IEEE Transactions on Automatic Control*, 43:225–238, 1996.

[71] J. R. Higgins. Five short stories about the cardinal series. *Bulletin (New Series) of the American Mathematical Society*, 12(1):45–89, 01 1985.

[72] F. Hölzl. The AutoFocus 3 C0 Code Generator. Technical Report TUM-I0918, Technische Universität München, 2009.

[73] Florian Hölzl and Martin Feilkas. AutoFOCUS 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, pages 317–322. Springer Berlin / Heidelberg, 2011.

[74] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[75] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *CAV*, pages 209–213, 2008.

[76] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-Driven Program Testing. In *VMCAI*, pages 151–166, 2009.

[77] T. E. Hull, W. H. Enright, B. M. Fellen, and A. E. Sedgwick. Comparing Numerical Methods for Ordinary Differential Equations. *SIAM Journal on Numerical Analysis*, 9(4):603–637, December 1972.

[78] Silvana Ilie, Gustaf Söderlind, and Robert M. Corless. Adaptivity and computational complexity in the numerical solution of ODEs. *J. Complexity*, 24(3):341–361, 2008.

[79] Daisuke Ishii, Kazunori Ueda, and Hiroshi Hosobe. An interval-based SAT modulo ODE solver for model checking nonlinear hybrid systems. *Int. J. Softw. Tools Technol. Transf.*, 13(5):449–461, October 2011.

[80] Raymond Holsapple; Ram Iyer and David Doman. Variable step-size selection methods for implicit integration schemes for ODES. *Int. J. Numer. Anal. Model.*, 4(2):210–240, 2007.

[81] Abdul J. Jerri. The Shannon sampling theorem - its various extensions and applications: a tutorial review. *Proc. IEEE*, 65(11):1565–1596, 1977.

[82] Capers Jones. Software Quality in 1997: What Works and What Doesn´t. 1997.

[83] J. Baugh Jr., R. Cleaveland, and W. Elseaidy. Modeling and verifying active structural control systems. *Sci. Comput. Program.*, 29(1-2):99–122, 1997.

[84] Matt Kaufmann and J Strother Moore. Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification. *Spanish Royal Academy of Science (RAMSAC)*, 98:181–196, 2004.

[85] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, 1999.

[86] Fabian Kratz, Oleg Sokolsky, George J. Pappas, and Insup Lee. R-Charon, a Modeling Language for Reconfigurable Hybrid Systems. In João P. Hespanha and Ashish Tiwari, editors, *HSCC*, volume 3927 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2006.

[87] Robert P. Kurshan. The complexity of verification. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 365–371, New York, NY, USA, 1994. ACM.

[88] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, Princeton, NJ, USA, 1994.

[89] Raymond H. Kwong and Edward W. Johnston. A variable step size LMS algorithm. *IEEE Transactions on Signal Processing*, 40(7):1633–1642, 1992.

[90] Jan Lunze and F Lamnabhi-Lagarrigue. *Handbook Of Hybrid Systems Control: Theory, Tools, Applications*. Cambridge University Press, New York, 2009.

[91] John Lygeros. Lecture notes on hybrid systems. Technical report, 2004.

[92] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O Automata. pages 496–510. Springer-Verlag, 1996.

[93] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[94] Olaf Müller and Peter Scholz. Functional Specification of Real-Time and Hybrid Systems. In *In Proc. Hybrid and Real-Time Systems*, pages 26–28. Springer-Verlag, 1997.

[95] Thomas Moor, Jörg Raisch, and Siu O'Young. Discrete Supervisory Control of Hybrid Systems Based on l-Complete Approximations. *Discrete Event Dynamic Systems*, 12(1):83–107, January 2002.

[96] G. Nair, F. Fagnani, S. Zampieri, and R. Evans. Feedback control under data rate constraints: an overview. *Proceedings of The IEEE*, 95:108–137, 2007.

[97] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[98] Meeko Oishi, Ian Mitchell, Alexandre Bayen, and Claire Tomlin. Invariance-preserving abstractions of Hybrid Systems: Application to User Interface Design. *IEEE TRANSACTIONS ON CONTROL SYSTEMS TECHNOLOGY*, 2005.

[99] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & Systems (2nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[100] Martin Ouimet and Kristina Lundqvist. Formal Software Verification: Model Checking and Theorem Proving. Technical Report, Mälardalen University, March 2007.

[101] Soo-Chung Pei and Min-Hung Yeh. An Introduction to Discrete Finite Frames. *IEEE Signal Processing Magazine*, 14(6):84–96, Nov. 1997.

[102] Doron Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer.

[103] M. Petreczky, D. A. van Beek, J. E. Rooda, P. J. Collins, and J. H. van Schuppen. Sampled-Data Control Of Hybrid Systems With Discrete Inputs And Outputs. In A. Giua, M. Silva, and J. Zaytoon, editors, *Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems*. International Federation of Automatic Control, 2009.

[104] André Platzer. Differential-algebraic Dynamic Logic for Differential-algebraic Programs. *Journal of Logic and Computation*, 20(1):309–352, November 2008.

[105] André Platzer. Differential Dynamic Logic for Hybrid Systems. *Journal of Automated Reasoning*, 41(2):143–189, August 2008.

[106] André Platzer and Jan-David Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

[107] Amir Pnueli. A temporal logic of concurrent programs. In *Theoretical Computer Science 13*, pages 45–60, 1981.

[108] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy. *Model-based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer, 2012.

[109] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[110] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer.

[111] Daniel Ratiu, Wolfgang Schwitzer, and Judith Thyssen. A System of Abstraction Layers for the Seamless Development of Embedded Software Systems. Technical Report TUM-I0928, Technische Universität München, 2009.

[112] Enric Rodríguez-carbonell and Ashish Tiwari. Generating polynomial invariants for Hybrid Systems. In *HSCC*, pages 590–605. Springer, 2005.

[113] Dieter Rombach, Marcus Ciolkowski, Ross Jeffery, Oliver Laitenberger, Frank McGarry, and Forrest Shull. Impact of research on practice in the field of inspections, reviews and walkthroughs: learning from successful industrial uses. *SIGSOFT Softw. Eng. Notes*, 33(6):26–35, 2008.

[114] A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994.

[115] Valérie Roy and Robert de Simone. Auto/Autograph. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 65–75, London, UK, 1991. Springer.

[116] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *Proc. IEEE WESTCON*. IEEE Press, August 1970. Reprinted in Proc. Int'l Conf. Software Engineering (ICSE) 1989, ACM Press, pp. 328-338.

[117] Sriram Sankaranarayanan and Ashish Tiwari. Relational abstractions for continuous and hybrid systems. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 686–702, Berlin, Heidelberg, 2011. Springer-Verlag.

[118] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The Rhapsody UML Verification Environment. In *2nd International Conference on Software Engineering and Formal Methods*, pages 174–183. IEEE Computer Society, 2004.

[119] Pierre Schobbens, Jean Raskin, Thomas A. Henzinger, and L. Ferier. Axioms for Real-Time Logics. Technical report, Berkeley, CA, USA, 1999.

[120] C. E. Shannon. Communication in the Presence of Noise. *Proc. Institute of Radio Engineers*, 37(1):10–21, 1949.

[121] Sandeep K. Shukla, Tevfik Bultan, and Constance L. Heitmeyer. Panel: given that hardware verification has been an uphill battle, what is the future of software verification? In *MEMOCODE*, pages 157–158, 2004.

[122] M. Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. PhD thesis, 2007.

[123] M. Spichkova. Architecture: Requirements + Decomposition + Refinement. *Softwaretechnik-Trends 31:4*, 2011.

[124] M. Spichkova, F. Hölzl, and D. Trachtenherz. Verified System Development with the AutoFocus Tool Chain. In *2nd Workshop on Formal Methods in the Development of Software*, WS-FMDS, 2012.

[125] Standardization Sector of International Telecommunication Union. Message Sequence Chart. Technical Report Z.120, Genève, 2004.

[126] Josef Stoer, Roland Bulirsch, Richard H. Bartels, Walter Gautschi, and Christoph Witzgall. *Introduction to numerical analysis*. Texts in applied mathematics. Springer, New York, 2002.

[127] Sabine Teufl, Dongyue Mou, and Daniel Ratiu. MIRA: A tooling-framework to experiment with model-based requirements engineering. In *RE*, pages 330–331. IEEE, 2013.

[128] A. Tiwari. Abstractions for hybrid systems. *Formal Methods in Systems Design*, 32:57–83, 2008.

[129] M. Unser and A. Aldroubi. A General Sampling Theory for Non-ideal Acquisition Devices. *IEEE Transactions on Signal Processing*, 42(11):2915–2925, November 1994.

[130] Michael Unser, Akram Aldroubi, and Murray Eden. Polynomial spline signal approximations: Filter design and asymptotic equivalence with Shannon's sampling theorem. *IEEE Trans. Inform. Theory*, 38(1):95–103, 1992.

[131] Štephan Kozak and Juraj Števek. Improved Piecewise Linear Approximation of Nonlinear Functions in Hybrid Control. In *18th World Congress of the International Federation of Automatic Control*, pages 14982–14987, Milan, Italy, 1996.

[132] Lawrence G. Votta, Jr. Does every inspection need a meeting? In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on*

Foundations of software engineering, pages 107–114, New York, NY, USA, 1993. ACM.

[133] Gilbert G. Walter. A sampling theorem for wavelet subspaces. *IEEE Trans. Inform. Theory*, 38:881–884, 1992.

[134] Xiaofeng Wang and M.D. Lemmon. Self-triggered feedback systems with state-independent disturbances. In *American Control Conference, 2009. ACC '09.*, pages 3842 –3847, june 2009.

[135] Arthur G. Werschulz. *Computational complexity of differential and integral equations - an information-based approach.* Oxford mathematical monographs. Oxford University Press, 1991.

[136] B. Widrow and M. E. Hoff, Jr. Adaptive switching circuits. *IRE WESCON Convention Record*, 4:96–104, 1960.

[137] Daniel Zwillinger. *Handbook of Differential Equations, Third Edition.* Academic Press, November 1997.