VDA | Verband der
Automobilindustrie

FAT | Forschungsvereinigung
Automobiltechnik

FAT-Schriftenreihe 316

MULTIC-Tooling

MULTIC-Tooling

OFFIS e.V.

Eckard Böde
Werner Damm
Günter Ehmen
Martin Fränzle
Kim Grüttner
Philipp Ittershagen
Bernhard Josko
Björn Koopmann
Frank Poppen
Michael Siegel
Ingo Stierand

Das Forschungsprojekt wurde mit Mitteln der Forschungsvereinigung Automobiltechnik e.V. (FAT) gefördert.

Contents

1 Introduction

Advanced Driver Assistance Systems (ADAS) and Automated Driving (AD) functions comprise modular interacting software components that typically build upon a layered architecture, such as depicted in Figure 1.1. The two bottom layers belong to the "classical" reactive control domain. Above these, ADAS and AD functions realize higher-level control regimes, requiring often complex sensor processing and fusion, feature extraction and maintenance of (environment) models. The components in these layers are typically developed by different teams, using different tools for different functional purposes. This results in a heterogeneous mixture of modeling and programming languages with different underlying models of computation (MoC) employed for the various functional components. Another key characteristic of ADAS/AD design is the heterogeneity with respect to the interaction between the individual functions. While we typically find synchronized and rather linear data flows at the lower (reactive) layers, the situation is largely different at the higher layers and across layers. Here, we find complex interactions, including service-oriented interfaces and highly asynchronous data flows such as for sensor fusion.

It is the heterogeneity which makes the design of ADAS/AD functions a challenging task. They are in general functions with high criticality level, which calls for rigorous application of formal methods that ensure the system is working properly. This in turn requires design processes where the system is developed along well-specified requirements. Presently, we focus on the timing aspect as an important part of safety. For example, obstacles must be detected within a certain time-frame in order to plan and perform appropriate avoidance maneuvers. The major issues when developing such systems are the integration of components with heterogeneous underlying MoCs and heterogeneous interaction needs while guaranteeing the satisfaction of all requirements, and a coherent handling of timing properties. This does not only hold for the functional design, but along the whole development process down to the target platform, where further synchronization challenges imposed by technical limitations need to be addressed.

In order to address those challenges, a common approach is to embed the different components into a common semantic framework, thereby enabling analysis of the components using the same notion of

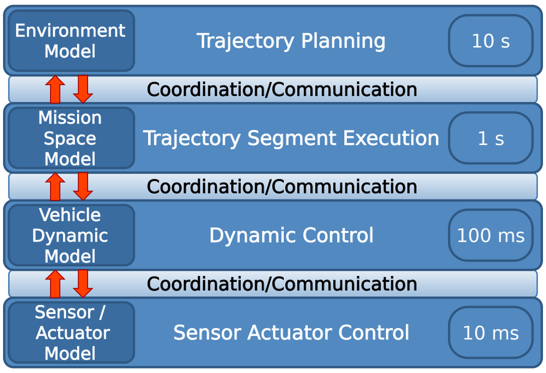


Figure 1.1: Layered Architecture for ADAS/AD

time. Such framework has been developed for example in the SPEEDS project [11], which builds upon a generic notion of components and contracts. Benefits in supporting the design process are elaborated e.g. in [4, 2]. Addressing similar objectives, four essential design and programming paradigms have been identified and elaborated in the MULTIC project with the German Association of the Automotive Industry (VDA) Working Group "Software and Electronics" of the Research Association for Automotive Technology (FAT) [3]. These paradigms, which are briefly introduced in Chapter 2, can be instantiated in various design processes using different modeling and programming languages.

An important objective of the MULTIC project was to cast these – conceptual – paradigms into an industrial context. To this end, the concepts have been exemplified by a case study where most of the proposed building bricks are applied to an automated driving function, including the well-defined proof obligations enabling safe and reliable reasoning about the design using industrially relevant modeling- and programming languages. To this end, a simulation based method has been developed for validation and verification of timing specifications, and has been manually implemented to perform the corresponding steps to show its applicability.

This document presents the results of the MULTIC-Tooling project, where the previous ad-hoc realization shall be replaced by an automated process that is based on a well-defined foundation. This mainly concerns the automatic translation of SysML models with their timing specifications into executable generators and monitors, which are in our case implemented in C++ code for the inclusion in a SystemC based simulation framework. To this end, an Eclipse-based tool has been implemented, which supports modeling and analysis according to these concepts. The tool integrates the Papyrus modeling environment, where engineers can design systems using the modeling language SysML, which we consider a particular instance of the compositional framework. The tool also allows expressing timing requirements in terms of contracts. For this, a pattern-based specification language has been defined in [3], enabling engineers to specify many relevant timing phenomena. Finally, the tool provides analysis of such models and requirements based on the SystemC [13] simulation framework, which allows for reasoning about valid refinement of specifications along the systems's decomposition, i.e., to perform *virtual integration testing* (VIT). To this end, the tool generates executable SystemC code from the input model, consisting of generators and observers according to the specified contracts. The generated code can be extended with user-defined function code, which enables functional analysis along the timing aspect.

Structure of the Document The document consists of two parts. The first part subsumes Chapter 3 – 6, and concerns the concepts and formal underpinning for the construction of executable generators and monitors for timing specifications. The second part provides an overview on the tool architecture, some interesting implementation details, as well as a list of test models used for the development of the tool.

The first conceptual part discusses in detail:

1. Syntax and Semantics of the MULTIC Timing Specification Language (MTSL).
2. Definition of a class of automata suitable to provide an operational semantics for MTSL specifications.
3. A language for specifying a sub class of these automata in an implementation-oriented way.
4. Corresponding generators and monitors for the elements of MTSL.

The first topic, the definition of MTSL, builds on the definitions made in the predecessor project MULTIC. It rephrases – with minor improvements and corrections – and extends the language according to the shortcoming identified in the MULTIC project. These extensions mainly concern first steps towards the definition of useful event relation functions needed for causal reaction and age patterns, and patterns that exploit local clocks. Compared to the previous version, the definition of trace composition has been revised at is was broken.

For the definition of a suitable automaton class, we also rely on previous work. The class of hybrid automata, developed by Thomas Henzinger in 1996 [7], has gained much attention since then as a universal formalism to specify and analyze many practically relevant systems. For example, translation schemes exist for Matlab Simulink models into hybrid automata for further exploration [1]. Also, there is considerable amount of analysis support for this system class, as for example elaborated in [12]. Finally, hybrid automata provide the formal basis for the definition of the contract theory introduced in the MULTIC project, able to capture all kind of variable evolutions. This document contains the original definition of [7] with only a mild modification in order to enable considering events and other variable evolutions in the same way.

Because the formal definition of hybrid automata is not well-suited for implementation purposes, the third topic concerns the definition of a more implementation-oriented specification language. It extensively borrows concepts from the programming language C++, yet makes some restrictions in order to maintain implementability. The resulting sub-class of automata is however expressive enough for our purposes (and many other). The fourth topic concerns the main goal of the present document, which is the provision of an operational semantics for MTSL in terms of the intermediate language.

The second part of the document, which is related to the implementation and usage of the tool consists of Chapter 7 – 11. It starts with a definition of so called Design Rules that impose particular constraints on the SysML models in order to be available for translation. The following parts concern the architecture of the frontend part of the tool, which is based on and extends the Eclipse modeling framework. Also some implementation details about the simulation backend are given for those who may modify it in the future. Chapter 10 and 11 are again rather for users of the tool, listing possible errors that may occur in the tool flow, and providing insights in the test models.

How to Read the Document The document provides information for three types of audience. For those who wish to use the tool, Chapters 7, 3 and 10 may be of interest. The former two explain what the input for the simulation should look like, while the latter gives insight of what may went wrong. Also the test models discussed in 11 may be useful, as they may serve as examples.

For developers of the tool and those who want to get an overview of how the tool is structured, Chapters 8 and 9 are of interest, discussing the overall architecture and implementation details. Also interesting details on how the simulation backend could be extended to include user-defined code is discussed in Chapter 9.

Formal background of what is happening behind the scenes, and why the tool is doing what it does, is discussed in Chapters 4 – 6. An overview of their content has been given above.

Additional Material The present document is complemented by a set of additional material. The tool prototype described in this document is provided pre-installed in a virtual box appliance. The appliance also contains the sample models described in Chapter 11. To make it easier to get started, there are two video tutorials in German and English. The first tutorial deals with the correct SysML modeling with focus on the Multic Design Rules (see Chapter 7) and the second video explains how to perform a *Virtual Integration Test*. Finally, the Doxygen documented source code is provided. Thus the possibility is given to adapt the prototype to the own needs or to develop it further.

2 Prerequisites

As this document builds upon the results of the predecessor project MULTIC, the reader is assumed to be at least partially familiar with the results reported in [3]. The focus of the MULTIC projects is on supporting methods and approaches to the engineering challenges imposed by the increasing complexity of functions and their interaction in today's and future automotive software applications. Advanced Driver Assistance Systems (ADAS) and Automated Driving (AD) comprise modular interacting software components that typically build upon a layered architecture. As these components are developed by different teams, using different tools for different functional purposes and building upon different models of computation, an integration of all components guaranteeing the satisfaction of all requirements is of major importance within the process of developing such systems.

System behavior typically subsumes different aspects, such as the functional aspect, timing, safety, and others. The MULTIC projects focus on the timing aspect, and a main objective of the first project was to elaborate on approaches that allow for capturing all relevant timing phenomena and effects for such systems in a consistent and coherent way across all system layers and functional domains, as well as ensuring traceability along the design process.

The project has identified four key design paradigms to support the development of future ADAS/AD, from which two are of particular importance in order to comprehend the content of this report. The first design paradigm is the **"Compositional Semantic Framework"**, which provides an architectural basis for system design by introducing a generic hierarchical component model. The model is intended to be instantiated with existing modeling languages (such as SysML) and tools by defining how to cast the individual modeling artifacts into artifacts of the conceptual model.

The component model serves as a carrier for the other three design paradigms:

- It defines a notion of contracts as a particular kind of (assume-guarantee style) specifications. Contracts give modeling entities and their interaction formal semantics, and enables one to reason about verification of the individual design steps, such as decomposition, refinement and realization.
- It supports the integration of different Models of Computation (MoC) for different parts within one system design.
- And it supports the integration of heterogeneous MoC using different abstraction of time through Converter Channel (CC).

The framework enables to set up design processes where systems are incrementally refined. It provides concepts allowing to relate different viewpoints like functional modeling and the technical realization, as well as different abstraction levels.

The second design paradigm **"Timing Specifications"** instantiates contract based design for the timing aspect of the system design. It inherits timing specifications from well established frameworks such as AUTOSAR, and defines extensions where needed in order to enable coherent reasoning about timing within complex scenarios. The timing specifications reported in [3] are rephrased (corrected and extended) in Chapter 3 of the present report.

A brief overview of these two paradigms and how they interact can be found in [3], Chapter 3. The semantic framework is extensively discussed in Chapter 4 in the same report. Chapters 10 and 11 extensively discuss a running example for the application of the two paradigms.

Because the semantic framework is not further discussed in the present report, we give a short introduction to its key elements in the remainder of this chapter. According to the framework, we assume systems to be built from *components* as depicted in Figure 2.1. Components may represent software functions, hardware elements or any other part of a system, depending on the design context. Components interact with their environment (such as other components) via *interfaces*. A component interface consists of a set of typed *ports*, where the type of a port defines which values can be observed at the port. Ports represent certain entities in the underlying (implementation) model of the component. For Matlab/Simulink models, for example, ports may represent input and output signals.

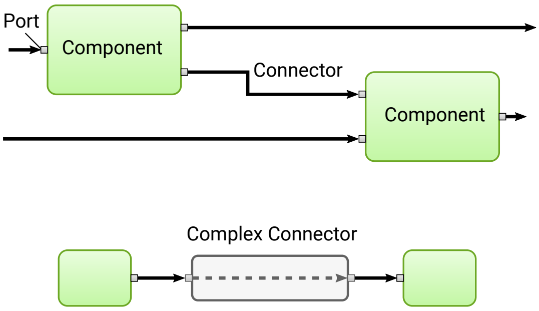


Figure 2.1: General Component Model

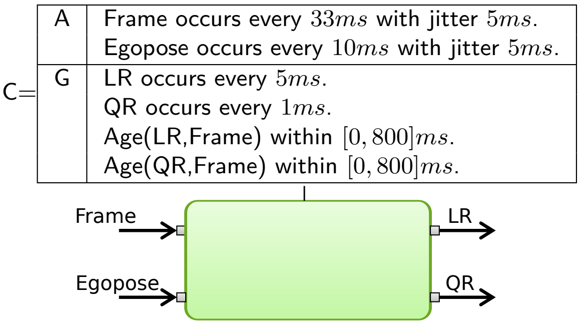
Ports of different components may be linked by *connectors*. We distinguish two types of connectors. A *simple* connector does not take time for transporting values between the connected ports. More precisely, a simple connector can be considered as an identification of the linked ports.

A *complex* connector, on the other hand, represents a physical transmission medium and imposes latencies as well as other complex behavior. Complex connectors are in fact also components as indicated at the bottom part of Figure 2.1. In this report, we restrict to simple connectors. Connectors are always directed such that data flows only in one direction. Service interfaces are modeled by sets of (combined) ports, which together realize the involved protocols.

In this framework, specifications about components are expressed in terms of *contracts*. Contract-based design is a paradigm, where specifications of design components not only express what a component is supposed to do, but also what the legal context for a component is. That means a contract expresses assumptions about the environment of a component. Further, it states a required behavior that must be guaranteed by the implementation of a component, provided it is used in a context that is compliant with the assumptions about the environment specified by the contract.

Figure 2.2 depicts an example of a contract that is expressed by instances of the timing specification language defined in [3], which is rephrased and extended this report. Both assumption and guarantee of the contract consists of a set of instantiated specification patterns (sentences). The assumptions (shown in the upper part of the figure, annotated with A) specify the context in which the component is expected to be executed. The first sentence states that some imaging source provides `Frame` inputs with a rate of $33ms$ and a jitter of $5ms$. The second sentence of the assumption states that the component gets an update of some `Egopose` every $10ms$ (and jitter of $5ms$).

The guarantee of the component (annotated with G) states that we observe updates of the control parameter `LR` every $5ms$, and of control parameter `QR` every $1ms$. The last two sentences of the guarantee serve two purposes. Firstly, the *Age* pattern specifies that every control parameter update refers to a concrete (and well defined) instance of the `Frame` input of the component. That means, every `LR` event, and `QR` event respectively, are a reaction to at least one (well defined) `Frame`. Secondly,



A	Frame occurs every $33ms$ with jitter $5ms$. Egopose occurs every $10ms$ with jitter $5ms$.
G	LR occurs every $5ms$. QR occurs every $1ms$. Age(LR,Frame) within $[0, 800]ms$. Age(QR,Frame) within $[0, 800]ms$.

Figure 2.2: Contract Example

it states that a reaction in terms of LR and QR control data should not be later than $800ms$ after the corresponding frame has been received.

Every sentence comes with a formal semantics that characterizes the corresponding behavior of the component in terms of a language over *traces*. The definition of multiple sentences for the assumption and guarantee, respectively, corresponds to the conjunction of their semantics. Because the formal semantics of contract $C = (A, G)$ is defined as $[[C]] := A \Rightarrow G$ (where $\Rightarrow$ denotes the logical implication), we get for a contract that contains sentences $A_1, ..., A_m$ and $G_1, ..., G_n$, respectively, $[[C]] = (A_1 \wedge ... \wedge A_m) \Rightarrow (G_1 \wedge ... \wedge G_n)$. The logical operator $\wedge$ corresponds to the composition of the individual languages, which is defined – together with a comprehensive discussion of the individual specification patterns – in the following chapter.

3 Timing Specification Language

This chapter defines syntax and formal semantics for timing specification patterns defined within the MULTIC project. The definitions employ a notion of *timed traces*, where variables get values in the time domain. Timed traces are based on a notion of sampled (event) signals and thus are suitable to specify and to recognize the meaning of the individual language constructs.

Timing specifications are defined over component interfaces, namely the *ports* of a component. Any behavior in the component model is solely observable at the component ports. Because ports are typed, every behavior observable at a port is restricted to its *value domain* specified by the port type. We denote by Σ_p the value domain of port p. We assume the special value $\perp$ to be member of every value domain, which represents the absence of a value.

Furthermore, timing specifications considered in this report focus on *event*-based communication, where ports have non-absent values only for $t \in T \subset \mathbb{T}$, where T is some discrete set (i.e., is order isomorphic to the natural numbers), and $\mathbb{T} = \mathbb{R}^{\geq 0}$ is our envisioned time domain. This allows us to represent semantics of port behavior in terms of *timed traces*:

Definition 1. *A* timed trace *over port p is defined as an infinite sequence $\omega_p = (t_i, \sigma_i)_{i \in \mathbb{N}}$, where $(t_i)_{i \in \mathbb{N}}$ forms a monotonic sequence of time instances, and $\sigma_i \in \Sigma_p$ are elements from the value domain of p. We require timed traces to be non-zeno, i.e., for each $t \in \mathbb{T}$ exists (t_i, σ_i) such that $t_i \geq t$. We denote by $\Omega_p = \{\omega = (t_i, \sigma_i)_{i \in \mathbb{N}}\}$ the set of timed traces observable at port p.*

For a set P of ports, we define timed traces $(t_i, \vec{\sigma}_i)_{i \in \mathbb{N}}$ over P, where $\vec{\sigma}_i = (\sigma_1, ..., \sigma_n) \in \Sigma_{p_1} \times ... \times \Sigma_{p_n}$. We denote $\Omega_P = \{\omega_P = (t_i, \vec{\sigma}_i)_{i \in \mathbb{N}}\}$.

We define projection *$\omega_P|_q$ of traces over port set P to port $q \in P$, where $\omega_P|_q = (t_i, \sigma_i^q)_{i \in \mathbb{N}}$ if and only if $\omega_P = (t_i, (..., \sigma_i^q, ...)_i)_{i \in \mathbb{N}}$. For any set $L_P \subseteq \Omega_P$, we denote $L_P|_q = \{\omega \in \Omega_q \mid \omega = \omega_P|_q \in L_P|_q\}$ the projection of L_P to q. We extend projection to subsets $P' \subseteq P$ in a canonical way, i.e., we define $\omega_P|_{P'}$ and $L_P|_{P'}$.*

For sets $L_{P_1} \subseteq \Omega_{P_1}$ and $L_{P_2} \subseteq \Omega_{P_2}$, we define their composition *as $L_P = \{\omega \in \Omega_P \mid \omega|_{P_1} \in L_{P_1} \wedge \omega|_{P_2} \in L_{P_2}\}$, where $P = P_1 \cup P_2$.* □

Note that, while the definitions above are sound and sufficient, the languages (in terms of trace sets) we are going to define are not well-suited for composition. This is because proper composition often requires existence of absent values ($\perp$) for particular time points and ports. For example, the trace $(1, (e, \perp)), (3, (\perp, f)), ...$ over two ports requires the existence of trace $(1, e), (3, \perp), ...$ for the one port and $(1, \perp), (3, f), ...$ for the other. Because we do not want to specify languages with absent events explicitly, we introduce a property called *stuttering invariance* for trace sets.

To this end, we define *event projections* for sets $\Sigma'_p \subseteq \Sigma_p$, where events are removed from a trace that do not belong to Σ'_p:

Definition 2. *Let be $\omega = (t_i, \sigma_i)_{i \in \mathbb{N}}$ over event set Σ_p, and let Σ'_p be a subset of Σ_p ($\Sigma'_p \subseteq \Sigma_p$).*

We define event projection $\phi(\omega, \Sigma'_p) = (t'_i, \sigma'_i)_{i \in \mathbb{N}}$ such that there is a monotonic sequence $(j_i)_{i \in \mathbb{N}}$ of natural numbers j_i where for all i holds that $t'_i = t_{j_i}$ and $\sigma'_i = \sigma_{j_i}$, and which is maximal in the sense that for all $j \notin (j_i)_{i \in \mathbb{N}}$ and (t_j, σ_j) holds that $\sigma_j \notin \Sigma'_p$.

We extend projection to sets of traces accordingly: $\phi(L, \Sigma'_p) = \{\phi(\omega, \Sigma'_p) \mid \omega \in L\}$. □

Note that removing events from a trace may lead to a finite sequence in general, which however is not allowed by the event projection operation by definition.

Definition 3. *We say language L_P is stuttering invariant if for all its projections to ports $q \in P$ holds that $\phi(L_P|q, \perp) = \Omega_\perp = \{(t_i, \perp)_{i \in \mathbb{N}}\}$, where $\Omega_\perp$ contains all traces with only absent values.* □

Every language can be transformed into a stuttering invariant language by "adding" respective absent-value events to the traces. Formally, this can be achieved by an inverse event transformation, which we omit here. In the following, we assume the existence of such transformation for the defined languages if needed for composition.

3.1 Basics

In the following sections, we define timing specification languages by a pattern based approach. Specification patterns are natural language like statements that consist of fixed keywords and parameters that are specified by the user. For each pattern, we define its syntax and semantics in terms of the set of traces that satisfy the pattern.

The patterns are defined in terms of BNF grammar. Herein, *parameters* are written in *slanted* font, and **`keywords`** are written in **`typewriter`** font. Sometimes, keywords are hard to recognize, in which cases they are additionally enclosed in quotation marks like in **`'keyword'`**. Optional parts are enclosed in brackets, followed by a question mark, like for example [optional part]?. Parts that may occur zero or more times are enclosed in brackets followed by a star, such as [repeated part]*. Grammar patterns are defined by a name (non-terminal) at the left side, followed by ::, followed by the definition. Alternatives in the definition are separated by | as for this | that.

The fundamental concept for timing specifications are events. Events, as stated above, are solely visible at ports, and are fixed to the corresponding value domains. Though specifications normally are attached to components, hence having a well defined context, ports are specified as follows:

Port :: *PortName* | *ComponentName* **`'.'`** *PortName*

All timing specifications refer to one or more events. The event value observed at a port may or may not be of importance. Event specifications comply to the grammar

EventSpec :: *Port* | *Port* **`'.'`** *EventValue*

The parameter *EventValue* is deliberately left open. It may consist of labels as well as (complex) values. If the event value is omitted then the corresponding *EventSpec* refers to any event that occurs at the specified port.

We introduce the following notion. Given a timed trace $\omega = (t_i, \sigma_i)_{i \in \mathbb{N}}$ and an event $(t_i, \sigma_i) \in \omega$, we say it satisfies the event specification *EventSpec*, denoted $(t_i, \sigma_i) \models$ *EventSpec*, if either *EventSpec* specifies a port and σ_i belongs to the value domain of the port except $\perp$, or *EventSpec* specifies an event value and σ_i is equal to that value.

Timing specifications may refer to event sequences or sets of events:

EventExpr :: *EventSpec* | **`'('`** *EventList* **`')'`** | **`'{'`** *EventList* **`'}'`**
EventList :: *EventSpec* [**`','`** *EventSpec*]*

We extend the notion of satisfaction to event expressions. Given an event sequence $es = (e_1, ..., e_n)$, we say $(t_i, \sigma_i), ..., (t_{i+n-1}, \sigma_{i+n-1}) \in \omega$ satisfies es if every $(t_{i+k-1}, \sigma_{i+k-1})$, $1 \leq k \leq n$, satisfies the event specification e_k. We say $(t_i, \sigma_i), ..., (t_{i+n-1}, \sigma_{i+n-1})$ satisfies the event set $es = \{e_1, ..., e_n\}$ if there is a sequence $(e_{s_1}, ..., e_{s_n})$ such that $\{e_{s_1}, ..., e_{s_n}\} = \{e_1, ..., e_n\}$ which is satisfied.

Time occurs in specifications either as (1) time point or as (2) interval:

TimeExpr :: *Value Unit*
Boundary :: **'['** | **']'**
Interval :: *TimeExpr* | *Boundary Value* **','** *Value Boundary Unit*

Time units may be derived from other basic units. In order to keep the definitions simple, we omit those specifications and stay with the usual time units:

Unit :: **s** | **ms** | **us** | **ns**

For time values, we restrict to simple numbers:

Number :: **0** .. **9** [**0** .. **9**]*
Value :: *Number* | *Number* **'.'** *Number*

3.2 Event Occurrence

For repetitive event occurrences on a particular port, we define a single simplified event pattern:

Repetition :: *EventList* **occurs every** *Interval*$_1$ [**with** *RepetitionOptions*]?**.**
RepetitionOptions :: *Jitter* [**and** *Offset*]? | *Offset* [**and** *Jitter*]?
Jitter :: **jitter** *TimeExpr*
Offset :: **offset** *Interval*$_2$

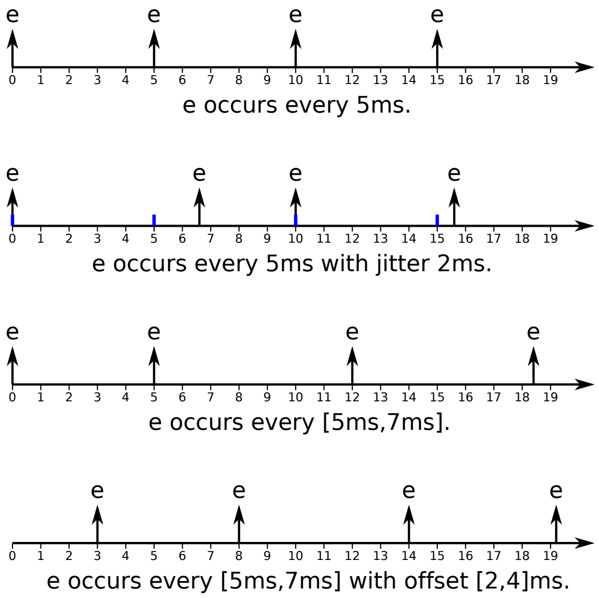
The parameter *Interval*$_1$ defines minimal and maximal time periods between the occurrence of subsequent events. The *jitter* defines an additional (non-deterministic) delay for the occurrence of an event. The optional *offset* defines a delay for the first event occurrence. The offset is set to 0 if omitted.

Definition 4. *Semantics of the repetition pattern "EL occurs every I with jitter J and offset O." is defined as the set of timed traces $(t_i, \vec{\sigma}_i)_{i \in \mathbb{N}}$ such that $\vec{\sigma}_i$ corresponds to the event list EL, $t_i = u_i + j_i \wedge u_0 \in [O^-, O^+] \wedge u_{i+1} - u_i \in I \wedge j_i \in [0, J]$ where $I = (P^-, P^+)$ (where (and) may be closed, i.e. replaced by [and], respectively) is the specified interval, $O = [O^-, O^+]$ is the offset interval, and $J \geq 0$ is the jitter (which is 0 if omitted). We require $0 < P^-$.* □

The pattern complies with the usual meaning of periodic patterns, as well as patterns with minimal and maximal inter-arrival times.

For large jitter (more precisely for $J > P^-$), the language definition is not quite correct because the t_i may not be monotonic ordered anymore. We correct this issue by some *reordering*, which denotes a bijective function $k : \mathbb{N} \to \mathbb{N}$, and where $(t_{k(i)}, \sigma_{k(i)})_{i \in \mathbb{N}}$ defines a trace (so $(t_{k(i)})_{i \in \mathbb{N}}$ forms a monotonic sequence again). Any such trace is part of the semantics of the pattern for which holds $t_{k(i)} = u_i + j_i \wedge u_0 \in [O^-, O^+] \wedge u_{i+1} - u_i \in I \wedge j_i \in [0, J]$.

Figure 3.1 depicts a number of pattern instances with different parameters. The first pattern shows a minimal instance of the pattern with a single-pointed period interval, no jitter and no offset. Because no offset is explicated, the first event occurrence takes places at time point 0. The second pattern add s jitter of maximum 2ms. The blue bars in the timeline show the period intervals as of the first pattern, revealing that the jitter is "added" to the "baseline" periodic behavior. The third pattern is instantiated with a period interval (between 5 and 7ms) as it occurs for example with



e occurs every 5ms.

e occurs every 5ms with jitter 2ms.

e occurs every [5ms,7ms].

e occurs every [5ms,7ms] with offset [2,4]ms.

Figure 3.1: Event Occurrence Pattern Examples

drifting clocks. Because also here no offset is defined, the first event occurs at time point 0. The successor of every event has distance in the interval $[5, 7]$ms of its predecessor. The last pattern shows the application of the offset parameter, which is in the interval $[2, 4]$ms. Hence, the first event occurs somewhere in the interval (3ms in this example). Again, the distances between two successive events is in the interval $[5, 7]$ms.

Sometimes one wants to specify a single event occurrence. The corresponding pattern defines an interval, which is interpreted as relative to the startup of the system:

SingleEvent :: *EventList* **`occurs within`** *Interval* **`.`**

Definition 5. *Semantics of the pattern "EL occurs within I." is the set of timed traces $(t_i, \vec{\sigma}_i)_{i\in\mathbb{N}}$ such that $\vec{\sigma}_i$ corresponds to the event list EL, $t_0 \in I \wedge \sigma_0 \models E \wedge \forall i > 0 : \sigma_i = \bot$.* □

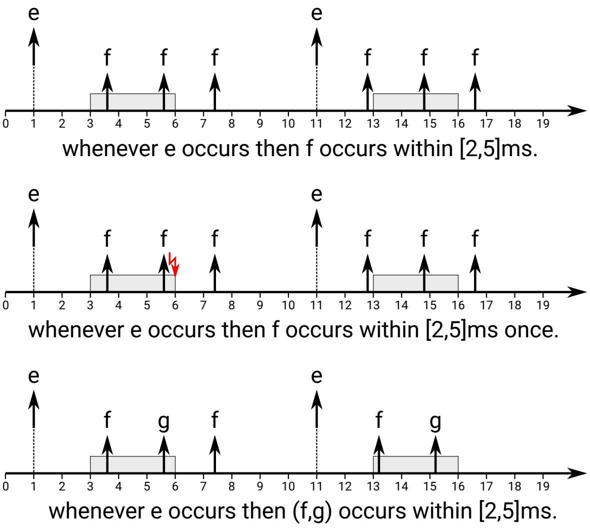
3.3 Reaction Constraints

The reaction pattern provides for forward delay specifications:

Reaction :: **`whenever`** *EventExpr* **`occurs then`** *EventExpr* **`occurs within`** *Interval*
[**`once`**]? **`.`**

The pattern also allows definition of reactions on event sets and event sequences.

Definition 6. *Semantics of the pattern "whenever es_1 occurs then es_2 occurs within I", where es_1 is either a sequence or set that contains k events, and es_2 contains l events, respectively, is defined*



whenever e occurs then f occurs within [2,5]ms.

whenever e occurs then f occurs within [2,5]ms once.

whenever e occurs then (f,g) occurs within [2,5]ms.

Figure 3.2: Reaction Pattern Examples

as the set of timed traces $(t_i, \sigma_i)_{i \in \mathbb{N}}$ *such that* $\forall (t_i, \sigma_i) ... (t_{i+k-1}, \sigma_{i+k-1}) \models es_1 : \exists j \geq i+k : (t_j, \sigma_j) ... (t_{j+l-1}, \sigma_{j+l-1}) \models es_2 \wedge t_{j+l-1} - t_{i+k-1} \in I$. □

Note that the interval recognized by the pattern always starts with the "detection" of the last element of es_1.

The optional **once** keyword forces the pattern to fail if more than one reaction occurs within the specified time window. That is, there is exactly one $j \geq i+k$ such that the corresponding sequence satisfies es_2.

Figure 3.2 depicts some examples of the reaction pattern. The top timeline shows a excerpt of a pattern instance where event f follows event e within an interval of $[2,5]ms$. Observe that the pattern is still satisfied if multiple events f are following an event e. The middle timeline forbids multiple instances of event f for an event e due to the keyword **once**. The bottom timeline shows an instance of the pattern that recognizes an event sequence (f, g) instead of a single event.

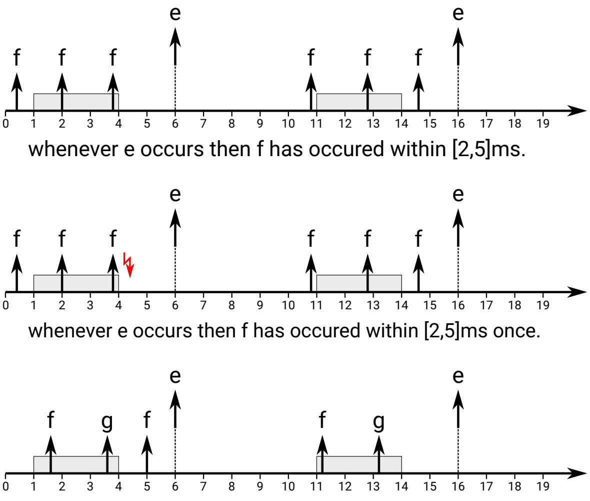
3.4 Age Constraints

The age pattern provides for backward delay specifications:

Age :: **whenever** *EventExpr* **occurs then** *EventExpr* **has occurred within**
Interval [**once**]? **.**

Formal semantics of the age pattern corresponds to the one for the reaction pattern, except that it points backward in time:

Definition 7. *The pattern "whenever* es_1 *occurs then* es_2 *has occurred within* I*", where* es_1 *contains* k *events and* es_2 *contains* l *events, respectively, is defined as the set of timed traces* $(t_i, v_i)_{i \in \mathbb{N}}$ *such that* $\forall (t_i, \sigma_i) ... (t_{i+k-1}, \sigma_{i+k-1}) \models es_1 : \exists j \leq i-l : (t_j, \sigma_j) ... (t_{j+l-1}, \sigma_{j+l-1}) \models es_2 \wedge t_j - t_{i+k-1} \in I$. □



whenever e occurs then f has occured within [2,5]ms.

whenever e occurs then f has occured within [2,5]ms once.

whenever e occurs then (f,g) has occured within [2,5]ms.

Figure 3.3: Age Pattern Examples

Note that the interval recognized by the pattern always starts with the "observation" of the last element of es_1.

As for reaction constraints, the optional **once** keyword forces the pattern to fail if more than one matching events occurs within the specified time window. That is, there is exactly one $j \leq i - l$ such that the corresponding sequence satisfies es_2.

Figure 3.3 depicts some examples of the age pattern. The pattern instances in the figure are symmetric to those of Figure 3.2.

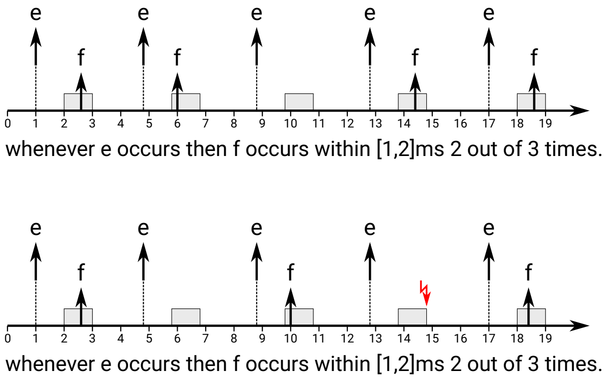
3.5 Restricting Over- and Undersampling

Reaction and age pattern are often used in conjunction with over- or undersampling scenarios. If for example a component receives more input values than it produces output values (undersampling), then an age pattern may be used to characterize such behavior (indeed w/o **once** extension). This kind of specification however does not allow for restricting the number of under- or oversampling situations, which can be achieved with the *restricted* flavor of the reaction and age patterns:

Reaction :: **whenever** *EventExpr* **occurs then** *EventExpr* **occurs within** *Interval*
[**once**]? [*Number* **out of** *Number* **times**]? **.**

Age :: **whenever** *EventExpr* **occurs then** *EventExpr* **has occurred within**
Interval [**once**]? [*Number* **out of** *Number* **times**]? **.**

The part *Number* **out of** *Number* **times** specifies that the condition defined earlier for the reaction (age) pattern may be violated for a particular fraction of occurrences. For example, the reaction pattern “whenever e occurs then f occurs within $[10, 12]ms$ 3 out of 5 times” specifies that event f must follow e in the time window $[10, 12]ms$ at least 3 times for every 5 successive occurrences of event e. The restriction k **out of** n **times** is interpreted as a *sliding window*, which must hold



whenever e occurs then f occurs within [1,2]ms 2 out of 3 times.

whenever e occurs then f occurs within [1,2]ms 2 out of 3 times.

Figure 3.4: Reaction Pattern with Undersampling

for any sequence of n matching occurrences of event expression es_1 at least k times. The formal definition is omitted here, as it can be derived from the specification of the reaction (age) pattern above. It basically modifies Definition 6 (Definition 7) in that a sequence of n event patterns es_1 are assumed, for which at least k matching event patterns es_2 must exist. Note, however, that these k event patterns are not needed to differ from each other.

Figure 3.4 depicts some examples of how the restriction works. The top timeline shows an excerpt of a trace that is satisfied by the restriction 2 **out of** 3 **times** for a reaction pattern instance. Although the third occurence of event e is not followed by corresponding event f, it does not lead to no violation of the pattern because at least 2 of any 3 successive occurrences are still satisfied. The situation at the bottom timeline is different. Here, the pattern is violated because the second, third and fourth occurrence of event e is not followed by an event f, and thus only one out of three occurrences are satisfied.

3.6 Causal Event Relations

It is a well-known problem that timing specifications that are based on event observations have only limited expressiveness when it comes to specifying functional relations between events. Several approaches exist to mitigate the issue, such as [6, 8]. These approaches can be applied in rather deterministic scenarios.Cases with complex functional relations, over- and under-sampling situations, or where event ordering is non-deterministic, e.g., due to variable execution times in parallel execution (multi-core) platforms call for more expressive approaches. The underlying problem can be shown by a simple example depicted at the top timeline in Figure 3.5. It shows an excerpt of a trace that is "observed" by a reaction pattern between input events e and output events f. The pattern is satisfied as it matches the first (second) occurrence of event e with the first (second) occurrence of event f. However, because of some reason (like a particular kind of event buffering or the parallel execution of internal functions), the the first event f is in fact *caused* by the second event e. The relations are indicated by dashed blue lines. Note that no non-causal specification is able to cover this aspect. The reason for this is the fact that the specification does not consider functional dependencies. If, for example, f is a function of e, i.e. $f = f(e)$, the problem would disappear if this dependency is observable. This is shown at the bottom timeline of Figure 3.5, where we assume $f_1 = f(e_1)$ and $f_2 = f(e_2)$, which allows us to exploit the (functional) input/output relations. The MTSL allows the definition of basic functional relations by expressing event values. However, more complex functional

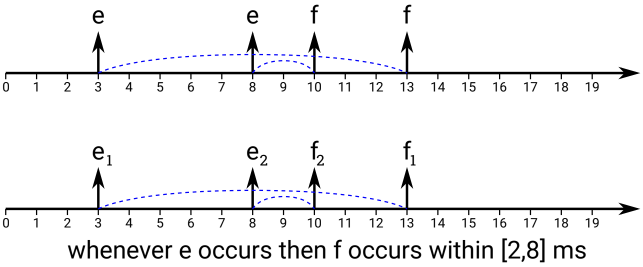


Figure 3.5: Event Causality Example

relations are not (yet) supported.

The authors of [10] propose a different approach, where events can be distinguished by coloring. A problem occurs with this approach when more than two events are related. For example, a reaction pattern that states *whenever (a,b) occurs ...* would require that the coloring of the two input events *a* and *b* are consistent, i.e., both events must have the same color in order to be recognized as associated events.

Another approach proposed in this report goes in the same direction, but avoids such issues by the definition of a more general concept called *causal event relation*. The advantage of this approach is that it enables the definition of (arbitrary) complex causal dependencies. This is achieved by strictly distinguishing between the definition of the actual causal relations and their realization. On the other hand, increasing complexity of event relations increases the difficulty to make those relations observable. In our case, observability can be reduced to the question of how events can be uniquely identified. As said above, it could be established by assuming that related events can be colored in a way that enables identification of the actual relation. Another way is the introduction of time stamps. Annotating events with the time point at which they are created allows to distinguish events from each other. We will provide two simple mechanisms allowing identification of event relations.

The formal definition of *causal event relations* is as follows:

Definition 8 (Causal Event Relation)**.** *Let p_1 and p_2 be ports, and let Ω_{p_1,p_2} be the semantics of p_1 and p_2. A* causal event relation *over p_1 and p_2 is a function*

$$\triangleright(p_1, p_2) : (\mathbb{T} \times \Sigma_{p_1}) \to 2^{\mathbb{T} \times \Sigma_{p_2}}$$

where for all $\omega \in \Omega_{p_1,p_2}$ and for all event occurrences $(t_i, \sigma_i) \in \omega|_{p_1}$ exist $(t_j, \sigma_j), ..., (t_k, \sigma_k) \in \omega|_{p_2}$ such that it holds $\triangleright(p_1, p_2)((t_i, \sigma_i)) = \{(t_j, \sigma_j), ..., (t_k, \sigma_k)\}$ and $t_i \leq t_j, ..., t_k$.

We also define a backward *causal event relation as a function*

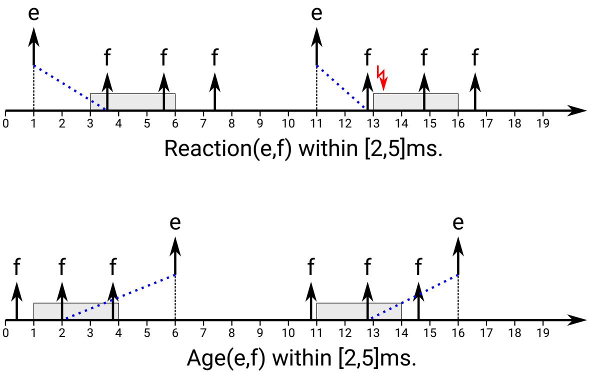
$$\triangleleft(p_1, p_2) : (\mathbb{T} \times \Sigma_{p_2}) \to 2^{\mathbb{T} \times \Sigma_{p_1}}$$

where for all $\omega \in \Omega_{p_1,p_2}$ and for all event occurrences $(t_i, \sigma_i) \in \omega|_{p_2}$ exist $(t_j, \sigma_j), ..., (t_k, \sigma_k) \in \omega|_{p_1}$ such that it holds $\triangleleft(p_1, p_2)((t_i, \sigma_i)) = \{(t_j, \sigma_j), ..., (t_k, \sigma_k)\}$ and $t_j, ..., t_k \leq t_i$. □

Causal event relations are transitive. Given three ports p_1, p_2, p_3, and causal event relations $\triangleright(p_1, p_2)$ and $\triangleright(p_2, p_3)$, then $\triangleright(p_1, p_3)$ is given by:

$$(t_j, \sigma_j) \in \triangleright(p_1, p_2)((t_i, \sigma_i)) \wedge (t_k, \sigma_k) \in \triangleright(p_2, p_3)((t_j, \sigma_j)) \Rightarrow (t_k, \sigma_k) \in \triangleright(p_1, p_3)((t_i, \sigma_i))$$

This property (though not surprisingly) gives means to the intuition of "additive" latencies, where we say that response times Xms and Yms sum up to $X + Yms$.



Reaction(e,f) within [2,5]ms.

Age(e,f) within [2,5]ms.

Figure 3.6: Causal Pattern Examples

Causal Reaction and Age Based on those causal event relations, we define a causal version of the reaction pattern:

CausalReaction :: **`Reaction(`***EventSpec* `','` *EventSpec***`)`** **`within`** *Interval* **.**

Definition 9. *Semantics of the pattern “***`Reaction(`**e_1, e_2**`) in`** I*.”, where* e_1 *refers to port* p_1*, and* e_2 *refers to port* p_2*, respectively, is defined as the set of timed traces* $\omega \in \Omega_{p_1,p_2}$ *where for all* $(t_i, \sigma_i)_{i\in\mathbb{N}} \in \omega|_{p_1}$*,* $(u_i, \rho_i)_{i\in\mathbb{N}} \in \omega|_{p_2}$*, and for all event occurrences* $(t_i, \sigma_i) \in (t_i, \sigma_i)_{i\in\mathbb{N}}$ *such that* $\sigma_i \models e_1$*, holds* $\triangleright(p_1, p_2)((t_i, \sigma_i)) \neq \emptyset$ *and* $\big((u_j, \rho_j) \in \triangleright(p_1, p_2)((t_i, \sigma_i)) \wedge \rho_j \models e_2\big) \Rightarrow u_j - t_i \in I$. □

Also a causal age pattern is defined:

CausalAge :: **`Age(`***EventSpec* `','` *EventSpec***`)`** **`within`** *Interval* **.**

Definition 10. *Semantics of the pattern “***`Age(`**e_1, e_2**`) in`** I*.” is defined as the set of timed traces* $\omega \in \Omega_{p_1,p_2}$*, where for all* $(t_i, \sigma_i)_{i\in\mathbb{N}} \in \omega|_{p_1}$*,* $(u_i, \rho_i)_{i\in\mathbb{N}} \in \omega|_{p_2}$*, and for all event occurrences* $(u_i, \rho_i) \in (u_i, \rho_i)_{i\in\mathbb{N}}$ *such that* $\rho_i \models e_2$*, holds* $\triangleleft(p_1, p_2)((u_i, \rho_i)) \neq \emptyset$ *and* $\big((t_j, \sigma_j) \in \triangleleft(p_1, p_2)((u_i, \rho_i)) \wedge \sigma_j \models e_1\big) \Rightarrow u_i - t_j \in I$. □

Figure 3.6 depicts examples of the two patterns. The top timeline shows an example of the reaction pattern. The blue lines indicate those events that are related by the definition of some causal event relation. It is because of the relation that the pattern is violated with the second occurrence of the event e. This is in contrast to the non-causal version of the pattern (cf. Figure 3.2). The bottom timeline shows the application of the causal age pattern. The blue lines indicate the related events again.

Causal Event Relation Functions The causal patterns require the existence of corresponding event relations. That is, every reference to a function $\triangleright$ and $\triangleleft$ is assumed to be specified in the respective contract. This calls for a specification language for such relations. We add a specification pattern that allows definition of such causal event relations:

CausalFuncDecl :: *CausalFuncName* **(** *Port* `','` *Port* **)** `':='` *CausalRelation*
CausalFuncName :: `'|>'` | `'<|'`

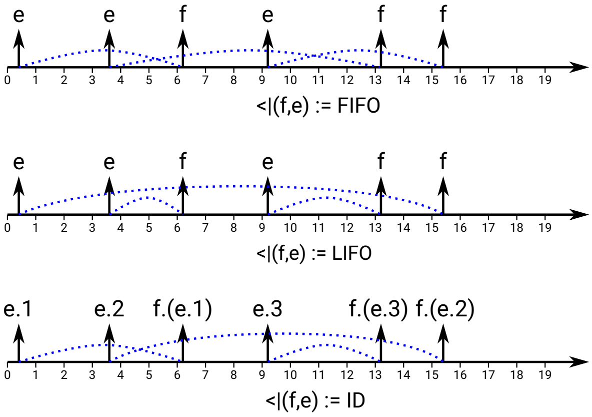


Figure 3.7: Causal Function Declaration Examples

We support two mechanisms in order to define causal event relations. The first mechanism supports the approach taken in [8], where a number of relevant relations are pre-defined. Because however, we define causal relations locally, only two functions remain, namely FIFO (first-in-first-out) and LIFO (last-in-first-out). So we define two *built-in* relations:

CausalRelation :: **FIFO** | **LIFO**

The semantics of these functions is simple. Events that occur at the input port are sent to the output port in FIFO (LIFO) order. Figure 3.7 depicts examples of how these functions are working. The blue lines indicate the events that are related due to the corresponding function.

The second supported mechanism relates to the approach taken in [10] and assumes that events are identifiable, either by time stamping or some other mechanism. To capture this, we define for port p the function $id_p : \mathbb{T} \times \Sigma_p \to \mathbb{N}$, which assigns an identifier to every event occurrence at that port. We further define *id-transducers*, which are functions $tid_{(p1,p2)} : (\mathbb{T} \times \Sigma_{p1}) \times (\mathbb{T} \times \Sigma_{p2}) \to \mathbb{N}$ according to the definition of causal event relations $\triangleright$ and $\triangleleft$, respectively. Based on these id_p functions and id-transducers, we define a corresponding causal event relation:

CausalRelation :: **ID**

It states that causally related events on the involved ports have identical id values. In other word, $(t_j, \sigma_j) \in \triangleright(p_1, p_2)((t_i, \sigma_i))$ implies $tid_{p_1,p_2}((t_i, \sigma_i), (t_j, \sigma_j)) = id_{p_1}((t_i, \sigma_i))$, and for $\triangleleft$ respectively.

Note that the definition does not impose any ordering of the output/input events with respect to the related output/input events as the function **FIFO** and **LIFO** do. Instead, it defines that those events are related, which have the specified relation of their ids, and more precisely, have the same id with respect to function tid.

The bottom timeline of Figure 3.7 shows an example of the **ID** function. The ids of occurrences of event e are depicted by a dot followed by the value. The tid values of the events f are indicated by the id of the corresponding e event, enclosed in parentheses.

3.7 Local Clocks

While timing specifications adhere to some globally defined time domain by default, there are applications where specifications based on local clocks are desired. An example is runtime observation of timing specifications, where local hardware clocks are used for the generation of time stamps. The timing specification language supports defining and referring to local clocks in timing specifications.

Local clocks are defined by a name, such as in **`Clock`** *Name*, and a set of optional clock properties. Timing specifications can refer to a local clock by the (optional) **`using clock`** *Name* construct. This construct says that all timing parameters of the specification refer to the specified clock. For example, a periodic occurrence of event *E* using (local) clock *C* instead of global time is specified as follows:

E **`occurs every`** *10 ms* **`using clock`** *C* **.**

All specification patterns support this construct. Local clock definitions have the following format:

ClockDefinition :: **`Clock`** *Name* **`has`** [**`resolution`** *TimeExpr*]? [**`skew`** *TimeExpr*]?
[**`drift`** *Interval*]? [**`maxdiff`** *TimeExpr*]? **.**

The optional parts of a clock definition allow specifying some practically relevant properties, such as resolution and drift. At least one property must be specified. The properties are defined as follows:

`resolution`	Resolution	Specifies that the clock advances in discrete steps of length **`TimeExpr`**.
`skew`	Skew	Similar to a jitter, skew adds an uncertainty interval of size **`TimeExpr`** to the clock. Requires presence of property **`resolution`**, and it must hold **`skew`** $<$ **`resolution`**. Cannot be used in conjunction with the **`maxdiff`** property.
`drift`	Drift	Specifies slow down/speed up factors of the clock. Cannot be used in conjunction with the **`maxdiff`** property.
`maxdiff`	Max. difference	Adds an uncertainty interval of size $2*$**`TimeExpr`** to the clock value. Cannot be used in conjunction with the **`drift`** and **`skew`** properties.

Semantically, there is no difference between specifications based on the global time domain and those based on a local clock. Every local clock, say $clock_k$, induces a corresponding time domain $\mathbb{T}_k$. Given that, semantics of a timing specification using $clock_k$ is defined by replacing time domain $\mathbb{T}$ by $\mathbb{T}_k$ in Definition 1. [1]

However, there are (at least) two important aspects to be considered when dealing with local clocks. Firstly, while different specifications may refer the same events, they may observe different time stamps for the occurrences of those events. Suppose the following specifications:

```
(C1) A: Clock clock1 has resolution 1ms.
        Event E occurs every 10 ms using clock1.
     G: Whenever E occurs then F occurs within 6ms using clock1.
(C2) A: Clock clock2 has resolution 10us.
        Event E occurs every 10 ms using clock2.
     G: ...
```

[1] This also holds for clocks with finite resolution, where the time domain is isomorphic to the natural numbers $\mathbb{N}$, because traces are non-zeno.

Specification (C1) may not be able to detect jitters in the timing that are below the resolution of **clock1**, whereas the resolution of **clock2** is much higher. Hence, it may happen that (C1) is satisfied for some observation while (C2) is violated. Depending on the specified properties of the individual clocks, the interpretation of verification results may become a challenging task.

In order to support such interpretation, it might be desirable to know for specifications involving local clocks corresponding specifications using a global clock. For example, for the specification

```
(C1) A: Clock clock1 has resolution 1ms.
        Event E occurs every 10 ms using clock1.
```

one might be interested to know for the corresponding specification based on the global time domain

```
(C1') A: Event E occurs every [L,U] ms with jitter J.
```

the exact values of **L,U** and **J**. There may be also application scenarios where the other direction is of interest: Given a specification based on the global time domain, what are corresponding specifications based on local clocks? Unfortunately, these are also challenging tasks in general, and even impossible for many combinations of clock properties. We do, however, in the longer run plan to support computing weakest pre-specifications in the mutually other clock domain.

The second important aspect concerns the composition of specifications that talk about different clocks. The operation of composition in Definition 1 requires a common time domain.

A key element for both aspects is the relation between the actual values of clocks. In order to reason about the correlation between local and global specifications, we need to know for any time stamp t^k of some local clock $\mathbb{T}_k$ the corresponding time stamp t of the global clock $\mathbb{T}$ (and vice versa). For composition, we use such correlation in order to *cast* traces into a common time domain. As specifications are defined over (infinite) sequences of event occurrences, we also reason about the relation between sequences $(t_i^k)_{i\in\mathbb{N}}$ and $(t_i)_{i\in\mathbb{N}}$, respectively, of time stamps.

Definition 11. *Let $\Omega(\mathbb{T}_k)$ be the set of all possible sequences of time stamps for clock $\mathbb{T}_k$, and $\Omega(\mathbb{T})$ set set of all such sequences of clock $\mathbb{T}$.*

The characteristic behaviour of clock $\mathbb{T}_k$ induces a relation between the two sets, which is expressed as $tr_k \subseteq \Omega(\mathbb{T}_k) \times \Omega(\mathbb{T})$. For every sequence $\pi^k = (t_i^k)_{i\in\mathbb{N}} \in \Omega(\mathbb{T}_k)$, the relation provides all corresponding sequences $\pi = (t_i)_{i\in\mathbb{N}} \in \Omega(\mathbb{T})$ (defined by $tr_k(\pi^k) := \{\pi \mid (\pi^k, \pi) \in tr_k\}$). Also the reverse holds: for every $\pi \in \Omega(\mathbb{T})$, tr_k provides all possible corresponding sequences $\pi^k \in \Omega(\mathbb{T}_k)$ of clock $\mathbb{T}_k$. □

For the clock properties specified above, the relation tr_k is always well-defined. For example, the relation for a clock with resolution res_k is defined by $\{(\pi^k, \pi) \mid \pi^k = (t_i^k)_{i\in\mathbb{N}} \wedge \pi = (t_i)_{i\in\mathbb{N}} \wedge \forall i \in \mathbb{N} : t_i^k = \lfloor \frac{t_i}{res_k} \rfloor \cdot res_k\}$. Table 3.1 provides the relations for all allowed combinations of properties.

While these relations are sufficient to perform trace composition, they provide only little support for interpreting specifications that are based on local clocks. The issue that prevents us from expressing correlating specifications is visualized in Figure 3.8. The bottom part shows the time line according to the global time domain $\mathbb{T}$. The top part shows the timeline of some local clock $\mathbb{T}_k$. The green bars indicate areas where some specification is satisfied, as for example some response time interval. Suppose that for the specification based on local clock $\mathbb{T}_k$, this interval is defined by $[A, B]$. In order to derive a corresponding interval for a specification based on global time, we have to investigate for every $\pi^k \in \Omega(\mathbb{T}_k)$ all corresponding $\pi \in \Omega(\mathbb{T})$ such that $(\pi^k, \pi) \in tr_k$. For every π^k and every time stamps $t_i^k \in [A, B]$ from π^k, we collect all corresponding time stamps t_i from any π obtained from

[2] french for: right continuous, left limits.

[3] where $tr_r \circ tr_d := \{(\pi^r, \pi) \mid \exists (\pi^r, \pi^d) \in tr_r \wedge \exists (\pi^d, \pi) \in tr_d\}$

[4] There are clock synchronization protocols that reset clocks. We, however, do not support those protocols because of the mathematical anomalies of the involved clock relations.

Properties	tr_k
resolution=res_k	$t_i^k = \lfloor \frac{t_i}{res_k} \rfloor \cdot res_k$
drift=d_k	$\exists d : \mathbb{T} \to d_k$, d is cadlag[2] in $\mathbb{T}$, $t_i^k = \int_0^{t_i} d(t)dt$

drift=d_k
resolution=res_k
	Clock $\mathbb{T}_d$ for d_k with relation tr_d and clock $\mathbb{T}_r$ for res_k with relation tr_r. $tr_k = tr_r \circ tr_d$.[3]

skew=s_k
resolution=res_k
	$\exists s : \mathbb{N} \to [0, s_k]$, $s(0) = 0$, $f_s : \mathbb{T} \to \mathbb{T}_k$ s.t. $\forall i \in \mathbb{N} : t \in [i \cdot res_k + s(i), (i+1) \cdot res_k + s(i+1)) \Leftrightarrow f_s(t) = i \cdot res_k$

drift=d_k
skew=s_k
resolution=res_k
	Clock $\mathbb{T}_d$ for d_k with relation tr_d and clock $\mathbb{T}_s$ for s_k, res_k with relation tr_s. $tr_k = tr_s \circ tr_d$.

maxdiff=d_k	$\exists d : \mathbb{T} \to [-d_k, d_k]$, d is differentiable in $\mathbb{T}$ except in a countable set $T \subset \mathbb{T}$, d is continuous[4], $t_i^k = t_i + d(t_i)$

maxdiff=d_k
resolution=res_k
	Clock $\mathbb{T}_d$ for d_k with relation tr_d, and clock $\mathbb{T}_r$ for res_k with relation tr_r. $tr_k = tr_r \circ tr_d$.

Table 3.1: Clock Properties and Corresponding Clock Relations

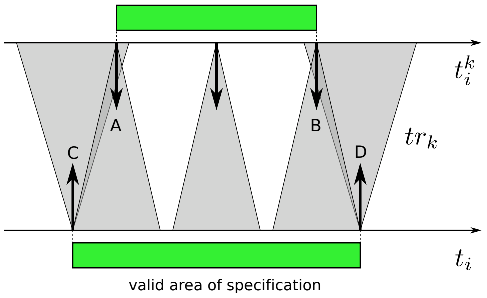


Figure 3.8: Non-determinism prevents Mapping of Specifications with different Clock Domains

tr_k. With this, we can determine an interval in which all such t_i are contained, in this case $[C, D]$. However, in general not only the number of sequences in tr_k for a given sequence π^k is larger than 1 (which is indicated by the triangle shapes in Figure 3.8) but this holds also for the other direction. In other words, we have in general $|tr_k(\pi^k)| > 1$ and $|tr_k(\pi)| > 1$. Hence, if we perform a reverse mapping, starting from the interval $[C, D]$, we will typically *not* end up in the corresponding interval $[A, B]$.

A sufficient condition for obtaining equivalence between local and global specifications is that tr_k can be either defined as a function $tr_k : \Omega(\mathbb{T}_k) \to \Omega(\mathbb{T})$, or as a function $tr_k : \Omega(\mathbb{T}) \to \Omega(\mathbb{T}_k)$. For example, this is the case for clocks with finite resolution and no other properties. However, it is always possible to express implied specifications in any direction. An investigation of this property would be subject to further work.

Concerning the application of local clocks, clock definitions should occur in the assumption of a specification, because clocks are typically not controlled by the respective component. Furthermore, although this is basically outside the definition of the specification language, clock definitions break the locality rule of contract-based design in order to ease specification. The locality rule says that specifications must reason solely about the interfaces of the respective component. A defined clock however should be visible also at subcomponents of a component, such as for all functions that are

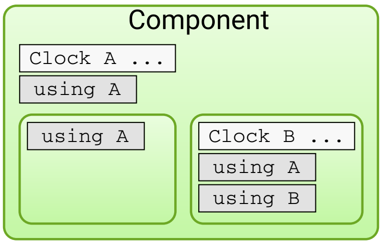


Figure 3.9: Definition of Clock Domains

allocated to the same ECU. Hence, we assume that clock definitions are *visible* at all subcomponents of a component as well, i.e. they can be referred by a **using clock** construct (cf. Figure 3.9).

3.8 BNF

The following grammar summarizes the specification patterns defined in the former sections.

```
TimeSpec            ::  SingleEvent | Repetition | Reaction | Age
                        | CausalReaction | CausalAge | CausalFuncDecl
                        | ClockDefinition
SingleEvent         ::  EventList occurs within Interval
                        [ using clock Name ]? .
Repetition          ::  EventList occurs every Interval [ with RepetitionOptions  ]?
                        [ using clock Name ]? .
RepetitionOptions   ::  Jitter [ and Offset ]? | Offset [ and Jitter ]?
Jitter              ::  jitter TimeExpr
Offset              ::  offset Interval
Reaction            ::  whenever EventExpr occurs then EventExpr occurs within Interval
                        [ once ]? [ Number out of Number times ]?
                        [ using clock Name ]? .
Age                 ::  whenever EventExpr occurs then EventExpr has occurred within
                        Interval [ once ]? [ Number out of Number times ]?
                        [ using clock Name ]? .
CausalReaction      ::  Reaction(EventSpec ',' EventSpec) within Interval
                        [ using clock Name ]? .
CausalAge           ::  Age(EventSpec ',' EventSpec) within Interval
                        [ using clock Name ]? .
EventExpr           ::  EventSpec | '(' EventList ')' | '{' EventList '}'
EventList           ::  EventSpec [ ',' EventSpec ]*
EventSpec           ::  Port | Port '.' EventValue
Port                ::  PortName | ComponentName '.' PortName
Interval            ::  TimeExpr | Boundary Value ',' Value Boundary Unit
TimeExpr            ::  Value Unit
Boundary            ::  '[' | ']'
Value               ::  Number | Number '.' Number
Number              ::  0 .. 9 [ 0 .. 9  ]*
```

```
Unit              ::  s | ms | us | ns
CausalFuncDecl    ::  CausalFuncName ( Port ',' Port ) := CausalRelation
CausalFuncName    ::  '|>' | '<|'
CausalRelation    ::  FIFO | LIFO | ID
ClockDefinition   ::  Clock Name has [ resolution TimeExpr ]?
                      [ skew TimeExpr ]? [ drift Interval ]?
                      [ maxdiff TimeExpr ]? .
```

3.9 Changes

During the MULTIC-Tooling project, some changes have been made to the MTSL w.r.t [3] in order to eliminate issues that may occur when using the specification in practice. This section collects and discusses these changes.

Event Specifications In the present version, the *Event* parameter in a number of specification patterns has been changed to *EventSpec* in order to avoid confusion with the terms event and event occurrence.

Event Lists The repetition pattern and the single event pattern have been extended to support sending events on multiple ports simultaneously. That is, in both patterns *EventSpec* has been replaced by *EventList*.

Event Occurrences In [3], the repetition pattern has been defined as

```
Repetition  ::  EventSpec occurs every Interval [ with jitter TimeExpr ]?
```

whereas the actual pattern allows the definition of an additional offset. The former definition does not allow for specifying time instances for the initial event occurrence. The change comes with a redefinition of the underlying semantics. In the former version, the first event occurrence was bound to the interval $[0, P^+] + [0, J]$, where P^+ is the upper bound of the period interval and J denotes the maximum jitter. In the actual version, the semantics bounds the first event to the interval $[O^-, O^+] + [0, J]$ where $[O^-, O^+]$ specifies the (explicitly defined) offset interval. Note that, if no offset is specified, it is implicitly set to 0, resulting in the first event occurrence to be bound to $[0, J]$.

The revised version allows to "emulate" the former semantics. In order to obtain, for example, the same event language as in the pattern "*e* **occurs every** *[5,7] ms* **with jitter** *2 ms*" according to the semantics in [3], one specifies "*e* **occurs every** *[5,7] ms* **with jitter** *2 ms* **and offset** *[0,7]ms*".

Causal Function Declarations Patterns have been added allowing for defining the causal relation function on which the *CausalReaction* and *CausalAge*, respectively, rely. This was missing in [3].

At the same time, causal relations have been removed from the definition of *EventExpr*, as there is a high potential of accidental misuse leading to semantic issues. As on the other hand *CausalReaction* and *CausalAge* do not (yet) support event sets or event sequences, this change represents a restriction of expressiveness.

4 Automata – Semantic Foundation

Whereas the MULTIC timing specification language defined in Chapter 3 employs a declarative specification style and semantically notion of timed traces as a suitable tool to specify languages, the definition of generators and monitors that adhere to such specifications calls for an operational semantics. This chapter defines syntax and semantics of a class of automata that are suitable to *implement* MULTIC timing specifications. It starts with the necessary definition of the semantic domain and its observables in terms of variables. Section 4.2 defines the automaton class. It foreshadows future developments in that its expressiveness is generally not required for the class of specifications considered here. It represents an instance of the well-known class of *hybrid automata* and thus leaves room for later extensions.

Section 4.3 provides a formal notion of semantics for this automaton class in terms of transition systems, for which in turn *trajectories* can be defined in order to explain automaton executions. Trajectories are finally related to timed traces, which form the basic notion on which semantics of MULTIC timing specifications is defined.

It is important to note that the definitions in this chapter remain purely mathematical, as they have to provide the link between the operational semantics and the languages of the specification patterns defined in Chapter 3. To keep however the notion as strong as possible to their original definitions, we deliberately omit the concept of ports, and come back to this in Chapter 6. The chapter does not provide examples in order to keep the presentation short. We refer the reader to [7] for a more comprehensive discussion. Chapter 5 presents an implementation-oriented notion for such automata.

4.1 Variables, Predicates and Assignments

We assume a set $\mathcal{V}$ of variables. Each variable $v \in \mathcal{V}$ carries values from a particular domain D_v. Primitive domains are $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$, namely the set of natural numbers (including 0), the integer numbers and the real numbers, respectively. Because we will use variables for event-based communication too, variables may carry values only at particular points in time, and otherwise are "absent". We thus introduce the special value $\bot$ and assume $\bot \in D_v$ for such variable domains. For a set $V \subseteq \mathcal{V}$ we denote $D_V = \bigcup_{v \in V} D_v$ the domain of V.

A valuation of v is an assignment of a value from D_v. For a set $V \subseteq \mathcal{V}$, we define a *valuation* as a type safe function $\sigma : V \to D_V$, which assigns each variable $v \in V$ a value in D_v. We denote Σ_V the set of all valuations over V. We extend valuations to functions whose free variables are from V in a canonical way: For function F, we denote $\sigma(F)$ the valuation of F with respect to the valuation of the free variables of F. For variable set $V' \subseteq V$, we define the projection of σ to the variables in V', denoted $\sigma|_{V'} \in V'$, such that $\sigma|_{V'}(v) = \sigma(v)$ for each $v \in V'$. For two variable sets V_1 and V_2, we define $\sigma := \sigma_1 \circlearrowleft \sigma_2$ such that $\sigma|_{V_1 \setminus V_2} = \sigma_1|_{V_1 \setminus V_2}$ and $\sigma|_{V_2} = \sigma_2$.

A predicate ψ over variable set V is defined by the grammar

$$\psi ::= \mathit{true} \mid c \bowtie F \mid v \bowtie F \mid \neg\psi_1 \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2$$

where $v \in V$, $c \in D_V$, $\bowtie \in \{=, <, >, \leq, \geq\}$, and F is an expression whose free variables are from V. We denote Ψ_V the set of predicates over V. Semantics of predicates is standard with respect to a valuation $\sigma \in \Sigma_V$:

$$\begin{aligned}
(\mathit{true})[\sigma] &:= \mathit{true} \\
(c \bowtie F)[\sigma] &:= c \bowtie \sigma(F) \\
(v \bowtie F)[\sigma] &:= \sigma(v) \bowtie \sigma(F) \\
(\neg\psi_1)[\sigma] &:= \neg(\psi_1)[\sigma] \\
(\psi_1 \wedge \psi_2)[\sigma] &:= (\psi_1)[\sigma] \wedge (\psi_2)[\sigma] \\
(\psi_1 \vee \psi_2)[\sigma] &:= (\psi_1)[\sigma] \vee (\psi_2)[\sigma]
\end{aligned}$$

The definition of $(v_1 \bowtie v_2)[\sigma]$ is somehow sloppy, as it may be undefined, depending on the relations available in the respective value domains, and due to type inconsistencies. We assume some kind of well-formedness of predicates ensuring that $\sigma(v_1) \bowtie \sigma(v_2)$ is always well defined.

An assignment ξ over variable sets V and V' is a function $\xi : \Sigma_V \to \Sigma_{V'}$. We denote $\Xi_{V,V'}$ the set of assignments over V, V'. We again assume some kind of well-formedness ensuring that assignment is always well defined.

4.2 Automata

In order to be as unconstrained as possible, we define a class of automata that is consistent with the definition of hybrid automata proposed by Henzinger [7]. We made two deviations from the original definition. Firstly, we consider direction of data flow, i.e., we distinguish input from output variables. Secondly, we distinguish continuously evolving variables from those variables that realize event flows. In fact, the original definition introduces a separate set of events on this behalf.

Definition 12. *An automaton is a tuple* $M = (V, L, T, l_0, \sigma_0, inv, flow, guard, act, set)$ *where*

- $V = V_I \uplus V_P \uplus V_O \subseteq \mathcal{V}$ *is a set of disjoint input, private, and output variables. We further define* $V_I = V_I^c \uplus V_I^e$, *and* $V_O = V_O^c \uplus V_O^e$ *in order to distinguish continuously evolving input/output variables (*V_I^c *and* V_O^c*) from event variables (*V_I^e *and* V_O^e*). For convenience, we also define* $V^c = V_I^c \cup V_P \cup V_O^c$, *and* $V^e = V_I^e \cup V_O^e$

- L *is a set of locations,*

- $T \subseteq L \times L$ *is a set of transitions,*

- $l_0 \in L$ *is the initial location,*

- $\sigma_0 \in \Sigma_{V_P \cup V_O}$ *is the initial valuation of the private and output variables,*

- $inv : L \to \Psi_{V^c}$ *is a labeling function that assigns an invariant predicate to every location of the automaton,*

- $flow : L \to \Psi_{V^c \cup \dot{V}^c}$ *is a labeling function that assigns a flow predicate to every location of the automaton.* $\dot{V}^c$ *denotes the set of dotted variables in* V^c, *which represent first derivatives of those variables.*

- $guard : T \to \Sigma_V$ *is a labeling function that assigns a guard predicate to every transition of the automaton,*

- $act : T \to \Sigma_{V_O^e}$ *is a labeling function that assigns an output action to every transition of the automaton,*

- *$set : T \to \Xi_{V_I \cup V_P, V_P \cup V_O}$ is a labeling function that assigns a variable assignment to every transition of the automaton.*

□

4.3 Semantics

Semantics of hybrid automata is defined in terms of a labeled transition systems:

Definition 13. *Given automaton $M = (V, L, T, l_0, \sigma_0, inv, flow, guard, act, set)$, its labeled transition system is $S_M = (Q, Q_0, A, \to)$ where*

- *$Q \subseteq L \times \Sigma_V$ is a set of states and $Q_0 \subseteq Q$ is a subset of initial states, such that $(l, \sigma) \in Q$ iff the predicate $inv(l)[\sigma]$ is true, and $(l, \sigma) \in Q_0$ iff $l = l_0$ and $\sigma|_{V_P \cup V_O} = \sigma_0$.*
- *$A = \Sigma_{V^e} \cup \mathbb{R}^{\geq 0}$ is a label set,*
- *$\to \subseteq Q \times A \times Q$ is a labeled transition relation, where*
  - *For each $e \in \Sigma_{V^e}$, define $(l, \sigma) \xrightarrow{e} (l', \sigma')$ iff there is a transition $t = (l, l') \in T$ such that $guard(t)[\sigma \circlearrowleft e|_{V_I^e}]$ is true, $act(t) = e|_{V_O^e}$, and $\sigma'|_{V_P \cup V_O} = set(\sigma|_{V_I \cup V_P} \circlearrowleft e|_{V_I^e})$.*
  - *For each $\delta \in \mathbb{R}^{\geq 0}$, define define $(l, \sigma) \xrightarrow{\delta} (l', \sigma')$ iff $l = l'$, and there is a differentiable function $f : [0, \delta] \to \Sigma_V$ such that (1) $f(0) = \sigma$ and $f(\delta) = \sigma'$, and (2) for all $\epsilon \in (0, \delta)$, $inv(l)[f(\epsilon)]$ is true as well as $flow(l)[f(\epsilon) \circlearrowleft \dot{f}(\epsilon)]$ is true, where $\dot{f}$ denotes the first derivative of f. Function f is also called a* witness *of the transition.*

□

Note that the definitions above do not explicitly discriminate continuous and event variables. We however assume that every witness evaluates to $\perp$ for each variable in V^e in the interval $[0, \delta]$. Further note that transition systems of hybrid automata are generally *stuttering invariant*. That is, for any $(l, \sigma) \xrightarrow{\delta} (l', \sigma')$ in the transition system, and $\delta_1, \delta_2 \in \mathbb{R}^{\geq 0}$ such that $\delta_1 + \delta_2 = \delta$ exist $(l, \sigma) \xrightarrow{\delta_1} (l'', \sigma'')$ and $(l'', \sigma'') \xrightarrow{\delta_2} (l', \sigma')$ in the transition system, too.

Trajectories and Traces The labeled transition system S_M of an automaton M defines all possible (infinite) evolutions of M. Every path through the transition system represents a valid execution of the automaton. We denote a valid execution path a *trajectory* of S_M. A trajectory is an infinite sequence $\tau = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \dots$ such that (1) $q_0 \in Q_0$, and (2) each $q_i \xrightarrow{a_i} q_{i+1} \in \to$.

Definition 14. *Given a trajectory $\tau = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \dots$, we define*

- *The* prefix *of length n, $\tau_n = q_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-1}} q_n$.*
- *The* duration *of every prefix $\tau_n = q_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-1}} q_n$ as follows:*
  - $d(\tau_0) = 0$
  - $\tau_{n+1} = \tau_n \xrightarrow{e_n} q_{n+1} \implies d(\tau_{n+1}) = d(\tau_n)$
  - $\tau_{n+1} = \tau_n \xrightarrow{\delta_n} q_{n+1} \implies d(\tau_{n+1}) = d(\tau_n) + \delta_n$

□

We are also interested in abstractions of trajectories where only event variables remain visible. To this end, we use variable valuations of event variables V^e for the definition of *events*. Given a valuation $\sigma \in \Sigma_{V^e}$, we identify σ with a set of events $\vec{\sigma} = \{v \mid \sigma(v) \neq \bot\}$.

A *timed trace* is a sequence $\omega = (t_i, \vec{\sigma}_i)_{i \in \mathbb{N}}$ where the t_i form a monotonic, non-zeno sequence of points in the time domain, and the $\vec{\sigma}_i$ form a sequence of event sets, according to Definition 1.

Definition 15. *Given a trajectory τ, we say, ω is the trace of τ iff for each $\tau_{i+1} = \tau_i \xrightarrow{e_i} q_{i+1}$ exists $(t_i, \vec{\sigma}_i) \in \omega$ such that $t_i = d(\tau_i)$ and $\vec{\sigma}_i = \vec{e_i}$, and vice versa.* □

4.4 Automaton Composition

Given two automata M_1 and M_2, we define the semantics of their parallel composition $M_1||M_2$. Automata interact via their input and output variables, respectively. We require that variables are not private variables of two different automata, and the same holds for output variables. More precisely, we require that $V_{P1} \cap V_2 = \emptyset$, $V_1 \cap V_{P2} = \emptyset$ and $V_{O1} \cap V_{O2} = \emptyset$. Hence, composition is a partial function.

We define composition on the transition systems of the involved automata:

Definition 16. *Given S_1 and S_2, the product $S_1 \otimes S_2$ is a transition system with $Q = Q_1 \times Q_2$, $Q_0 = Q_1^0 \times Q_2^0$ and label set $A_\otimes = \Sigma_{V_1^e \cup V_2^e} \cup \mathbb{R}^{\geq 0}$.*

We define $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$ iff there are transitions $q_1 \xrightarrow{a_1} q_1'$ and $q_2 \xrightarrow{a_2} q_2'$ such that $a_1 = a|_{V_1^e}$ and $a_2 = a|_{V_2^e}$.

We define $(q_1, q_2) \xrightarrow{\delta} (q_1', q_2')$ iff (i) there are transitions $q_1 \xrightarrow{\delta} q_1'$ and $q_2 \xrightarrow{\delta} q_2'$, and (ii) $\forall \delta_1, \delta_2 > 0 : \delta_1 + \delta_2 = \delta \implies \exists q_1'', q_2'' : (q_1, q_2) \xrightarrow{\delta_1} (q_1'', q_2'') \xrightarrow{\delta_2} (q_1', q_2')$. □

The latter part (ii) ensures that the valuation of shared variables is consistent among the involved automata. It means that valuations $\sigma_1 \in \Sigma_{V_1}$ and $\sigma_2 \in \Sigma_{V_2}$ always correspond on shared variables, i.e., $v \in V_1 \cup V_2 \implies \sigma_1(v) = \sigma_2(v)$. Enforcing composition to be also stuttering invariant is a mean to achieve this important property.

5 Intermediate Format

While the definition of automata in Chapter 4 allows for a very general class of discrete and continuous evolutions of variables, a more implementation oriented notion of automata is desired for the definition of concrete generators and monitors. To this end, we define a suitable subclass of the elements of automata. We employ a kind of pseudo code for the definition to simplify the presentation. It should be noted that any similarities of the pseudo code with the programming language C are purely coincidental and not intended.

According to Definition 12, an automaton consists of variables, locations, transitions, and labeling functions for state invariants and flows, as well as guards, actions and assignments for transitions. In the following, we will provide corresponding definitions for each of these elements.

5.1 Data Types

First, we define a set of data types for the value domains of the variables of an automaton. We allow for primitive data types with the following meaning:

- `bool` : $\{true, false\}$
- `int` : $\mathbb{Z}$
- `real` : $\mathbb{R}$

We also define computational subsets of these primitive data types:

- `int8`, `int16`, `int32`, `int64`
- `uint8`, `uint16`, `uint32`, `uint64`
- `float`, `double`

Complex data types can be defined based on the primitive types above:

- `struct { int16 x; bool y; }`. The elements of a struct can be accessed by specifying their member name preceded by a dot (`.`).
- `hash<float,uint32>` where the first parameter defines the data type of the hash keys, and the second the value data type. Elements in a hash table can be accessed by specifying a key enclosed in square brackets (e.g. `[1.2]`), or by calling `value(1.2)`. For every hash table exist functions `insert,remove` and `contains`. Function `insert` adds or replaces an element in the hash table. Function `remove` removes an element from the hash table. Function `contains` returns a value of type `bool` indicating whether an element exists in the hash table.

  Hash tables are iteratable (cf. Section 5.6). Iteration delivers the keys of the hash table in arbitrary order.

- `multihash<float,uint32>` where the first parameter defines the data type of the hash keys, and the second the value data type. In contrast to `hash`, a multi-hash can store multiple elements for the same key. For multi-hash tables exist functions `insert,remove` and `count`. Function `insert` adds an element in the hash table. Function `remove` comes in two flavors. The function `remove(key)` removes all elements stored for `key`. Calling `remove(key,elem)` removes the element `elem` from the hash table which has been added first. Function `count` returns the number of elements stored for a key, and `values(key)` returns the list of values stored for the key.

- `list<double>`. Elements in a list are accessed by an index that is type compatible with (i.e. is a subset of) the data type `integer`, enclosed in square brackets (e.g. `[7]`). The first index of a list is `0`. For every list exist functions `length, append, prepend, removeFirst` and `removeLast`. Function `length` returns the number of elements in the list. Functions `append` and `prepend` add elements to the end and beginning of the list, respectively.

  Lists are iteratable (cf. Section 5.6. Iteration delivers the elements of the list in ascending order.

- `array<int8>[`$length$`]`. Elements in an array are accessed by an index that is type compatible with (i.e. is a subset of) the data type `integer`, enclosed in square brackets (e.g. `[7]`).

  Arrays are iteratable (cf. Section 5.6. Iteration delivers the elements of the array in ascending order.

All data types except `clock` can be defined as event data types:

```
event int16
```

Event data types can be further refined to be identifiable:

```
event id struct { x : int16; y : bool; }
```

Identifiable event variables provide function `id`, which refers to the identifier of the event occurrence.

Clocks and Timer For dealing with time, we define two particular data types, namely `clock` and `timer`. Variables of type `clock` represent clocks defined in the specifications. A clock of the global time domain is defined as follows:

```
clock c;
```

For “local” clocks, the data type comes with parameters. There are two possible parameter sets:

```
clock<mindrift,maxdrift,skew,resolution> c;
```

and

```
clock<maxdiff,resolution> c;
```

where the parameters correspond to the local clock definitions in Section 3.7. The parameter `resolution` can be omitted, in which case the clock is assumed to be dense.

Variables of type `timer` are used to refer to clocks. A timer that refers to clock `c` has the same derivate than `c`, as well as the same jumps of the same height. The only exception is that timers can be reset (set to 0). The definition of a timer variable is as follows:

```
timer<c> t;
```

5.2 Variables

Variables are defined according to the automaton specification of Definition 12 either as input, private or output variables. In its simplest form, a variable is specified by a name and a data type, preceded by its role:

```
input int16 x;
```

Variables of the different roles (input, private, output) can be grouped in blocks:

```
input:
  int16 x;
  ...
private:
  ...
output:
  ...
```

Variables can be defined as event types, which carry values only at time points when they are set, i.e., get a value assigned:

```
input:
  event int16 x;
```

Event variables can be further defined to be identifiable. Identifiable event variables get a (hidden) `id` element, that can be retrieved similarly to the member of a `struct` variable. The `id` element always has the data type `int32`:

```
input:
  event id double y;
private:
  int32 myid;
...
myid = y.id;
```

The value of an identifiable event variable can be stored and retrieved:

```
input:
  event id double y;
private:
  int32 myid;
  array<event id double>[7] store;
...
store[i] = y;
...
myid = store[3].id
```

All variables are either explicitly initialized or implicitly by a default initializer. For primitive types, the default initializer is 0, except for the data type `bool`, where the default initializer is `false`. List and hash tables are initially empty. An array is initialized with the default initializer for all its elements, and the same holds for structs.

Primitive variables can be explicitly initialized while being defined:

```
input:
  int16 x = 7;
  double y = 1.2;
  bool b = true;
```

5.3 Locations

According to Definition 12, an automaton consists of a set L of locations. For implementation purposes, every location is identified by a name. Further, every location gets an invariant assigned. A location hence is defined by the name of the location, followed by a sequence of predicate statements defining the invariant (inv) of the location:

```
loc1:
  t <= 500;
  ...
```

Logically, the invariant statements in such sequence are interpreted as conjunction.

Every automaton must define at least its initial location. This location has the predefined name `init`.

A location may contain a flow section, where the flow predicate is defined:

```
loc1:
  // invariant
  ...
  flow:
    ...
```

5.4 Transitions

Transitions are defined in the context of their source location. A transition is characterized by its target location, its guard $(guard)$, and action (act) as well as set (set) statements:

```
loc1:
  // invariant
  ...
  // flow
  ...
  trans loc2:
    guard:
      t <= 500;
      input1;
      input2 == 17;
    act:
      output = 8;
    set:
      store.append(input);
    ...
loc2:
  ...
```

The example above shows three event variables, namely `input1,input2` and `output`. The statement `input1;` means that the guard applies to any event value on the `input1` variable. The statement `input2 == 17;` means that the guard applies only to event value `17` on the `input2` variable. The statement `output = 8;` sets the value of the variable at the point in time where the transition is taken. Note that variable assignments in the `act` section can only be applied to event variables. Furthermore, assignments to event variables are not allowed in the `set` section.

Note that the distinction between `act` and `set` section is syntactic sugar in the intermediate format. The difference between actions and assignments is defined by the port types (i.e. whether it defines an event or not), and the respective reading and sending operations. Hence, the respective statements are allowed in both sections.

5.5 Automata

In order to put things together, we finally define the structure of a set of automaton definitions. Every specification may start with a set of global data type definitions, followed by a set of automaton definitions:

```
typedef array<event id double>[7] eventarray;
...
automaton A1 {
  // variables
  input:
    ...
  private:
    eventarray store;
    ...
  output:
    ...

  // locations
  init:
    ...
}

automaton A2 {
  ...
}

...
```

Data types defined by `typedef` start with the data type definition and end with the type name. All defined data types can be referred in the variable sections of any automaton. Automata are defined by the keyword `automaton`, followed by a name and its variable, location and transition definitions enclosed in curly brackets.

5.6 Functions and other Extensions

In order to simplify the presentation, we will occasionally introduce additional constructs beyond those presented above. Firstly, we make use of additional functions, for which we usually do not provide a mathematical definition, but which are simple enough for intuitive understanding. For example, we make use of a function random(max), which returns a randomly generated value from the interval $[0, \mathtt{max}]$ using uniform distribution.

We also introduce additional data types for internal book keeping tasks within the automata. Normally, those data types are defined in a C++ class style. That means, functions associated with those data types are used in the form type.func().

Furthermore, we introduce constructs for selections and iterations. The construct

```
cond => assignment;
```

executes the assigment statement only if cond evaluates to true. Note that this construct is consistent with the formal assigment function defined in Chapter 4, as it takes valuations as arguments. The construct can also be used for invariants and guards. Here the construct

```
cond1 => cond2;
```

expands to !cond1 || cond2.

For repetition, we introduce the foreach construct:

```
foreach (var,store)
{
  // do something with var
}
```

iterates over all elements in the variable store. The semantics of iterations are defined by unrolling, i.e., the block definition of the construct is applied to all its elements in parallel.

A more sophisticated extension is the while construct:

```
while (condition)
{
}
```

which iterates the block definition while condition evaluates to true. There is in fact no consistent relation to the automaton semantics defined in Chapter 4 in a simple transition. It requires instead multiple locations and transitions.

For convenience and for improving readability, we define the special invariant urgent, which is true only in the particular moment when a location is entered. It can be used to force an automaton to stay in a location only for a single time point. This invariant could be implemented by a distinct clock, say urgentclock, which is reset to 0 in every transition. The corresponding invariant is urgentclock == 0.

6 Generators and Monitors

While Chapter 3 defines semantics of timing specifications, the notion of timed traces is not well suited to generate or observe event streams according to these specifications. This chapter defines semantics of MULTIC timing specifications in terms of automata, which have been introduced in Chapter 4. For each specification pattern introduced in Chapter 3, the present chapter defines two (sets of) automata. The first one defines a *generator*, which is able to produce every trace that adheres to the specification. The second one defines a *monitor* that recognizes every trace that adheres to the specification.

In order to ease the implementation of automaton specifications, the definitions will be given in the notion of the intermediate format discussed in Chapter 5. The relation between the two definitions is sufficiently clear so that the reader should be able to translate one representation into the other. The relation between the automaton semantics on one hand and the trace semantics on the other hand is not that obvious, but certainly key. We expect that the automata definitions discussed in this section indeed reflect the intended semantics. We will provide hints and references that give reason to the relation between the two representations. However, a comprehensive analysis is highly expensive in labor and out of scope of the present work.

6.1 Variables, Ports and Data Types

The interested reader may have recognized a small deviation between the definition of traces in Definition 1 and Chapter 4. Where the former defines traces based on ports, the latter refers to variables. The procedure is intentional, because putting the definition of automata into the context of component based design would have added an unnecessary layer of complexity to the definitions. In the following, we overcome the deviation by identifying all input and output variables with ports. Doing so results in convergence of the two definitions.

The notions of both variables and ports however rely on data types that define which values can occur. Timing specifications in Chapter 3 do not provide for definition of data types as this is out of scope of the specification language. As the automata require data types to be defined for variable and port definitions, which do not come from the timing specifications, we introduce a common notion of type definition. Events are specified by a port optionally followed by a value:

$$\mathit{Event} \quad :: \quad \mathit{Port} \mid \mathit{Port}\ \texttt{'.'}\ \mathit{EventValue}$$

For every port that is referred in a specification, we assume a type definition of the form `typedef event ... PortX`, where `PortX` identifies the corresponding port. The actual data type is hereby left open. If multiple ports are referred in a specification, the context is explicitly stated.

Events occur in two different roles, namely as input and output events. Note that these roles are no concept of the specification itself but of the automaton semantics. The role of a particular port is defined by the role of the automaton. For monitor automata, all ports are defined as input variables. For generators, the role depends on the specification. For example, in the specification `whenever E1 occurs then E2 occurs...`, the event `E1` has the role input event, and `E2` has the role output event.

Because events may occur either in form of a port name, or with a dedicated event value, the automata specifications fastly become complex and unintuitive. We ease the specification by two "macros". For output events, we introduce the macro `hasEvent(event)`, which expands simply to `event` if the event specification refers to a port. It expands to `event == value` if the specification is of the form `event.value`.

Output events occur in the `act` sections of transition definitions in an automaton, which requires a value for that event. If a specification defines a port only, then it adheres by definition to every possible event value on that port. We thus introduce a built-in function `selectValue(Type)`, which produces randomly a value of the specified data type. For output events, we further introduce the macro `sendEvent(event)`, which expands to `event = selectValue(Type)` if the event specification refers to a port. It expands to `event = value` if the specification is of the form `event.value`.

For causal patterns, we extend the definitions by assigning tuples of values to events, such as in `event = value,id` where `event` gets value `value` and id `id`. We also introduce the corresponding function `sendEvent(Port,id)`, which ensures that the sending event has id `id`.

Function *EventSpec*	*Port*	*Port.EventValue*
`hasEvent(Port)`	`Port`	`Port == EventValue`
`sendEvent(Port)`	`Port = selecValue(Type)`	`Port = EventValue`
`sendEvent(Port,id)`	`Port = selecValue(Type),id`	`Port = EventValue,id`

6.2 Event Sequences and Sets

Some specification patterns reason about sequences or sets of events. According to Chapter 3, the semantics of an event sequence $es = (e_1, ..., e_n)$, is as follows. A sequence $(t_i, \sigma_i), ..., (t_{i+n-1}, \sigma_{i+n-1}) \in \omega$ in a trace satisfies es if every (t_{i+k}, σ_{i+k}), $0 \leq k < n$, satisfies the event specification e_k. Monitors that recognize event sequences must be able to store the portion of an event sequence it has already detected. To this end, we introduce a suitable data type and respective functions.

The data type is an event sequence store with type name `EventSequence`, and can store events of arbitrary data type. The event sequence that shall be recognized is defined by iteratively calling the function `addToExpr()`. For example, in order to recognize the even sequence `(Event1, Event2, Event3)`, the sequence store is initialized as follows:

```
EventSequence store;
...
store.addToExpr(Event1);
store.addToExpr(Event2);
store.addToExpr(Event3);
```

Events are added to the sequence by the function `addEvent()`. The function

```
store.addEvent(e);
```

adds event `e` to the recognized event sequence. Note that the addition of the first event of the defined event sequence clears the list. Also the addition of an event that does not match the event sequence clears the list:

```
// event sequence is (a,b,a,c,d)
store.addEvent(a); // state = (a)
store.addEvent(b); // state = (a,b)
store.addEvent(a); // state = (a,b,a)
store.addEvent(c); // state = (a,b,a,c)
store.addEvent(a); // state = (a)
store.addEvent(c); // state = ()
```

In order to check for completeness of event sequences, the function `complete()` returns a boolean value that states whether the stored events match the specified event sequence. The number of events that are required to complete the expression can be obtained with function `remainingEvents()`. Function `clear()` empties the list.

For event sets, the complementary data types `EventSet` is introduced, which provides the same set of functions. The addition of events however results in clearing the list only if an event is added that was already in the list.

```
EventSet store;
...
// event set is {a,b,c,d}
store.addEvent(a); // state = {a}
store.addEvent(c); // state = {a,c}
store.addEvent(b); // state = {a,b,c}
store.addEvent(a); // state = {a}
store.addEvent(c); // state = {a,c}
store.addEvent(c); // state = {c}
```

Both data types provide two further functions. For event sequences, the function `nextEvent()` delivers the next expected event. For event sets an event from the missing ones is randomly selected. Note that once an event is selected, successive calls of the function return the same value. The function `matchEvent(e)` returns whether event `e` can be used to extend the stored sequence (or set).

Function	Semantics
`addToExpr(e)`	Adds event `e` to the store as expected event for initialization.
`addEvent(e)`	Adds event `e` to the store.
`nextEvent()`	Returns the next expected event.
`matchEvent(e)`	Returns whether `e` belongs to the next expected events.

6.3 Intervals

Interval parameters come in different flavors: as a single value or as an interval with either open or closed bounds in any combination. We also introduce macros for predicates that check for lower and upper bounds, respectively, of interval specifications. Given a clock variable `clock t` and an interval `I`, we introduce the macros `matchLB(I,t)`, `matchUB(I,t)`, `equalUB(I,t)`, `greaterUB(I,t)` and `geqUB(t,I)`. The macros expand with respect to `I` as follows, where `L` defines the lower and `U` the upper bound of `I`:

I	V	]L,U[	]L,U]	[L,U[	[L,U]
matchLB(I,t)	t == V	t > L	t > L	t >= L	t >= L
matchUB(I,t)	t == V	t < U	t <= U	t < U	t <= U
equalUB(I,t)	t == V	na.	t <= U	na.	t <= U
greaterUB(I,t)	t > V	t >= U	t > U	t >= U	t > U
geqUB(I,t)	t >= V	na.	t >= U	na.	t >= U

The additional macro `matchBounds(I,t)` returns the conjunction of `matchLB(I,t)` and `matchUB(I,t)`. It is important to note that the function `equalUB(I,t)` can be used only for right-closed intervals. The same holds for function `geqUB(I,t)`. Hence, every automaton that uses these functions is restricted to use it only for right-close intervals.

6.4 Sliding Windows

The reaction and the age pattern can be specified with a `'K out of N'` construct. In order to maintain sliding windows that keep track of satisfied and unsatisfied event expressions, we introduce the data type `Window`, which takes the parameters `K` and `N` on instantiation. The data type is basically an ordered hash table, which takes clock values as keys, and booleans as values.

Several functions are defined for the data type. Function `first()` returns the value of the element with the smallest key. Function `append(t,v)` adds a new element with key `t` and value `v` to the hash table. If more than `N` elements are in the hash table, then the element with the smallest key is removed. Function `append(t)` adds a new element with key `t` and value `false` to the hash table. The function `setSat(t)` sets the value with key `t` to `true`. The function `isSat(t)` retrieves the value with key `t`. Function `isSat()` returns whether there are no more than `N-K` elements with value `false` in the hash table. Function `isUnsat()` returns `!isSat()`. The function `isFailed(t,I)` returns `true` if the function `isUnsat()` returns `true`, and for the smallest key `u` in the hash table holds `greaterUB(I,t-u)`. Function `wouldBeUnsat(n)` returns true if the window would be unsatisfied if `n` (unsatisfied) elements would be added via `append()`. Finally, function `wouldBeUnsat()` returns the same as `wouldBeUnsat(1)`.

Function	Semantics
`first()`	Returns value of first element.
`append(t,v)`	Adds (t,v) to the hash table, ensuring max. size `N`.
`append(t)`	Adds (t,false) to the hash table, ensuring max. size `N`.
`setSat(t)`	Sets value of element `t` to `true`.
`isSat(t)`	Returns value of element `t`.
`isSat()`	Returns whether number of elements with value `false <= N-K`.
`isUnsat()`	Returns `!isSat()`.
`isFailed(t,I)`	Returns `isUnsat()` and exists element `u` with `greaterUB(I,t-u)`.
`wouldBeUnsat(n)`	Returns `true` if addition of `n` unsatisfied elements leads to `isUnsat()`.
`wouldBeUnsat()`	Returns `wouldBeUnsat(1)`.

We also exploit lists and hash tables over real values for those patterns:

```
list<real> times
hash<real,X> timestore1
multihash<real,X> timestore1
```

and introduce several functions for these container types. Function `matchingBoundCount(c,I,t)` returns the number of elements/keys `u` in the container `c` for which `matchBounds(I,t-u)` holds. The function `hasMatchingBound(c,I,t)` returns true if `matchingBoundCount(c,I,t)` > 0. Function `hasMatchingUB(c,I,t)` delivers true if for any of the elements/keys `u` holds `matchUB(t-u,I)`. The function `hasEqualUB(c,I,t)` delivers true if for any of the elements/keys `u` holds `equalUB(I,t-u)`. The function `equalUBCount(c,I,t)` returns the number of elements for which `equalUB(I,t-u)` holds. Function `removeMatchingBound(c,I,t)` removes all values for which `matchBounds(I,t-u)` holds. Function `removeGreaterBounds(c,I,t)` removes all values for which holds `greaterUB(I,t-value)`.

Function	Semantics
`matchingBoundCount(c,I,t)`	# u with `matchBounds(I,t-u)`.
`hasMatchingBound(c,I,t)`	Returns `matchingBoundsCount(c,I,t)` > 0.
`hasMatchingUB(c,I,t)`	True if for any element/key u holds `matchUB(I,t-u)`.
`equalUBCount(c,I,t)`	# u with `equalUB(I,t-u)`.
`hasEqualUB(c,I,t)`	Returns `equalUBCount(c,I,t)` > 0.
`geqUBCount(c,I,t)`	# u with `geqUB(I,t-u)`.
`removeMatchingBound(c,I,t)`	Removes all u with `matchBounds(I,t-u)`.
`removeGreaterBounds(c,I,t)`	Removes all u with `greaterUB(I,t-u)`.

6.5 Causal Event Relations

Causal event relations are required to determine the exact behavior of the corresponding causal reaction and age patterns (cf. Section 6.11 and Section 6.12, respectively). We expect to have a definition of the causal event relation in a specification that contains such causal pattern. In order to implement generators and monitors for these pattern, we introduce respective event stores `LIFO`, `FIFO` and `IDSTORE`. These stores are very similar to hash tables, except that they provide access to their elements only with respect to their intended semantics. For example, one can remove only the most recently added element from a `LIFO` store. The following functions are defined for causal stores.

Function `insert(t,e)` adds element e to the store with time stamp `t`. Function `hasMatch(I,t)` returns whether the store has a matching element. Matching means that the macro `matchBounds(I,t)` returns true. Which elements are considered by `hasMatch(I,t)` depends on the store type. For `LIFO` stores, only the most recently added element is considered, and for `FIFO` only at the least recently added element, respectively. `IDSTORE` stores consider all elements. Function `removeMatch(I,t)` removes the next matching element (i.e. for which `matchBounds(I,t)` holds). This event can be retrieved by the function `nextMatchEvent(I,t)`.

Function `hasMatch(I,t,id)` is similar to the function with the same name but implements only two parameters. It returns whether the store has a matching element with id `id`. Function `removeMatch(I,t,id)` removes the first element with id `id`, for which `matchBounds(I,t)` holds.

Function `hasEqualUB(I,t)` returns whether there is any element for which `equalUB(I,t)` holds. Function `hasGreaterUB(I,t)` returns whether there is any element for which `greaterUB(I,t)` holds. Function `removeGreaterUB(I,t)` removes all elements for which `greaterUB(I,t)` holds.

Function	Semantics
`insert(t,e)`	Inserts `(t,e)`.
`hasMatch(I,t)`	Returns whether `(u,e)` with `matchBounds(I,t-u)` exists.
`hasMatch(I,t,id)`	Returns whether `(u,e.id)` with `matchBounds(I,t-u)` exists.
`nextMatchEvent(I,t)`	Returns the next event for which `hasMatch(I,t)` holds.
`nextMatchEvent(I,t,id)`	Returns the next event for which `hasMatch(I,t,id)` holds.
`removeMatch(I,t)`	Removes the event `(u,e) = nextMatchEvent(I,t)`.
`removeMatch(I,t,id)`	Removes the event `(u,e.id) = nextMatchEvent(I,t,id))`.
`hasEqualUB(I,t)`	Returns whether `(u,e)` with `equalUB(I,t-u)` exists.
`hasGreaterUB(I,t)`	Returns whether `(u,e)` with `greaterUB(I,t-u)` exists.
`removeGreaterUB(I,t)`	Removes all`(u,e)` with `greaterUB(I,t-u)`.

6.6 Dealing with Clocks

In order to realize observations with multiple clocks on one hand, and because of the fact that clocks may be referred in multiple components on the other hand, we follow an approach where clocks are provided by separate automata and used by generator and monitor automata in terms of input variables.

For every clock defined by a `Clock Name has ...` pattern, we construct a corresponding "clock" automaton with a single output variable that realises the respective clock. The individual clock variables in turn are referred in the various generator and monitor automata as input variables (cf. Section 6.7 ff). Because of the large number of possible combinations of clock properties, we omit a detailed discussion of how to express the required invariants and flows for an adequate automaton representation. The general schemes are detailed in Section 3.7. Instead, we assume for each clock variable a corresponding invariant `clock_inv()` and flow predicate `clock_flow()`. Note that these predicates may contain randomly generated constants due to non-determinism induced by the clock, such as clock drift intervals and skews.

Clocks with finite resolutions also require the definition of discrete transitions for clock steps. Hence, we also introduce a respective predicate `clock_step()` that indicates the guard for such steps. To summarize, the general structure of a clock automaton is shown below:

```
automaton ClockName {
  output:
    clock c;  // Here come the corresponding parameters of the clock, e.g.
              //  clock<mindrift,maxdrift,skew,resolution> c;
              //  clock<maxdiff,resolution> c;

  init:
    clock_inv(c);
    clock_flow(c);

    trans init:
      guard:
        clock_step(c);
}
```

It is important to note that clocks with finite resolution require for all involved automata the possibility to perform a corresponding transition according to the definition of automata composition discussed in Section 4.4. In order to keep the presentation simple, we omit those *stuttering* transitions in the presentation of the automata.

Note that the approach of instantiating clock automata to simulate local clocks remains valid for exhaustive verifications. Because the invariants and flows contain randomly generated parameters (such as drift functions and skew), formal verification takes all possible combinations of such parameters into account, which in turn establishes the desired kind of non-determinism when dealing with local clocks.

6.7 Single Event Pattern

The most simple specification pattern is the *SingleEvent* pattern:

SingleEvent :: *EventList* **occurs within** *Interval* **.**

To demonstrate the use of automata for the *SingleEvent* pattern and the pattern that follow in the next chapters, we make use of example instances. Here we define the automata by the following pattern instance as the example **'e,f occurs within I'**.

6.7.1 Generator

The generator consists of a simple automaton with two locations. After the (single) event has been send the automaton remains in the final location, silently.

```
typedef event ... Port;

automaton SingleEventGen {
  input:
    clock c;
  output:
    Port e;
    Port f;
  private:
    timer<c> t = 0;

  init:
    // we have to leave the state when upper bound elapses
    matchUB(I,t);

    trans final:
      guard:
        // Transition can be taken within the interval
        matchBounds(I,t);
      act:
        sendEvent(e);
        sendEvent(f);
  final:
    // remain in this location
}
```

6.7.2 Monitor

The monitor of the single event pattern is very similar to the generator. The automaton has a third location bad, which is reached when the pattern is violated.

```
typedef event ... Port;

automaton SingleEventMon {
  input:
    Port e;
    Port f;
    clock c;
  private:
    timer<c> t = 0;

  init:
    // we leave the state when Interval has elapsed
    !greaterUB(I,t);
    trans good:
      guard:
        matchBounds(I,t);
        hasEvent(e);
        hasEvent(f);
    trans bad:
      guard:
        greaterUB(I,t);

  good:
    // satisfied: remain in this location unless a second event arrives
    trans bad:
      guard:
        hasEvent(e);
    trans bad:
      guard:
        hasEvent(f);

  bad:
    // violated: remain in this location
}
```

6.8 Repetition Pattern

Recurring event occurrences are specified with the *Repetition* pattern:

Repetition	::	*EventList* **occurs every** *Interval*$_1$ [**with** *RepetitionOptions*]? **.**
RepetitionOptions	::	*Jitter* [**and** *Offset*]? | *Offset* [**and** *Jitter*]?
Jitter	::	**jitter** *TimeExpr*
Offset	::	**offset** *Interval*$_2$

The pattern comes with optional jitter and offset specifications. In the following, we discuss the version with both parameters set. The version without jitter is derived from the one with jitter in a canonical way. The same holds for the offset.

We define the automata by the pattern instance '**e,f occurs every I with jitter J and offset O**' as an example.

6.8.1 Generator

The generator of the pattern consists of two automata. The first one generates events periodically according to the *Interval*$_1$ parameter. The second one adds the jitter. The first automaton is very similar to the *SingleEvent* pattern automaton. The main difference is that the automaton is reset after sending an event. The first event may occur in the interval $[O^-, O^+]$, possibly further delayed by the jitter.

The generator automata are an extension of the one defined in [5], together with a proof that it adheres to the trace semantics of the specification pattern. The first automaton produces events within the minimal and maximal time bound. The second automaton adds the jitter delay to these events. For the jitter automaton, we introduce the functions `random()` and `getSmallestKey()`. The `random(U)` function delivers a real value between 0 and U. The `getSmallestKey()` function delivers the smallest key in a hash or multi-hash table. In our case, the function delivers the next time point in which events have to be send.

```
typedef event ... Port;

automaton RepetitionPeriodGen {
  input:
    clock c;
  output:
    Port ip; // 'internal' port
  private:
    timer<c> t = 0;

  init:
    // we have to leave the state when upper offset bound elapses
    matchUB(O,t);

    trans loop:
      guard:
        matchBounds(O,t);
      act:
        sendEvent(ip);
      set:
        // reset the timer clock
        t = 0;

  loop:
    // we have to leave the state when upper bound elapses
    matchUB(I,t);

    trans loop:
      guard:
        matchBounds(I,t);
      act:
        sendEvent(ip);
      set:
        // reset the timer clock
        t = 0;
}

automaton RepetitionJitterGen {
  input:
    Port ip; // 'internal' port
    clock c;
  output:
    Port e;
    Port f;
  private:
    timer<c> t = 0;
    real  next = 0;
    multihash<real,Port> events;

  init:
    (t <= next) || (events.count(next) == 0);

    trans init:
      guard:
        hasEvent(ip);
      set:
        // Add event with the time offset and re-calculate next
        events.insert(t + random(J),ip);
        next = getSmallestKey(events);

    // We go into the send location if time is up
    trans send:
      guard:
        t >= next;

  send:
    urgent;  // We do not stay in this location

    // Sending events is iterated as long as further events are
    // in the event list for the time point.
    trans send:
      guard:
        events.count(next) >= 1;
      act:
        sendEvent(e,events.value(next));
        sendEvent(f,events.value(next));
      set:
        events.remove(next);
    trans init:
      guard:
        events.count(next) == 0;
      set:
        next = getSmallestKey(events);
}
```

6.8.2 Monitor

Interestingly, accurate monitoring of the *Repetition* pattern is not possible in general. This is because the combination of (minimal and maximal) period with large jitters prevents deciding which fraction of a time period belongs to which parameter. Large means in our context jitter larger than the minimal period.

But also for $J \leq P^-$, construction of an accurate monitor is not obvious. The following proposition provides suitable properties for such construction. Semantics of the repetition pattern is the set of traces $\{(e_i, t_i)_{i\in\mathbb{N}} \mid t_i = u_i + j_i, u_0 \in [O^-, O^+], u_i - u_{i-1} \in [P^-, P^+], j_i \in [0, J]\}$. For the discussion we focus on time sequences, and define the language $L_{Rep} = \{(t_i)_{i\in\mathbb{N}} \mid t_i = u_i + j_i, u_0 \in [O^-, O^+], u_i - u_{i-1} \in [P^-, P^+], j_i \in [0, J]\}$.

Proposition 1. *Given a time sequence $\omega = (t_i)_{i\in\mathbb{N}}$. We define sets $J_i = [J_i^-, J_i^+]$ for $i \in \mathbb{N}$ as follows:*

- $J_0^- = \max(0, t_0 - O^+)$,
- $J_0^+ = \min(J, t_0 - O^-)$,
- $J_i^- = \max(0, t_i - t_{i-1} - P^+ + J_{i-1}^-)$,
- $J_i^+ = \min(J, t_i - t_{i-1} - P^- + J_{i-1}^+)$.

Then the following holds:

(a) $\omega \in L_{Rep} \Leftrightarrow \forall i \in \mathbb{N} : J_i^- \leq J_i^+ \wedge J_i \subseteq [0, J]$

(b) $\omega \in L_{Rep} \Rightarrow \forall (j_i)_{i\in\mathbb{N}} \in (J_i)_{i\in\mathbb{N}} : \exists (u_i)_{i\in\mathbb{N}} : \forall i \in \mathbb{N} : t_i = u_i + j_i \wedge u_0 \in [O^-, O^+] \wedge u_i - u_{i-1} \in [P^-, P^+]$.

□

Property (a) states that for any time sequence $\omega = (t_i)_{i\in\mathbb{N}}$, the calculated sets J_i are not empty and within the jitter bounds if and only if ω is a sequence of the repetition pattern ($\omega \in L_{Rep}$). While property (b) looks like a tautology, it states that an observed time sequence, if it belongs to the language, can indeed be constructed by any jitter j_i that belongs to the calculated jitter set. In other words, it can be constructed by parameters u_i and j_i for any $j_i \in J_i$.

The proposition provides an approach for the construction of a corresponding monitor. The calculation of the parameters J_i^-, J_i^+ shows how they can be updated iteratively. Property (a) provides the satisfaction check for the pattern.

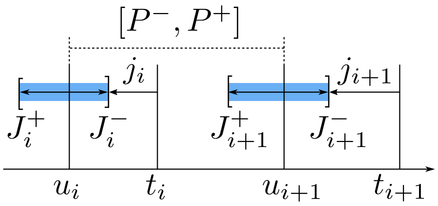


Figure 6.1: Degree of freedom of choice for jitters j_i, j_{i+1} when t_i, t_{i+1} are observed

The proof of the proposition is inductive, and needs both properties:
$i = 0$: We know by definition that for $\omega \in L_{Rep}$ holds $t_0 = u_0 + j_0, u_0 \in [O^-, O^+], j_i \in [0, J]$. We further observe that by definition holds $0 \leq J_o^-$ and $J_0^+ \leq J$. We distinguish four cases. The second and third case considers $\omega \in L_{Rep}$, which is equivalent to $O^- \leq t_0 \leq O^+ + J$.

1. $t_0 < O^-$: Here, t_0 cannot be member of an $\omega \in L_{Rep}$ by definition. Furthermore, we get $J_0^+ = t_0 - O^- < 0$, and hence $J_0^- > J_0^+$.
2. $O^- \leq t_0 \leq O^+$: $\Leftrightarrow t_0 - O^+ \leq 0 \Leftrightarrow J_0^- = \max(0, t_0 - O^+) = 0$. Furthermore, we have $t_0 - O^- \geq 0 \Leftrightarrow J_0^+ = \min(J, t_0 - O^-) \in [0, J]$. Because $J_0^- = 0$ and $J_0^+ \in [0, J]$ we get (a). Furthermore, for every $J_0^- = 0 \leq j_0 \leq \min(J, t_0) = J_0^+$ we can find u_0 such that $t_0 = u_0 + j_0$, which establishes also (b).

3. $O^+ < t_0 \le O^+ + J$: $\Leftrightarrow 0 < t_0 - O^+ \le J \Leftrightarrow J_0^- = t_0 - O^+ \in (0, J]$. In order to get (a) we need to show $J^- \le J^+$, or equivalently $t_0 - O^+ \le \min(J, t_0 - O^-)$. This is satisfied because (i) $t_0 - O^+ \le t_0 - O^-$ holds trivially, and (ii) $t_0 - O^+ \le J$ by precondition of the case. We establish (b) as follows. Setting $u_0 = O^+$ and $j_0 = J_0^- = t_0 - O^+$ shows the existence for the smallest jitter. Also for $j_0 = J_0^+ = \min(J, t_0 - O^-)$ we can find a corresponding u_0, for which it holds $O^- \le u_0 \le O^+$. It is easy to see that we can find u_0 also for all jitter between J_0^- and J_0^+.

4. $O^+ + J < t_0$: Here, t_0 cannot be member of an $\omega \in L_{Rep}$ by definition. Furthermore, we get $J_0^- = t_0 - O^+ > J$, and hence $J_0^- > J_0^+$. This concludes the induction base.

$i \rightarrow i + 1$: By induction we know that for any $j_i \in J_i$ there is a u_i such that $t_i = u_i + j_i$. Hence, we get (1) $u_i + J_i^- \le t_i \le u_i + J_i^+$.

Because $u_{i+1} - u_i \in [P^-, P^+]$ and $j_{i+1} \in [0, J]$ holds by definition of L_{Rep}, ω belongs to the language only if $t_{i+1} \in u_i + [P^-, P^+] + [0, J]$, or equivalently (2) $u_i + P^- \le t_{i+1} \le u_i + P^+ + J$.

From (1) and (2), we can calculate minimal and maximal distances between t_i and t_{i+1}, for which we get (3) $P^- - J_i^+ \le t_{i+1} - t_i \le P^+ + J - J_i^-$, and hence:

(3a) $P^- - J_i^+ \le t_{i+1} - t_i \Leftrightarrow J_{i+1}^+ = \min(J, t_{i+1} - t_i - P^- + J_i^+) \ge 0.$

(3b) $t_{i+1} - t_i \le P^+ + J - J_i^- \Leftrightarrow J_{i+1}^- = \max(0, t_{i+1} - t_i - P^+ + J_i^-) \le J.$

Furthermore, from $J_i^- \le J_i^+$ we can derive $P^- + J_i^- \le P^+ + J_i^+ \Rightarrow -P^+ + J_i^- \le -P^- + J_i^+ \Rightarrow t_{i+1} - t_i - P^+ + J_i^- \le t_{i+1} - t_i - P^- + J_i^+$ and thus $J_{i+1}^- \le J_{i+1}^+$.

All together establish (a).

Property (b) is established by revisiting (1). Construction of J_{i+1} along (3) ensures that for any $j_i \in J_i$ and $j_{i+1} \in J_{i+1}$, we get $u_{i+1} = t_{i+1} - j_{i+1}$ and $u_i = t_i - j_i$ such that $u_{i+1} - u_i \in [P^-, P^+]$ (cf. Figure 6.1). This concludes the induction step and hence the proof.

Restrictions For monitors, the jitter must be smaller or equal to the minimal period. Furthermore, the period interval must be closed: `I = [Pmin,Pmax]` and `J` $\le$ `Pmin`.

The monitor maintains sets J_i and checks for the property (a) above.

```
typedef event ... Port;

automaton RepetitionMon {
  input:
    Port e;
    Port f;
    clock c;
  private:
    timer<c> t = 0;
    real Jmin = 0;
    real Jmax = J;

  init:
    // (0 <= Jmin) && (Jmin <= Jmax) && (Jmax <= J);

    trans check:
      guard:
        hasEvent(e);
        hasEvent(f);
      set:
        Jmin = max(0,t-Omax);
        Jmax = min(J,t-Omin);
        t = 0;

  run:
    // (0 <= Jmin) && (Jmin <= Jmax) && (Jmax <= J);

    trans check:
      guard:
        hasEvent(e);
        hasEvent(f);
      set:
        Jmin = max(0,t-Pmax+Jmin);
        Jmax = min(J,t-Pmin+Jmax);
        t = 0;

  check:
    urgent;

    trans run:
      guard:
        (0 <= Jmin) && (Jmin <= Jmax) && (Jmax <= J);

    trans bad:
      guard:
        !((0 <= Jmin) && (Jmin <= Jmax) && (Jmax <= J));

  bad:
    // pattern is violated
}
```

6.9 Reaction Pattern

The reaction pattern reasons about distances between event expressions:

Reaction :: **whenever** *EventExpr* **occurs then** *EventExpr* **occurs within** *Interval*
[**once**]? [**,** *Number* **out of** *Number* **times**]? **.**

Note that the generator automaton for the reaction pattern defines two different roles for events. The first event expression refers to input ports. This means, the generator does not generate the whole language of the pattern, but only the event expressions of the specified reaction. Indeed, other interpretations for generators of this pattern could be defined. However, the selected one fits best the intended use cases.

The generator and the monitor exploit the data types EventSequence and EventSet, which have been introduced in Section 6.2 for supporting recognition of event expressions.

We define the automata by the pattern instance ’**whenever {e1,e2} occurs then (e3,e4) occurs**

within I once, K out of N times'. The following automata contain comments for the statements that have to be removed if the optional once is omitted. If the restriction K out of N times is omitted then we assume K = N = 1.

6.9.1 Generator

The generator automaton starts with the definition of all events that occur in the specification. The init location is only visited at system startup for initializing the event stores. The "main loop" is location run.

Restrictions Generators for the pattern can be applied only for intervals which are right-closed, i.e., for intervals of the form (L,U] or [L,U]. The generator also in general cannot enforce once semantics. Hence it is not allowed for generators of the pattern, and the example pattern reduces to '**whenever {e1,e2} occurs then (e3,e4) occurs within I, K out of N times**'.

```
typedef event ... Port1;
typedef event ... Port2;
typedef event ... Port3;
typedef event ... Port4;

automaton ReactionGen {
  input:
    Port1 e1;
    Port2 e2;
    clock c;
  out:
    Port3 e3;
    Port4 e4;
  private:
    timer<c> t = 0;
    EventSet instore;
    list<real> intimes;
    EventSequence outstore;
    Window<K,N> win;

  init:
    // location is used for initialization only
    urgent;

    trans run:
      set:
        instore.addToExpr(e1);
        instore.addToExpr(e2);
        outstore.addToExpr(e3);
        outstore.addToExpr(e4);

  run:
    // We remain in this state unless there are enough unsatisfied
    // input event expressions that would let the sliding window fail.
    !win.wouldBeUnsat(geqUBCount(intimes,I,t));

    // We store all input events.
    // This transition is duplicated for each event ein in the input expression.
    trans run:
      guard:
        hasEvent(ein); // ein = e1,e2
      set:
        instore.addEvent(ein);

        // If an input event expression is completed then store it.
        instore.complete() => intimes.append(t);
    // Send events from the output expression anytime
    trans run:
      act:
        sendEvent(outstore.nextEvent());
      set:
        outstore.addEvent(outstore.nextEvent());
        outstore.complete() =>
        {
          // Add unsat input event expressions to the window
          foreach (u,intimes)
          {
            greaterUB(I,t-u) =>
            {
              win.append(u);
              intimes.remove(u);
            }
          }

          // All input event expressions within I are sat.
          foreach (u,intimes)
          {
            matchBounds(I,t-u) =>
            {
              win.append(u,true);
              intimes.remove(u);
            }
          }
        }
    // Sometimes we have to send output events ...
    trans sendnow:
      guard:
        // ... if there would be unsatisfied inputs.
        // win.wouldBeUnsat(geqUBCount(intimes,I,t));

  sendnow:
    urgent;
```

```
    trans sendnow:
      guard:
        // repeat the transition while there are unfulfilled sequences.
        win.wouldBeUnsat(geqUBCount(intimes,I,t));
      act:
        sendEvent(outstore.nextEvent());
      set:
        outstore.addEvent(outstore.nextEvent());
        outstore.complete() =>
        {
          foreach (u,intimes)
          {
            greaterUB(I,t-u) =>
            {
              win.append(u);
              intimes.remove(u);
            }
          }

          foreach (u,intimes)
          {
            matchBounds(I,t-u) =>
            {
              win.append(u,true);
              intimes.remove(u);
            }
          }
        }
    trans run:
      // otherwise go back to "main-loop"
      guard:
        !win.wouldBeUnsat(geqUBCount(intimes,I,t));
}
```

6.9.2 Monitor

The monitor is simpler than the generator, because it does not need to ensure satisfaction. Hence, the respective state is not needed. On the other hand, we need to track unsatisfied input expressions.

```
typedef event ... Port1;
typedef event ... Port2;
typedef event ... Port3;
typedef event ... Port4;

automaton ReactionMon {
  input:
    Port1 e1;
    Port2 e2;
    Port3 e3;
    Port4 e4;
    clock c;
  private:
    timer<c> t = 0;
    EventSet instore;
    multihash<real,bool> intimes;
    EventSequence outstore;
    Window<K,N> win;
    bool winfailed = false;
    bool oncefail = false;   // needed only if once keyword exists

  init:
    // init location is used for initialization only
    urgent;

    trans run:
      set:
        instore.addToExpr(e1);
        instore.addToExpr(e2);
        outstore.addToExpr(e3);
        outstore.addToExpr(e4);

  run:
    !winfailed;
    !oncefail;

    // We store all input events
    // This transition is duplicated for each event ein in the input expression.
    trans run:
      guard:
        hasEvent(ein); // ein = e1,e2
      set:
        instore.addEvent(ein);

        // If an input event expression is completed then store it,
        // and update sliding window.
        instore.complete() =>
        {
          // add event expression as unsat.
          intimes.insert(t,false);

          // what is older than I goes into window.
          foreach (u,intimes)
          {
            greaterUB(I,t-u) =>
            {
              foreach (v,intimes.values(u))
              {
                win.append(u,v);
                // window that is failed inbetween remains failed
```

```
                win.isFailed(t,I) => winfailed = true;
              }
              intimes.remove(u);
            }
          }
        }
    // Check for output events
    // This transition is duplicated for each event eout in the output expression.
    trans run:
      guard:
        hasEvent(eout); // eout = e3,e4
      set:
        outstore.addEvent(eout);

        outstore.complete() =>
        {
          // Set all inputs matching I to be satisfied
          foreach(u,intimes)
          {
            matchBounds(I,t-u) =>
            {
              list<bool> values = intimes.values(u);
              intimes.remove(u);

              foreach (v,values)
              {
                // set expression as satisfied.
                intimes.insert(u,true);
              }

              // This is for once only
              (values.count() > 1) => oncefail = true;
            }
          }
        }
    // Missing output violates the pattern
    trans bad:
      guard:
       winfailed;
    // This is for patterns with ’once’ only.
    // Too much matches violate the pattern.
    trans bad:
      guard:
        oncefail;

  bad:
    // pattern is violated
}
```

6.10 Age Pattern

The age pattern provides the backward view of time spans between event expressions.

Age :: **whenever** *EventExpr* **occurs then** *EventExpr* **has occurred within**
Interval [**once**]? [*Number* **out of** *Number* **times**]? **.**

We define the automata by the example pattern instance '**whenever (e3,e4) occurs then {e1,e2} has occurred within I once, K out of N times**'. The following automata contain comments for the statements that have to be removed if the optional once is omitted. If the restriction `K out of N times` is omitted then we assume `K = N = 1`.

6.10.1 Generator

Restrictions The generator of the age pattern can in general not enforce once semantics. Hence it is not allowed for generators of the pattern, and the example pattern reduces to '**whenever (e3,e4) occurs then {e1,e2} has occurred within I, K out of N times**'.

```
typedef event ... Port1;
typedef event ... Port2;
typedef event ... Port3;
typedef event ... Port4;

automaton AgeGen {
  input:
    Port1 e1;
    Port2 e2;
    clock c;
  output:
    Port3 e3;
    Port4 e4;
  private:
    timer<c> t = 0;
    EventSet instore;
    list<real> intimes;
    EventSequence outstore;
    Window<K,N> win;

  init:
    // init location is used for initialization only
    urgent;

    trans run:
      set:
        instore.addToExpr(e1);
        instore.addToExpr(e2);
        outstore.addToExpr(e3);
        outstore.addToExpr(e4);

  run:
    // We store all input events
    // This transition is duplicated for each event ein in the input expression.
    trans run:
      guard:
        hasEvent(ein); // ein = e1,e2
      set:
        instore.addEvent(ein);

        // If an input event expression is completed then store it.
        instore.complete() => intimes.append(t);

        // Cleanup old input expressions
        removeGreaterBounds(intimes,I,t);
    // We may send an event of the output expression as long as we
    // do not violate the sliding window.
    trans run:
      guard:
        hasMatchingBound(intimes,I,t)
         || !win.wouldBeUnsat()
         || (outstore.remainingEvents() > 1);
      act:
        sendEvent(outstore.nextEvent());
      set:
        outstore.addEvent(outstore.value(key).nextEvent());
        outstore.complete() =>
        {
          win.append(t);

          hasMatchingBound(intimes,I,t) =>
          {
            win.setSat(t);

            // This is for "once", but cannot enforce once,
            // because the function may remove multiple matching
            // input expressions.
            removeMatchingBound(intimes,I,t);
          }
        }
}
```

6.10.2 Monitor

The main structure of the monitor of the age pattern is equivalent to the one for reactions. However, we need to keep track of time points of completed input expressions.

```
  typedef event ... Port1;
  typedef event ... Port2;
  typedef event ... Port3;
  typedef event ... Port4;
```

```
automaton AgeMon {
  input:
    Port1 e1;
    Port2 e2;
    Port3 e3;
    Port4 e4;
    clock c;
  private:
    timer<c> t = 0;
    EventSet instore;
    list<real> intimes;
    EventSequence outstore;
    Window<K,N> win;
    bool oncefail = false;

  init:
    // init location is used for initialization only
    urgent;

    trans run:
      set:
        instore.addToExpr(e1);
        instore.addToExpr(e2);
        outstore.addToExpr(e3);
        outstore.addToExpr(e4);

  run:
    !win.isFailed(t,I);
    !oncefail;

    // We store all input events
    // This transition is duplicated for each event ein in the input expression.
    trans run:
      guard:
        hasEvent(ein); // ein = e1,e2
      set:
        instore.addEvent(ein);

        // If an input event expression is completed then store it.
        instore.complete() => intimes.append(t);

        // Cleanup old input expressions
        removeGreaterBounds(intimes,I,t);
    // Check for output events
    // This transition is duplicated for each event eout in the output expression.
    trans run:
      guard:
        hasEvent(eout); // eout = e3,e4
      act:
        outstore.addEvent(eout);

        outstore.complete() =>
        {
          win.append(t);

          hasMatchingBound(intimes,I,t) =>
          {
            win.setSat(t);

            // This is for once only
            matchingBoundCount(intimes,I,t) > 1 => oncefail = true;
          }
        }
    // Missing output violates the pattern
    trans bad:
      guard:
        win.isFailed(t,I);
    // Too much output violates the pattern, too, with 'once' only.
    trans bad:
      guard:
        oncefail;

  bad:
    // pattern is violated
}
```

6.11 Causal Reaction Pattern

The causal reaction pattern checks for reactions on causally related events only. In order to keep definition of this pattern simple, it relates one input event stream with one output event stream:

CausalReaction :: **`Reaction(`** *EventSpec* ',' *EventSpec* **`)`** **`within`** *Interval* **.**

Multiple event streams can be entangled by the definition of multiple causal reaction patterns. It is important to keep in mind that the pattern requires the definition of a causal relation function. If no such definition exists in the specification, we assume the default definition `[>(e1,e2)=ID`. Note that relating multiple input event streams to a single output event stream may impose problems when the ids of incoming input events are not consistent. The definition of more expressive causal relation function which can deal with such situations is subject of ongoing work.

We define the automata by the example pattern instance '**`Reaction(e1,e2) within I`**'.

6.11.1 Generator

Restrictions Generators for the pattern can be applied only for intervals which are right-closed, i.e., for intervals of the form `(L,U]` or `[L,U]`. Moreover, generators cannot be used in conjunction with the `LIFO` causal relation function. This is because the generator cannot be forced to follow the pattern in any case as this would require prophetic gifts. Suppose the pattern `Reaction(e1,e2) within [8,10]ms`. Further suppose the LIFO stores contains the elements `[(e1,42),(e1,44)]` at time point

50, where the next element has time stamp 45. The latest time point at which this element could be sent is 52 because otherwise the element with time stamp 42 would miss its deadline. However, no other input event must arrive before that time because otherwise the pattern is violated. Reacting on such event is also not possible because this still violates the LIFO semantics.

For FIFO and ID relation functions this is no problem, and the generator works this way by sending output events if the corresponding stored events would otherwise violate their deadline.

```
typedef event id ... Port1;
typedef event id ... Port2;

automaton CausalReactionGen {
  input:
    Port1 e1;
    clock c;
  output:
    Port2 e2;
  private:
    timer<c> t = 0;
    STORE<Port1> intimes;  // Select the correct store type

  init:
    !intimes.hasEqualUB(I,t);

    trans init:
      guard:
        hasEvent(e1);
      set:
        // Store event.
        intimes.insert(t,e1);
    trans init:
      guard:
        intimes.hasMatch(I,t);
      act:
        sendEvent(e2,intimes.nextMatchEvent(I,t).id);
      set:
        // Remove matching event pair.
        intimes.removeMatch(I,t,intimes.nextMatchEvent(I,t).id);
    trans sendnow:
      guard:
        intimes.hasEqualUB(I,t);

  sendnow:
    intimes.hasEqualUB(I,t);

    trans sendnow:
      guard:
        intimes.hasMatch(I,t);
      act:
        sendEvent(e2,intimes.nextMatchEvent(I,t).id);
        // Remove matching event pair.
        intimes.removeMatch(I,t,intimes.nextMatchEvent(I,t).id);
    trans init:
      guard:
        !intimes.hasEqualUB(I,t);
}
```

6.11.2 Monitor

The monitor of the causal reaction pattern is very simple. It detects violation by checking for violations of the interval for all stored input events. Matching output events are recognized and the store is updated accordingly. Non-matching events are ignored.

```
typedef event id ... Port1;
typedef event id ... Port2;

automaton CausalReactionMon {
  input:
    Port1 e1;
    Port2 e2;
    clock c;
  private:
    timer<c> t = 0;
    STORE<Port1> intimes;  // Select the correct store type

  init:
    !intimes.hasGreaterUB(I,t);

    trans init:
      guard:
        hasEvent(e1);
      set:
        // Store event.
        intimes.insert(t,e1);
    trans init:
      guard:
        hasEvent(e2);
        intimes.hasMatch(I,t,e2.id);
      set:
        // Remove matching event pairs.
        intimes.removeMatch(I,t,e2.id);
    trans bad:
      guard:
        intimes.hasGreaterUB(I,t);

  bad:
    // pattern is violated
}
```

6.12 Causal Age Pattern

The causal age pattern checks for ages on causally related events only. In order to keep definition of this pattern simple, it relates one input event stream with one output event stream:

CausalAge :: **Age(** *EventSpec* **','** *EventSpec* **) within** *Interval* **.**

As for the causal reaction pattern, multiple event streams can be entangled by the definition of multiple causal reaction patterns. It is important to keep in mind that the pattern requires the definition of a causal relation function. If no such definition exists in the specification, we assume the default definition `<](e1,e2)=ID(e1)`. Note that relating multiple input event streams to a single output event stream may impose problems when the ids of incoming input events are not consistent. The definition of more expressive causal relation function which can deal with such situations is subject of ongoing work.

We define the automata by the example pattern instance '**Age(e1,e2) within I**'.

6.12.1 Generator

Restrictions Generators for the pattern can be applied only for intervals which are right-closed, i.e., for intervals of the form `(L,U]` or `[L,U]`.

In contrast to the causal reaction pattern, the generator of the causal age pattern can be applied together with the `LIFO` relation function. This is because input event which are "to old" can simply be ignored.

The ability to simply ignore events that do not match the deadline anymore makes the generator very simple.

```
typedef event id ... Port1;
typedef event id ... Port2;

automaton CausalAgeGen {
  input:
    Port1 e1;
    clock c;
  output:
    Port2 e2;
  private:
    timer<c> t = 0;
    STORE<Port1> intimes;  // Select the correct store type

  init:
    trans init:
      guard:
        hasEvent(e1);
      set:
        // Store event.
        intimes.insert(t,e1);
        // Remove elements which are too old
        intimes.removeGreaterUB(I,t);
    trans init:
      guard:
        intimes.hasMatch(I,t);
      act:
        sendEvent(e2,intimes.nextMatchEvent(I,t).id);
}
```

6.12.2 Monitor

The monitor is even simpler than the one for causal reactions. It detects violation by checking for non-matching events. Matching output events are recognized and the store is updated accordingly.

```
typedef event id ... Port1;
typedef event id ... Port2;

automaton CausalAgeMon {
  input:
    Port1 e1;
    Port2 e2;
    clock c;
  private:
    timer<c> t = 0;
    STORE<Port1> intimes;  // Select the correct store type

  init:
    trans init:
      guard:
        hasEvent(e1);
      set:
        // Store event.
        intimes.insert(t,e1);
        // Remove elements which are too old
        intimes.removeGreaterUB(I,t);
    trans init:
      guard:
        hasEvent(e2);
        intimes.hasMatch(I,t,e2.id);
    trans bad:
      guard:
        hasEvent(e2);
        !intimes.hasMatch(I,t,e2.id);

  bad:
    // pattern is violated
}
```

7 Design Rules

The MULTIC-Tooling prototype allows for modeling systems in terms of SysML. In order to successfully perform analysis on such models, they must adhere to a number of “design rules”, which are presented in this chapter. Based on the results of the MULTIC project [3], the particularities of the modeling of components, inputs, outputs, and channels, as well as the specification of timing requirements are discussed. They serve as a basis for automated model checks.

All guidelines for modeling systems presented in this chapter are explained in detail in the “Papyrus SysML Modeling” video tutorial, which comes with this report. It is available in both English and German. Through the explicit annotation of the applied design rules, the underlying principles can be easily understood and learned step by step using a practical example. For a better understanding, the relevant timestamps of the video will be referred in the further course of this chapter.

7.1 Best Practices

In order to increase the comprehensibility of the models used and to be able to provide assistance with concrete problems, the best practice rules listed below should be taken into account when modeling architectures. Since their compliance is not checked by the presented tool prototype, the analysis of differently structured models is also possible in the current stage of development.

While rules BR-1 to BR-3 are fully considered in the video tutorial, their use is always implicit. If the modeling of the system under consideration is done according to the demonstrated principles, the resulting architecture conforms to the subsequent requirements automatically.

BP-1 *A model should contain exactly one block definition diagram.*

A block definition diagram (BDD) defines the hierarchical structure of the architecture under consideration. While each component is represented by a `Block`, hierarchical structures can be represented using `Aggregation` relationships. An `InterfaceBlock` defines a port type that can be used in the `ProxyPort` instances of the inputs and outputs. The definition of a `FlowProperty` element in the interface block is used to differentiate between input and output ports.

BP-2 *A model should contain exactly one internal block diagram for each decomposition of a block.*

An internal block diagram (IBD) describes the interfaces between the components involved. The previously modeled `ProxyPorts` can be connected to each other using `BindingConnector` relationships.

BP-3 *A model should have exactly one requirement diagram.*

A requirement diagram (RD) specifies timing requirements for the existing blocks. A `Requirement` can be assigned to a `Block` using a `satisfy` relation and contains always exactly one contract.

7.2 Modeling of Components

DR-1 *A model must contain exactly one context block.*

A context block is a `Block` that has no higher-level blocks and no input or output ports. The explicit modeling of the system context serves as an entry point for the automated analysis.

DR-2 *A model must have exactly one system block.*

A system block is a `Block` that is connected to the context block by exactly one `Aggregation` and has at least one output port. Within the context of this project, a virtual integration test is carried out for exactly one system. A future extension for modeling multiple systems can be implemented.

DR-3 *A model can have any number of subsystem blocks.*

A subsystem block is a `Block` that is connected to the system block or a subsystem block subordinate to it by exactly one `Aggregation`. According to this design rule, the functionality of the system can be divided into as many subsystem blocks as required.

DR-4 *A model must have exactly one interface block called 'Event'.*

An interface block is an `InterfaceBlock` that has exactly one `FlowProperty` element called *EventFlow* and the attribute *Direction* with the value *out*. The explicitly modeled interface block forms the basis for the typing of the ports included. To facilitate the integration of functional aspects, the implementation of user-defined functions using additional interface blocks can be simplified.

Figure 7.1 exemplifies the application of the design rules DR-1 to DR-4. The illustrated block definition diagram of our `tinySysMLModel` example contains exactly one context and one system block, two subsystem blocks and one interface block (see video tutorial, 1:07 to 7:20 min.).

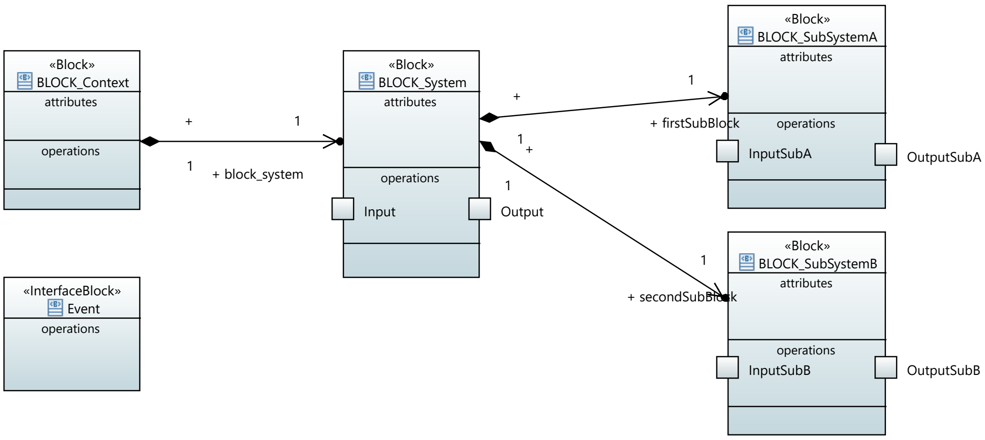


Figure 7.1: Block Definition Diagram

7.3 Modeling of Inputs and Outputs

DR-5 *Each input and output must be represented by a `ProxyPort`.*

While `FullPorts` provide their own behavior, `ProxyPorts` expose some of the behavior of the owning block [9]. The `ProxyPorts` therefore correspond exactly to our idea of the semantics of ports and are used exclusively in the context of this project [3, p. 140]. Each port must possess the attribute *Type* with the value *Event*. Moreover, the attribute *isConjugated* must be set to *true* for each input port, and to *false* for each output port. The ability to clearly distinguish between inputs and outputs is an important prerequisite for generating the simulation model.

The correct modeling of inputs and outputs can be seen from the internal block diagram of `BLOCK_System` depicted in Figure 7.2 (see video tutorial, 8:16 to 13:06 min.). Each input and output is represented by a `ProxyPort` instance. A positive value of the attribute *isConjugated* is indicated by the presence of the prefix ~ preceding the type name *Event*.

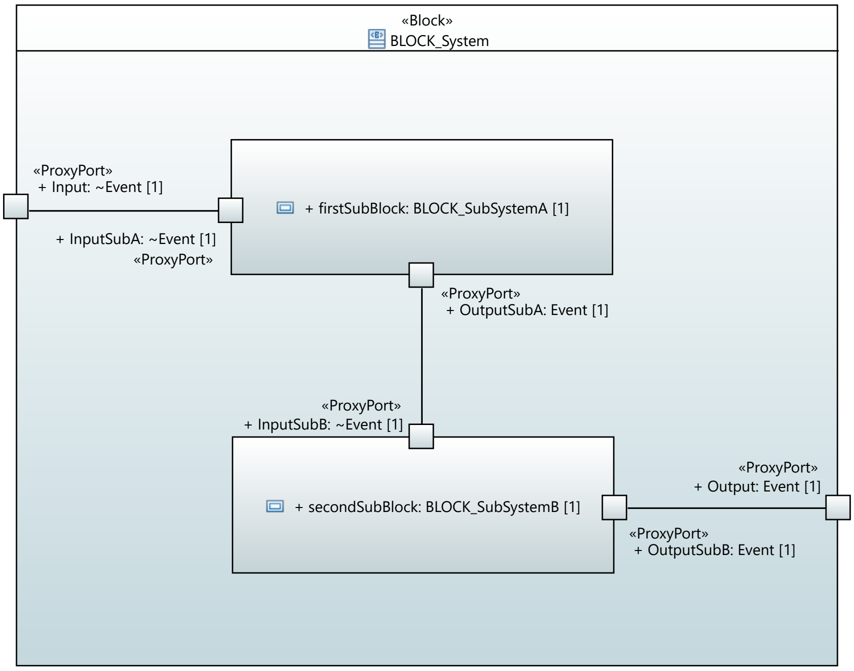


Figure 7.2: Internal Block Diagram of `BLOCK_System`

7.4 Modeling of Channels

As explained in Chapter 2, channels (or also connectors) are used to connect the inputs and outputs existing in the model. Within this project `BindingConnectors` are used for their representation. The rules listed below ensure that the system can be analyzed without any restrictions.

DR-6 *Each input port of a higher-level block must be connected to at least one input port of a lower-level block using a `BindingConnector`.*

DR-7 *Each output port of a higher-level block must be connected to at least one output port of a lower-level block using a `BindingConnector`.*

DR-8 *Each remaining free input port must be connected to at least one output port of another block on its hierarchy level using a `BindingConnector`.*

DR-9 *Each remaining free output port must be connected to at least one input port of another block on its hierarchical level using a `BindingConnector`.*

The application of rules DR-6 to DR-9 can also be seen in Figure 7.2. While the input port of `BLOCK_System` is connected to the input port of `BLOCK_SubSystemA`, another `BindingConnector` connects its output to the input of `BLOCK_SubSystemB`. The output port of `BLOCK_SubSystemB` is in turn connected to the output port of `BLOCK_System` (see video tutorial, 8:16 to 13:06 min.).

7.5 Specification of Timing Requirements

DR-10 *A model must contain exactly one requirement block for each system and subsystem block.*

A requirement block is a `Requirement` that has a name in the *name* attribute, an ID in the *id* attribute, and a timing specification in the *text* attribute. The listed attributes are required for the clear identification of the contracts and the output of suitable error messages.

DR-11 *Each system and subsystem block must be connected to its requirement block through exactly one `satisfy` relation.*

Each system and subsystem block must be connected to its requirement block by exactly one `satisfy` relation. For the sake of traceability, complex requirement relationships are omitted in this project.

DR-12 *Each timing specification in the 'text' attribute of a requirement block can have an assumption and must have a guarantee.*

The assumption must be identified by the prefix *A:*, the guarantee by the prefix *G:*. For multi-line specifications, each identifier is valid until the next occurrence of *A:* or *G:*. A non-existent assumption is interpreted as *A: true* and is therefore always satisfied. As a result of the these rules, any pattern used in a timing specification can be uniquely assigned to its assumption or guarantee clause.

Finally, two further rules are needed to ensure the consistency of the model and its structure with the contents of the annotated contracts and the principles of the methodology used.

DR-13 *Each assumption of a block may only contain references to its input ports.*

DR-14 *Each guarantee may contain references to both input and output ports of the corresponding block.*

Figure 7.3 shows the requirement diagram of our `tinySysMLModel` example. According to the design rules, each block has exactly one requirement block that is connected by a satisfy link and contains a timing specification that meets the specified conditions (see video tutorial, 13:07 to 16:08 min.).

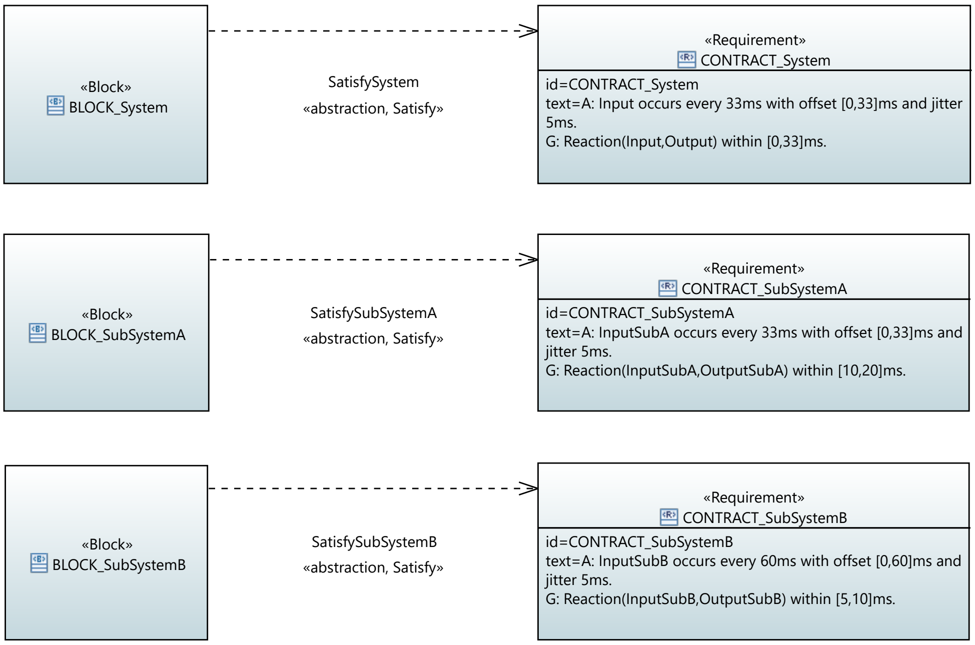
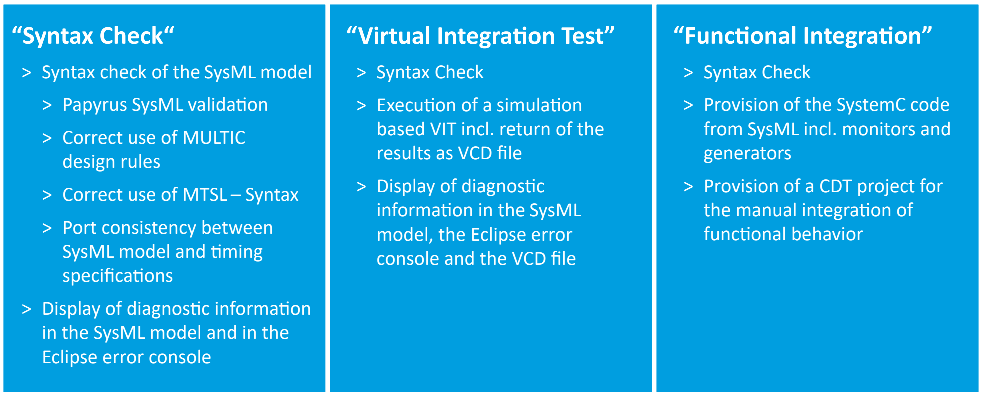

«Block»
BLOCK_System
SatisfySystem
«abstraction, Satisfy»
«Requirement»
CONTRACT_System
id=CONTRACT_System
text=A: Input occurs every 33ms with offset [0,33]ms and jitter 5ms.
G: Reaction(Input,Output) within [0,33]ms.
«Block»
BLOCK_SubSystemA
SatisfySubSystemA
«abstraction, Satisfy»
«Requirement»
CONTRACT_SubSystemA
id=CONTRACT_SubSystemA
text=A: InputSubA occurs every 33ms with offset [0,33]ms and jitter 5ms.
G: Reaction(InputSubA,OutputSubA) within [10,20]ms.
«Block»
BLOCK_SubSystemB
SatisfySubSystemB
«abstraction, Satisfy»
«Requirement»
CONTRACT_SubSystemB
id=CONTRACT_SubSystemB
text=A: InputSubB occurs every 60ms with offset [0,60]ms and jitter 5ms.
G: Reaction(InputSubB,OutputSubB) within [5,10]ms.

Figure 7.3: Requirement Diagram

8 Tool Flow and Architecture

The realization of the tool prototype is based on three identified uses cases, which are illustrated in Figure 8.1. The *Syntax Check* use case (UC1) checks the SysML model created by the user on the one hand for correctness and on the other hand for the correct application of the MULTIC methodology and the presented design rules (see Chapter 7). At the end, diagnostic information is displayed to the user. Based on the SysML model, a *Virtual Integration Test* is performed in the second use case (UC2). The SysML model is translated to a SystemC simulation model and executed. The simulation results are made available as a VCD file including diagnostic information. The third use case *Functional Integration* (UC3) enables the user to integrate his own function code into the simulation model. The simulation model generated from the SysML model is provided as an Eclipse CDT project.



“Syntax Check“

- Syntax check of the SysML model
  - Papyrus SysML validation
  - Correct use of MULTIC design rules
  - Correct use of MTSL – Syntax
  - Port consistency between SysML model and timing specifications
- Display of diagnostic information in the SysML model and in the Eclipse error console

“Virtual Integration Test”

- Syntax Check
- Execution of a simulation based VIT incl. return of the results as VCD file
- Display of diagnostic information in the SysML model, the Eclipse error console and the VCD file

“Functional Integration”

- Syntax Check
- Provision of the SystemC code from SysML incl. monitors and generators
- Provision of a CDT project for the manual integration of functional behavior

Figure 8.1: Overview of the Identified Use Cases

Based on the identified use cases, three tool flows were instantiated and transferred into suitable system architectures. In the further course of this chapter, the use cases and their realization are examined in more detail. Note that only the conceptual design of the tool is described. More detailed information on the features of the implementation can be found in the corresponding Doxygen or JavaDoc documentation, which comes with this report.

8.1 Syntax Check

As for the two subsequent use cases, the following subsections firstly explain the tool flow of the *Syntax Check* and then the resulting architecture.

8.1.1 Tool Flow

The tool flow of the *Syntax Check* is depicted in Figure 8.2. After an initialization phase in which system parameters are set, the model to be analyzed is read in. The individual analyses are then performed step by step. Firstly, the SysML model validation is performed with Papyrus. If the model conforms to the SysML standard, the correct application of the MULTIC Design Rules is checked in the next step. If this step has also been successfully completed, the MULTIC Tooling Specification Language (see Chapter 7) application is checked for syntax errors and port consistency. The SystemC simulation model is generated from the SysML model, then compiled and executed. The individual phases are triggered and synchronized by an orchestration. If a phase could not be successfully completed, corresponding diagnostic information is returned to the user.

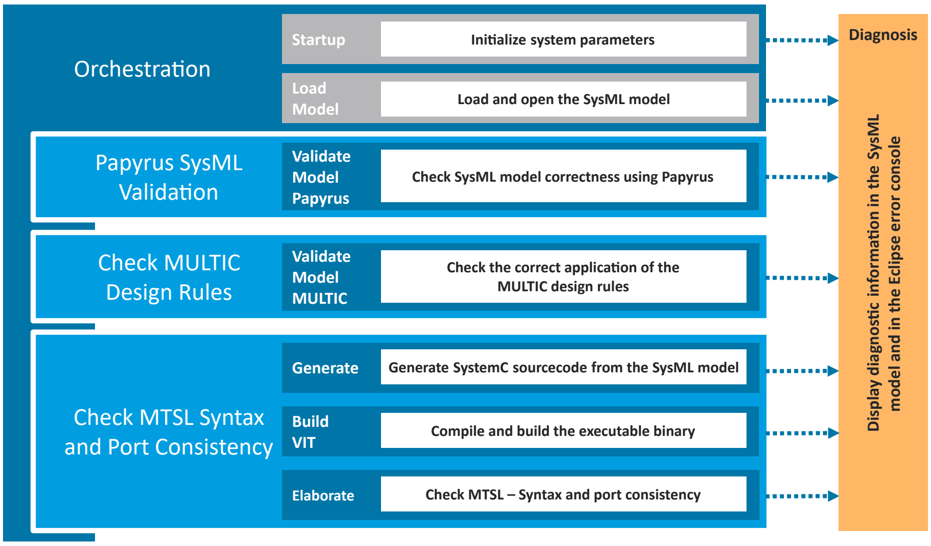

Orchestration
Startup
Initialize system parameters
Load Model
Load and open the SysML model
Papyrus SysML Validation
Validate Model Papyrus
Check SysML model correctness using Papyrus
Check MULTIC Design Rules
Validate Model MULTIC
Check the correct application of the MULTIC design rules
Check MTSL Syntax and Port Consistency
Generate
Generate SystemC sourcecode from the SysML model
Build VIT
Compile and build the executable binary
Elaborate
Check MTSL – Syntax and port consistency
Diagnosis
Display diagnostic information in the SysML model and in the Eclipse error console

Figure 8.2: Tool Flow of *Syntax Check* (UC1)

8.1.2 Architecture

In the next step, the tool flow described before was transferred to the architecture depicted in Figure 8.3, consisting of the `Syntax Check` plugin, the `Diagnosis` module and some external tools. While the `Papyrus Model Editor` comes with a complete modeling environment for creating SysML diagrams, the `Papyrus Model Validation` provides methods for checking consistency. The `Eclipse User Interface` is also included for user interaction. In addition, the `UI Interaction` plugin is used to decouple the external user interface and the tool core developed within the context of this project.

The individual steps of the `Syntax Check` are implemented as methods that are started by the `Orchestration` method (black arrow), which controls and monitors the entire analysis process. Each individual method provides the `Orchestration` with return values (blue arrow) and diagnostic information to the `Diagnosis` module in the event of an error (dotted arrow).

Orchestration

The `Orchestration` method is the heart of every use case plugin. As already described in Section 8.1.1, the execution starts with the initialization of the other plugins as well as the required methods and parameters. Subsequently, an interaction with the `Eclipse User Interface` via the `UI Interaction` plugin opens a dialog in which the user can select the SysML model to be analyzed.

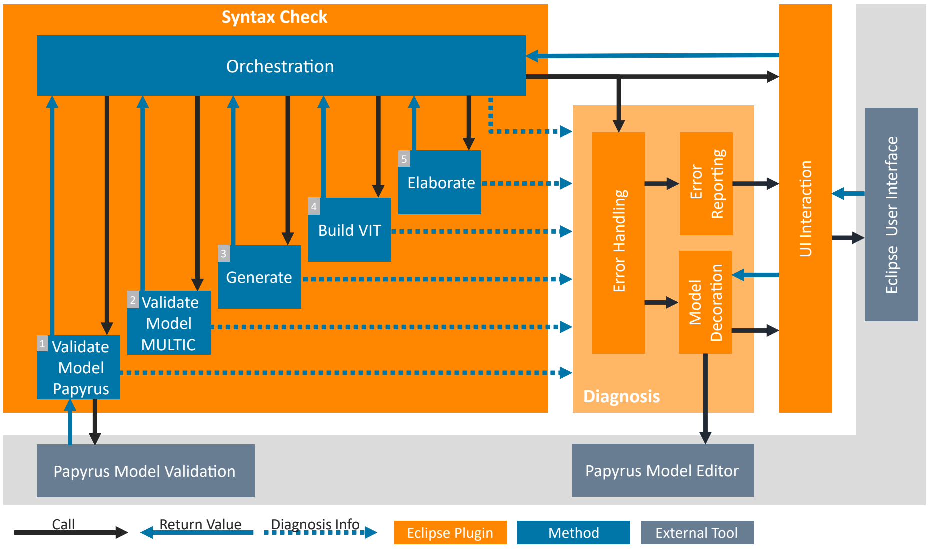


Figure 8.3: System Architecture of *Syntax Check* (UC1)

During the individual process steps, the `Orchestration` method also informs the user about the current progress. Following the step-by-step call of the process step methods, it also displays a summary and triggers the termination of the superordinate `Syntax Check` plugin.

Validate Model Papyrus

In order to check the correctness of the loaded model and its consistency with the SysML specification, the `Validate Model Papyrus` method uses a number of library functions of the `Papyrus Model Validation`. By calling an internal validation method, the check is finally executed. The return value contains the information whether errors occurred during validation. Due to the profound integration into the underlying Eclipse instance, the errors are automatically displayed to the user. The occurrence of an EXISTING_ERRORS error indicates a failure of this initial model check (see Section 10.1.3).

Validate Model MULTIC

After executing the `Validate Model Papyrus` step, the compliance with the MULTIC Design Rules presented in Chapter 7 is tested. The corresponding `Validate Model MULTIC` method is called by the `Orchestration` method and returns control after successfully completing its task.

Generate

Assuming that both methods are successfully completed to ensure correctness and consistency of the SysML model used, the automatic translation of the architectural description into a corresponding SystemC simulation model can be performed. The `Generate` method runs through the entire model hierarchy and generates a number of C++ files, whose contents are explained later in Chapter 9.

Build VIT

To construct an executable simulation model from the previously generated files, all source files must be compiled and linked against the TSLsim library. The `Build VIT` method realizes this behavior by calling an external script, which is integrated in the tool and fully automates the build process.

Elaborate

Preparing the execution of a simulation run of the previously generated SystemC simulation model, the so-called *elaboration phase* is an important step. In the course of this SystemC-specific preparation phase, the module hierarchy is instantiated, i.e., a network of processes and channels representing the simulation model is built at runtime. This task is performed by the `Elaborate` method.

In order to handle all SystemC errors that occur during elaboration, the output streams of the executable are passed back into the tool environment and converted to the error types listed in Section 10.1.7 using a custom error message parser. Without any confusion of the user, the purified error messages are forwarded to the `Diagnosis` module and handled with the established approach.

Diagnosis

The `Diagnosis` module itself is realized by three plugins. The `Error Handling` plugin collects all diagnostic information and forwards them to the latter two plugins. The `Error Reporting` ensures that the user receives diagnostic information in the status console, which is part of the `Eclipse User Interface`. In addition, the `Model Decoration` plugin annotates the affected SysML model elements within the `Papyrus Model Editor` so that the user also receives graphical feedback.

8.2 Virtual Integration Test

For reasons of maintainability, tool flow and architecture of the *Virtual Integration Test* are very similar to those of the *Syntax Check*. This chapter therefore gives priority to the differences.

8.2.1 Tool Flow

After all necessary parameters have been initialized, the SysML model to be analyzed is loaded. In the further course, the *Syntax Check* is performed using the `Syntax Check` plugin (see Section 8.1). Following a selection of simulation time and resolution by the user, the simulation-based *Virtual Integration Test* is performed. At the end, the user receives a VCD file with simulation and diagnostic information. In the event of an error, the corresponding diagnostic information is also displayed to the user. The resulting tool flow of use case *Virtual Integration Test* is illustrated in Figure 8.4.

8.2.2 Architecture

The *Virtual Integration Test* is also implemented as an Eclipse plugin with methods for all individual steps, whose structure can be seen in Figure 8.5. When executing the *Syntax Check*, however, the `Syntax Check` plugin is used. While the same concepts are applied for handling errors (see Section 10.2) as in the previous use case, the external tool `GTK Wave` is used besides the already known `Papyrus Model Editor` and the `Eclipse User Interface` for displaying the simulation results.

Following the examination of the SysML model, the generation of the C++ source files as well as the compilation and elaboration of the simulation model, the `Orchestration` method opens a dialog using the `UI Interaction` plugin that allows the user to select the simulation duration and resolution.

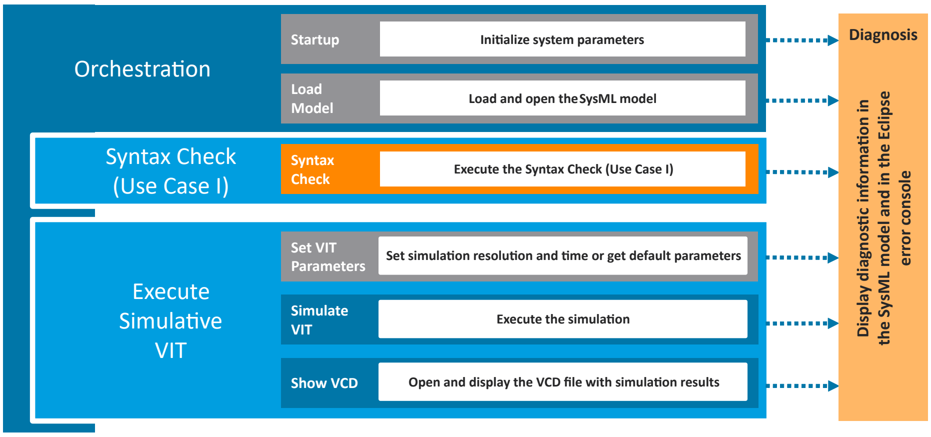

Orchestration
Startup
Initialize system parameters
Load Model
Load and open the SysML model
Syntax Check (Use Case I)
Syntax Check
Execute the Syntax Check (Use Case I)
Execute Simulative VIT
Set VIT Parameters
Set simulation resolution and time or get default parameters
Simulate VIT
Execute the simulation
Show VCD
Open and display the VCD file with simulation results
Diagnosis
Display diagnostic information in the SysML model and in the Eclipse error console

Figure 8.4: Tool Flow of *Virtual Integration Test* (UC2)

Simulate VIT

Based on the previously generated simulation model and the knowledge of the desired simulation parameters, the virtual integration test can be performed. For this purpose, the external executable from the `Elaborate` method is used again and is invoked with the corresponding status flags. The output streams are redirected in the same way as for the process step mentioned before. Similarly, arising SystemC errors are automatically forwarded to the `Diagnosis` module.

Show VCD

The results of the virtual integration test are presented in the external tool `GTK Wave` following a call by the `Show VCD` method. In addition to the value sequences of all ports, the states of the observer automatons as well as textual annotations of a carefully selected set of relevant diagnostics information are presented to the user. After closing the VCD viewer, the plugin terminates automatically.

8.3 Functional Integration

Again, tool flow and architecture of the *Functional Integration* are nearly identical to those of the previously considered *Syntax Check* and *Virtual Integration Test* use cases.

8.3.1 Tool Flow

After initialization, opening the model and performing the *Syntax Check*, the SystemC simulation model is generated from the SysML model and then imported into the Eclipse environment as a CDT project. Each step generates diagnostic information that is made available to the user.

8.3.2 Architecture

The *Functional Integration* is implemented in the same way as the *Virtual Integration Test*. Firstly, the `Syntax Check` plugin is called to check syntax and semantics. An Eclipse CDT project is then generated and finally imported into the `Eclipse CDT Editor` by an external call. Possible errors are

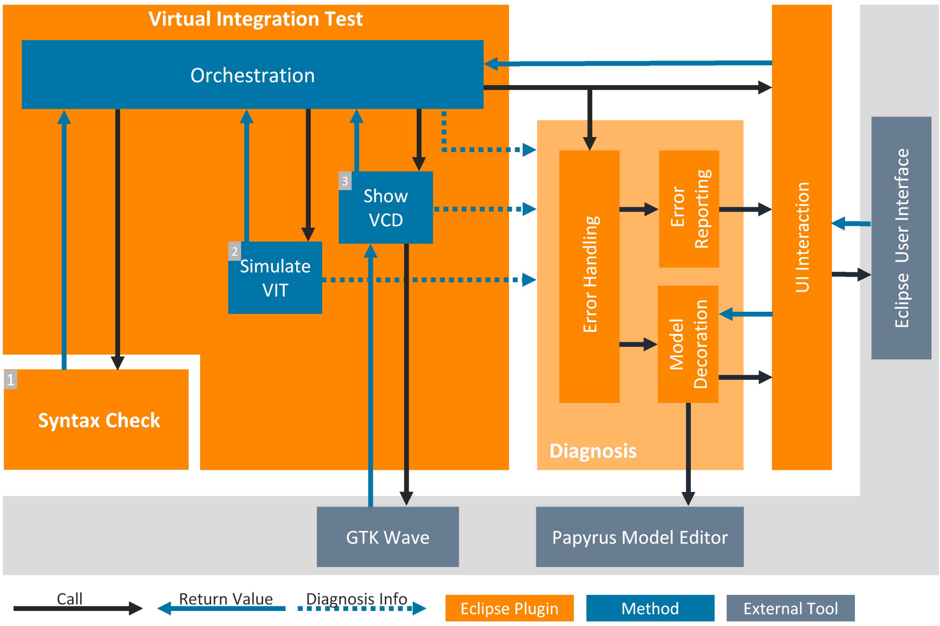


Figure 8.5: System Architecture of *Virtual Integration Test* (UC2)

again handled via the Diagnosis module (see Section 10.3). In addition to the Eclipse CDT Editor, also the Papyrus Model Editor and the Eclipse User Interface are involved.

Build Functional Integration

To build-up a fully functional Eclipse CDT project from the previously generated files, an additional .project file has to be created. For this purpose, the Build Functional Integration method calls another external script, which fully automates the generation process. The output stream is redirected to the tool environment, which handles possible errors using the Diagnosis module appropriately.

Import CDT

Once the CDT project has been created, it can be imported into the Eclipse CDT Editor. In addition to the source files generated during the Generate process step, the project also includes a set of predefined build scripts for executing a functionally enriched simulation model. Note that the correctness of the simulation can not be guaranteed for arbitrary modifications.

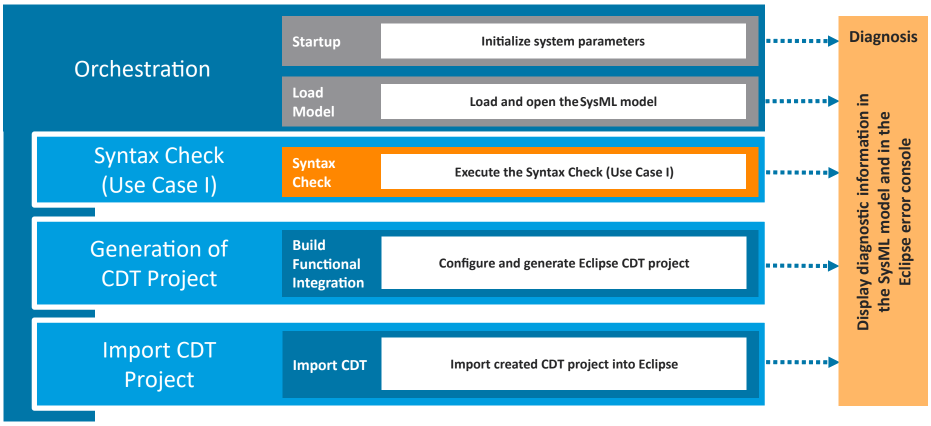

Orchestration
Startup
Initialize system parameters
Load Model
Load and open the SysML model
Syntax Check (Use Case I)
Syntax Check
Execute the Syntax Check (Use Case I)
Generation of CDT Project
Build Functional Integration
Configure and generate Eclipse CDT project
Import CDT Project
Import CDT
Import created CDT project into Eclipse
Diagnosis
Display diagnostic information in the SysML model and in the Eclipse error console

Figure 8.6: Tool Flow of *Functional Integration* (UC3)

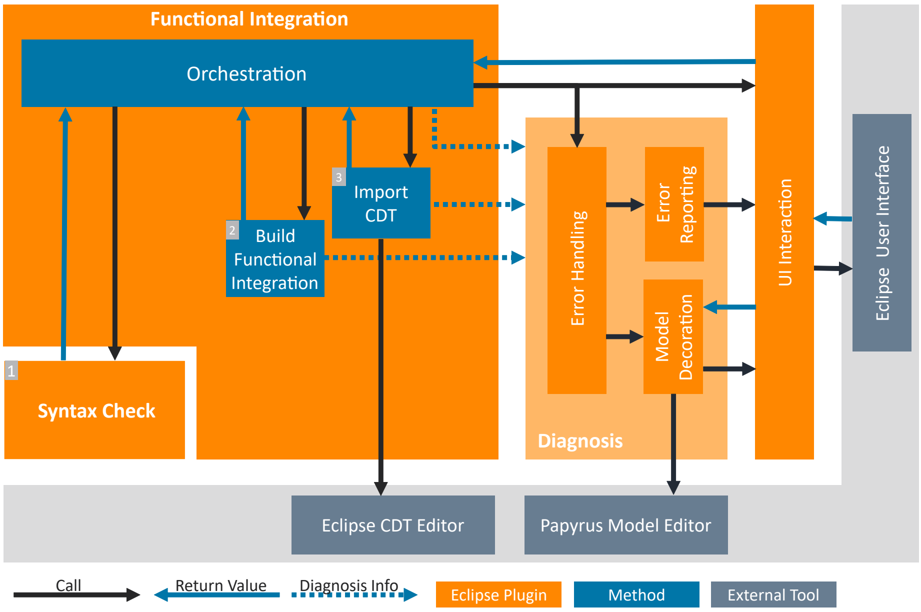


Figure 8.7: System Architecture of *Functional Integration* (UC3)

9 Simulation Backend

The simulation backend is the part of the MULTIC Tooling prototype which actually performs the simulative analysis of SysML models and their contracts. To this end, a `Generate` step (cf. Chapter 8) translates a specified SysML model into an executable piece of C++ code consisting of a representation of the model and its contracts, and the functions in order to perform the simulation. This mainly includes the SystemC library as well as the contract-compiler.

In order to simplify code generation and handling of simulation setup procedures, many SystemC related aspects are encapsulated into a library called `TSLsim`. This library provides basic blocks and components to execute a simulation-driven *Virtual Integration Test (VIT)* of the specified SysML model and its attached timing contracts. In other words, it serves as an implementation of basic blocks, which are instantiated in the generated code extracted from the SysML model. The generated code can be compiled and linked against this library.

This chapter aims at giving an overview of the functionality of this library. It is divided into three section. Section 9.1 discusses an example, where a simple SysML model has been generated into TSLsim code, and gives a brief overview on the involved concepts. Section 9.2 shows how user-specific implementation code can be added to the simulation. This instantiated what is called *Functional Integration* in Section 8.3. Finally, Section 9.3 discusses some details of implementation wrt. multiple generatorson single output ports.

9.1 Introduction Example

This example consists of a system block containing two subsystem blocks. The full source code of this example can be found at introduction.cpp in the examples folder. Each of the three blocks has a contract attached which defines the timing properties on its ports. In this example, each block has an `Input` and an `Output` port.

Hierarchy and Timing Behaviour in (Sub)System Blocks The first Subsystem Block can be defined as:

```
struct BLOCK_SubSystemA : tslsim::block_base
{
   tslsim::input_port InputSubA{ "InputSubA" };
   tslsim::output_port OutputSubA{ "OutputSubA" };

   BLOCK_SubSystemA(tslsim::module_name = "BLOCK_SubSystemA")
   : block_base("A: InputSubA occurs every 33ms with offset [0,33]ms and jitter 5ms.\n"
                "G: Reaction(InputSubA, OutputSubA) within [10,20]ms.")
   {
     this->instantiate_generator();
   }
   private:
};
```

The block BLOCK_SubsystemA contains two ports InputSubA and OutputSubA and a contract. The **input_port** and **output_port** are so-called observable ports and consist of a port and an internal singal named s_<PortName>. Internally, an input port observes the port values and forwards them to the signal. The output port observes values on the signal and forwards them to the port.

Since there is no subsystem defined, this block also implements the timing behaviour of the contract guarantee part using a generator based on SystemC processes. This is achieved using the call **this**->instantiate_generator(). It instructs the underlying base class to instantiate a generator for the guarantee part of the contract and bind the generator ports to the signals of the observable ports.

It is important to name the input/output ports correctly, because the port identifier in the contract string is used to search for the structural component in the current SystemC hierarchy level. In this example case, a port Input specified in the contract will be bound to the signal s_Input, and the lookup for the signal is performed in the current hierarchy level BLOCK_SubSystemA. Hence, the port will dynamically bind to the internal signal s_Input of the observable port Input.

Every block derived from **tslsim::block_base** also contains a monitor which checks if the assumption of the contract is valid and if the contract guarantee is implemented by the underlying timing behaviour. The monitor semantics implementation is based on Timed-Value Streams. Each input/output port therefore contains a timed-value stream where all observed values are written to. After parsing the contract, the corresponding monitor instances attach to the observers via timed-value streams by a name-based lookup.

In a similar manner, a second Block_SubSystemB can be defined.

```
struct BLOCK_SubSystemB : tslsim::block_base
{
   tslsim::input_port InputSubB{ "InputSubB" };
   tslsim::output_port OutputSubB{ "OutputSubB" };

   BLOCK_SubSystemB(tslsim::module_name = "Block_SubSystemB")
   : block_base("A: InputSubB occurs every 60ms with offset [0,60]ms and jitter 5ms.\n"
                "G: Reaction(InputSubB, OutputSubB) within [5,10]ms.")
   {
     this->instantiate_generator();
   }
   private:
};
```

The encapsulating BLOCK_System consisting of both subsystems can then be described as:

```
struct BLOCK_System : tslsim::block_base
{
   tslsim::input_port Input{ "Input" };
   tslsim::output_port Output{ "Output" };

   BLOCK_System(tslsim::module_name = "BLOCK_System")
   : block_base("A: Input occurs every 33ms with offset [0,33]ms and jitter 5ms.\n"
                "G: Reaction(Input,Output) within [0,33]ms.")
   {
    firstSubBlock.InputSubA(Input);
    firstSubBlock.OutputSubA(firstSubBlock_OutputSubA__secondSubBlock_InputSubB__);
    secondSubBlock.InputSubB(firstSubBlock_OutputSubA__secondSubBlock_InputSubB__);
    secondSubBlock.OutputSubB(Output);
   }

   private:
   BLOCK_SubSystemB secondSubBlock;
   BLOCK_SubSystemA firstSubBlock;
   tslsim::signal_type firstSubBlock_OutputSubA__secondSubBlock_InputSubB__;
};
```

Both subsystems are instantiated within BLOCK_System and an internal signal is used to wire the ports appropriately. In this case, the Input port of the first block is bound to the Input port of the system block, while both subsystems are interconnected with an internal signal. Finally, the Output port of the second subsystem is bound to the Output port of this system block. Here, the subsystems implement the timing behaviour, which is why no call to **this**->instantiate_generator() is performed. Nevertheless, the block contains a monitor and observes the timing properties on the Input and Output ports as defined in the attached contract.

Putting it All Together: The Context Block Finally, the system block is instantiated in the context block, which acts as the SystemC top-level module.

```
struct BLOCK_Context : tslsim::context_base
{
   BLOCK_Context(tslsim::module_name = "BLOCK_Context")
   : context_base("A: Input occurs every 33ms with offset [0,33]ms and jitter 5ms.\n"
                  "G: Reaction(Input,Output) within [0,33]ms.\n")
   {
       sys_.Input(s_Input);
       sys_.Output(s_Output);
   }

   private:
      tslsim::signal_type s_Input{ "s_Input" };
      tslsim::signal_type s_Output{ "s_Output" };
      BLOCK_System sys_;
};
```

The context block (derived from **tslsim::context_base**) differs from the **tslsim::block_base** due to the fact that all input ports of the underlying system instance need to be driven by generators derived from the specified contract. Therefore, the generated code should define a signal with the name s_<PortName> for each port PortName of the system block instantiated in the context block. The ports of the system block are then bound to the signals, such that they can drive the system instance.

The specified top-level contract is forwarded to the the base class for instantiating a generator which drives the input ports of the system block.

This is done by instantiating a generator for the assumption part of the specified top-level contract. Due to the consistent naming requirement, the generator can look up the signals and bind its ports to them.

Instantiating and Starting the Simulation The library provides an implementation of **sc_main** which – after initialising the context – calls the user's implementation of **tslsim_main** which in turn is responsible for instantiating the simulation components:

```
void tslsim_main()
{
   BLOCK_Context b;
   tslsim::start_simulation();
}
```

The SystemC timing resolution and the VCD backend can be customised via parameters. By default, the VCD file name where all observed port values and contract validation are stored is called 'out.vcd' and the simulated time is 30 seconds with the default resolution provided by the SystemC library (1 ps).

9.2 Functional Integration: Adding User-specific Implementation Code

The generated code from Section 9.1 can be extended via SystemC ports, channels, and processes to include custom functional behaviour from existing C/C++ code.

This example shows how the system description can be extended such that a new port is used for communicating values and processes are triggered for performing calculations.

The code that was added in theses examples is marked with the FI-START and FI-END comments.

```
struct BLOCK_System : tslsim::block_base
{
    tslsim::input_port Input{ "Input" };
    tslsim::output_port Output{ "Output" };

    /** ******* FI-START ******* */
    // Specify port(s) for input/output values
    sc_core::sc_in<int> FunctionalInputPort{ "FunctionalInputPort" };
    sc_core::sc_out<int> FunctionalOutputPort{ "FunctionalOutputPort" };

    // Prepare SystemC for defining processes in this module
    SC_HAS_PROCESS(BLOCK_System);
    /** ******* FI-END ******** */

    BLOCK_System(tslsim::module_name = "BLOCK_System")
    : block_base("A: Input occurs every 33ms with offset [0,33]ms and jitter 5ms."
                 "G: Reaction(Input,Output) within [0,33]ms.")
    {
        this->instantiate_generator();

        /** ******* FI-START ******* */
        // Specify SystemC thread and sensitivity to event port
        SC_THREAD(function_example);
        sensitive << Input;
        /** ******* FI-END ******** */
    }

    /** ******* FI-START ******* */
    // Simple functional behaviour implemented as a SystemC Process
    // The process is sensitive on the event Input port.  Upon event arrival, it
    // performs a calculation based on the FunctionalInputPort and forwards the
    // result to the FunctionalOutputPort.

    void function_example()
    {
        while (true) {
            auto val = FunctionalInputPort.read();
            val *= 2;

            std::cout << "@" << sc_core::sc_time_stamp()
            << ": Performing calculation. result:" << val << "\n";

            FunctionalOutputPort.write(val);

            // wait for the next event
            // (according to static sensitivity list defined above)
            wait();
        }
    }
    /** ******* FI-END ******** */
};
```

Finally, we create a process that provides input values on the System Context layer:

```
struct BLOCK_Context : tslsim::context_base
{
    tslsim::signal_type s_Input{ "s_Input" };
    tslsim::signal_type s_Output{ "s_Output" };

    /** ******* FI-START ******* */
    // Stimuli signal(s) for connecting the new ports of the block above
    sc_core::sc_signal<int> s_stimuli{ "s_stimuli" };
    sc_core::sc_signal<int> s_stimuli_out{ "s_stimuli_out" };

    // Prepare SystemC for defining processes in this module
    SC_HAS_PROCESS(BLOCK_Context);
    /** ******* FI-END ******** */

    BLOCK_System sys_;

    BLOCK_Context(tslsim::module_name = "BLOCK_Context")
    : context_base( "A: Input occurs every 33ms with offset [0,33]ms and jitter 5ms.\n"
                    "G: Reaction(Input,Output) within [0,33]ms.\n")
    {
        sys_.Input(s_Input);
        sys_.Output(s_Output);

        /** ******* FI-START ******* */

        SC_THREAD(stimuli);

        // Trigger the stimuli process whenever something on the output happens
        sensitive << s_stimuli_out;

        // bind the ports of the BLOCK_System to the stimuli signals
        sys_.FunctionalInputPort(s_stimuli);
        sys_.FunctionalOutputPort(s_stimuli_out);
        /** ******* FI-END ******** */
    }

    /** ******* FI-START ******* */
    // Example process that writes an incrementing value to a signal
    // and waits for its output.
    void stimuli()
    {
        int tmp = 1;
        while (true) {
            std::cout << "@" << sc_core::sc_time_stamp()
            << ": Writing stimuli " << tmp << " to FunctionalInputPort\n";
            s_stimuli.write(tmp++);

            // wait for the next event
            // (according to static sensitivity list defined above)
            wait();
        }
    }
    /** ******* FI-END ******** */
};


void tslsim_main()
{
    BLOCK_Context b;
    tslsim::start_simulation();
}
```

9.3 Supporting Multiple Generators on the Same Output Ports

While Chapter 6 discusses how timing specifications are made operational by the implementation of monitor and generator automata, it leaves the question open how multiple timing specifications patterns interact with each other when they refer the same ports, and particularly the same output ports. For monitors the situation is simple. All events on both input and output ports are observed by the individual monitoring automata, and each automaton decides whether the observed event stream belongs to the corresponding pattern or not. A violation of any pattern automaton causes the violation of the whole specification.

The situation is more complicated for generators. Formally, there is no difference between monitoring and generation. All patterns must be satisfied in order to satisfy the whole specification. The problem lies in the synchronization between the individual patterns. One could implement some parallel composition operation that constructs a combined automaton that produces the intended event streams, or raises an alarm if the composition results in a inconsistent specification. Such an approach however comes with high implementation effort, as it requires an engine that compiles a product automaton from a set of generators, which is possible only based on a fully established intermediate representation machinery.

More important, the resulting automata are in general non-deterministic if we consider them as generators. Non-determinism occurs when the generator has different options to react on input events, such as to decide a time point for sending an output event, or whether to send an event at all or not. This may become a serious problem when the automaton has to decide to generate an output event depending on whether an input event will occur or not. In other words, there may exist states in which decisions have to be taken before all necessary information is available. It is important to note that this lack of knowledge can be reduced but not completely eliminated in general by involving corresponding contract assumptions on how the environment of the considered components will act.

As an example, consider the two specifications **`Reaction(a,c) within [3,6]ms`** and **`Reaction(b,c) within [1,2]ms.`** The first specification expresses that for every input event **a** of some component an output event **c** follows within 3 and $6ms$. The second specification states that for every input event **b** of the same component an output event **c** follows within 1 and $2ms$. We want to construct a generator, which produces event streams that match both specifications. Let's assume that an event **a** occurs at time point t_0. For this event, the generator has to decide a time point t in the interval $[t_0 + 3, t_0 + 6]$ (here and in the following we omit time units) at which it produces a corresponding output event **c** in order to comply with the first specification. If we ask for the latest time point where this decision can be taken, then we find that this is t_0+3. If, otherwise, the decision is made later, say $t_0 + 4$, then the generator will never create an event at any time point in the interval $[t_0 + 3, t_0 + 4)$, which does not comply with the specification. On the other hand, the generator cannot produce a *consistent* decision *before* time point $t_0 + 4$. This is because an event **b** may occur at time point t_0+4 and, according to the second specification, the generator has to produce a corresponding output event **c** in the interval $[t_0 + 5, t_0 + 6]$ for this event. If the two inputs events are causally related with the same event **c**, then the generator has to produce an event **c** in the interval $[t_0 + 5, t_0 + 6]$, which is the intersection of the two identified intervals for event a and `b`, respectively.

The two constraints on the decision point obviously contradict each other. On one hand, the generator has to take a decision no later then $t_0 + 3$, one the other hand, it has to wait until $t_0 + 4$, because there *may* be a related event **b** to be taken into account for the decision. We claim – without proof – that this contradiction cannot be resolved in general. The example above represents a valid counter-example for the construction of generator automata in general, which are able to generate every possible event stream on one hand, and also adhere to the specification in every situation. To sketch such proof, one could cast the above specifications into a game theoretic setting, where the environment generates input events according to the contract assumption, and the generator as the second player has to adhere to the specification of the guarantee by sending output events accordingly. It is to be shown that there is no winning strategy on the generator side, which is able to generate every event stream adhering to the specification.

Therefore, we propose an approach, which is able to generate every event stream adhering to the specification as long as there is a winning strategy for the given specification. However, the generator

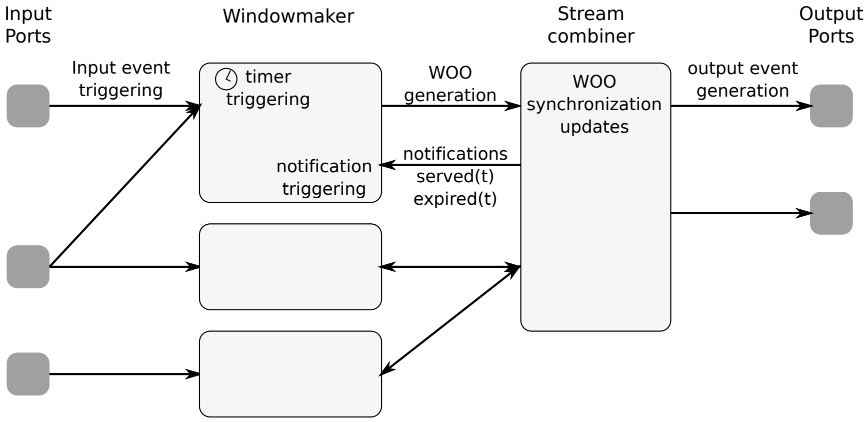


Figure 9.1: Implementation Approach for Multiple Generators

may fail because it can not proceed without producing inconsistencies. In such case, no other strategy exists that is able to generate every possible event stream adhering to the specification.

The approach relies on a notion of *window of opportunity*. Every generator of a specification pattern maintains a set of time windows in which output events may be send. For example, a generator for the first reaction pattern from the above example opens a window in the time span $[t_0 + 3, t_0 + 6]$ whenever it has seen in input event **a** (at time t). A generator for a repetition pattern (re)opens a window of opportunity depending on the period and jitter whenever it has send an output event.

A window of opportunity is defined by a tuple $w = (es, n^-, n^+, excl, t, t^- t^+)$ where

- es is an event specification. For sake of brevity we omit the discussion of event sequences and sets, and define $es = (p, \sigma, id)$, where p is a port, σ is an event value, and id is either specifying that the event must have the corresponding causal identifier, or equals to $*$ which denotes that there is no such causal relation.
- $n^-, n^+ \in \{0, 1, n\}$ is the occurrence requirement for the window. The parameter n^- defines the minimum number of occurrence, and n^+ the maximum, respectively. We require $n^- \leq n^+$ and define $k \leq n$ for any $k \in \mathbb{N}$.
- $excl \in \{Y, N\}$ specifies whether the specified event is exclusive. In the case that $excl = Y$ the window is violated by an event that does not fit the causal identifier. Otherwise, such event is ignored.
- t specifies the creation time of the window.
- $[t + t^-, t + t^+]$ specifies the time window for the occurrences of es.

We say that a window $w = (es, n^-, n^+, excl, t, t^-, t^+)$ is *served* by an event occurrence $(p.(\sigma, id), u)$, where $u \in [t + t^-, t + t^+]$, if $n^+ \neq 0$ and either $es = (p, \sigma, id)$ or $es = (p, \sigma, *)$.

We say that a window $w = (es, n^-, n^+, excl, t, t^-, t^+)$ is *violated* by an event occurrence $(p.(\sigma, id_2), u)$, where $u \in [t + t^-, t + t^+]$, if either $n^+ = 0$ or $es = (p, \sigma, id_1)$, $excl = Y$ and $id_1 \neq id_2$.

We say that a window is *safely ignored* by an event occurrence, if it is neither served nor violated by the event.

Based on the definitions above, we define an algorithm that tries to generate consistent event streams also for multiple specifications that are referring the same output ports. The architecture of the approach is depicted in Figure 9.1. Here, the generators for every component are split into two parts. The figure shows at the left side *Window-Maker* processes - one for each specification pattern, which may listen on input ports, and which create windows according to the pattern semantics. The right side shows a single *Stream-Combiner*, which processes the provided windows, and generates consistent output events according to the definitions above. Whenever an event is generated that serves a window then the Stream-Combiner notifies the corresponding Window-Maker processes about the time stamp and the event that has been generated. Whenever a window expires then the Stream-Combiner notifies the corresponding Window-Maker about the time of expiration.

Whenever a Window-Maker is triggered, either by an incoming event, by some timeout, or by a notification form the Stream-Combiner then the process updates its window(s). The Stream-Combiner is activated whenever at least one window is updated. The process decides which currently existing windows are served – and which are not. It also identifies potential inconsistencies, which will lead to aborting the simulation with a corresponding error. The algorithm skeleton is shown below:

```
Windowmaker(trigger) {
  switch (trigger)     // process can be triggered in four different ways
  {
    case timer(t):     // a timer event has occurred at t
      create window w;
      send w to Streamcombiner;
    case event(e,t):   // Input event e has occurred at t
      ...
    case served(w,t):  // window w has been served at t
      ...
    case expired(w,t): // window w has been expired at t
      ...
  }
}

Streamcombiner(trigger) {
  switch (trigger)
  {
    case window(w):  // A Windowmaker has setup a new window
      process new window w and select new events to be send;
    case event(t):   // An event has to be send
      send event;
      notify Windowmaker about serving;
    case expire(w):  // A window has expired.
      notify Windowmaker about expiration;
  }
}
```

Multiple windows may cause inter-dependencies, which have to be considered in the Stream-Combiner process. Particularly, windows may be inconsistent:

- If $w_i = (E, 0, 0, ...)$ and $w_j = (E, 1, , ...)$, i.e., a "must not" window intersects with "must" window.
- Windows may be conflicting. If $w_i = (E.id_i, , , Y, ...)$ and $w_j = (E.id_j, , , Y, ...)$, and $id_i \neq id_j$, i.e. exclusive windows with different ids.

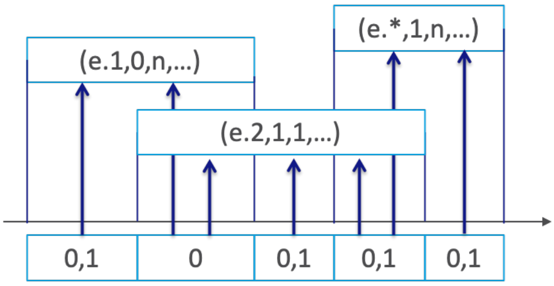


Figure 9.2: Window Partitioning Example

Given a set of windows, the Stream combiner process must select events and time points that match all window constraints, and that do not conflict. To this end, the process implements the algorithm as follows. Suppose a set of active windows $W = \{w_i\}$. First, the algorithm calculates a partitioning $\bar{W} = \{\bar{w_j}\}$ of W with respect to the intersection of all windows (cf. Figure 9.2). This way, every window $w_i \in W$ is partitioned into a set $\bar{W_i} = \{\bar{w_{ij}}\} \subseteq \bar{W}$.

Next, the algorithm calculates a decision function $d : \bar{W} \rightarrow \{0,1\}$ that satisfies the following constraint system:

1. For all $w_i = (E, min_i, max_i, ...)$ holds $min_i \leq \sum_{\bar{w_{ij}} \in \bar{W_i}} d(\bar{w_{ij}}) \leq max_i$.

2. For all i, j and all $\bar{w} \in \bar{W_i} \cap \bar{W_j}$ holds $d(\bar{w}) = 1$ implies that w_i and w_j are not conflicting.

If no decision function that satisfies the constraints above can be found then the Stream-Combiner creates an error message informing the user about the potential inconsistency, and terminates the simulation. Otherwise, the algorithm finally selects for every $\bar{w} \in \bar{W}$ such that $d(\bar{w}) = 1$ a non-conflicting event and a time point within the intersection window of $\bar{w}$.

Decision taken by Stream combiner are buffered. That is, a decision to send an event (E, t) is deferred until t. At this time point also the notification `served(w,t)` is emitted. On every window update, some decisions may become invalid and are revised if necessary.

The individual Window-Maker processes update their window(s) with respect to the underlying specification pattern whenever they are triggered. As Figure 9.1 shows there are three potential trigger sources. All Window-Maker processes are triggered by corresponding notifications from the Stream-Combiner process. According to the actual notification (serve, expire) they update their internal state and check whether the corresponding pattern may be violated. For example, an expired window with an occurrence of $occ = must$ must not expire but being served.

All except the event occurrence patterns are also triggered by incoming events. Typically, an incoming event causes the generation of a new window. Thus, every Window-Maker process may possess multiple windows at the same time. The repetition pattern is not triggered by an incoming event. It is triggered at time 0 for the initial event generation. After that, it is only triggered by notifications.

The following tables show the window parameters generated by the Stream-Combiner for the individual specification patterns. The repetition pattern creates non-causal events $id = *$ that however must occur. The time window depends on the selection of periods, jitters, and on the time point of the previous event. Reaction and age patterns also generate non-causal events $(id = *)$. If a reaction pattern contains a `"k out of n"` clause, then the event generation may be optional, depending on the actual number of ignored events. However, if no once clause exists then the pattern creates optional windows after the first event generation until the reaction time interval has elapsed. For age pattern, the windows are always optional. The causal sisters of reaction and age patterns produce events with fixed id parameter. While a reaction window allows the occurrence of events with different id, the age pattern does not. For the occurrence holds the same as for the non-causal case.

In the tables below we make the following notions. For a window $w = (es, n^-, n^+, excl, t, t^-t^+)$ we denote $es(w) = es$ and $t(w) = t$.

Single Event: `E occurs within I` The process is triggered by timer event at $t = 0$, `served(w,t)`, and `expired(w,t)`.

Trigger	Window
Timer($t = 0$)	$w = (E.*, 1, 1, N, 0, I^-, I^+)$
Served(w, t)	ignore
Expired(w, t)	fail

Repetition: `E occurs every I with offset O and jitter J` The process is triggered by timer event at $t = 0$, `served(w,t)`, and `expired(w,t)`.

Trigger	Window
Timer($t = 0$)	$w = (E.*, 1, 1, N, 0, O^-, O^+ + J)$
Served(w, t)	calculate u_i from t and $w(t)$, $w = (E.*, 1, 1, N, u_i, I^-, I^+J)$
Expired(w, t)	fail

Reaction: `whenever E occurs then F occurs within I` The process is triggered by events (E, t), `served(w,t)`, and `expired(w,t)`.

Trigger	Window
Event(E, t)	$w = (F.*, 1, n, N, t, I^-, I^+)$
Served(w, t)	$w' = (es(w), 0, n, N, t(w), I^-, I^+)$
Expired(w, t)	fail if $w = (F.*, 1, ...)$, ignore otherwise

Reaction: `whenever E occurs then F occurs within I, K out of N times` The process is triggered by events (E, t), `served(w,t)`, and `expired(w,t)`.

Trigger	Window
Event(E, t)	$w = (F.*, min, n, N, t, I^-, I^+)$, where $min \in \{0, 1\}$ depending on number of previously expired windows.
Served(w, t)	$w' = (es(w), 0, n, N, t(w), I^-, I^+)$
Expired(w, t)	fail if $w = (F.*, 1, ...)$, ignore otherwise

Reaction: `whenever E occurs then F occurs within I once` The process is triggered by events (E, t), `served(w,t)`, and `expired(w,t)`.

Trigger	Window
Event(E, t)	$w = (F.*, 1, 1, N, t, I^-, I^+)$
Served(w, t)	$w' = (es(w), 0, 0, N, t(w), I^-, I^+)$
Expired(w, t)	fail if $w = (F.*, 1, ...)$, ignore otherwise

Age: `whenever E occurs then F has occurred within I` The process is triggered by events (E, t), `served(w,t)`, and `expired(w,t)`.

Trigger	Window
Event(E, t)	$w = (F.*, 0, n, N, t, I^-, I^+)$
Served(w, t)	$w' = (es(w), 0, n, N, t(w), I^-, I^+)$
Expired(w, t)	ignore

Causal Reaction: `Reaction(E,F) within I` The process is triggered by events (E, t), `served(w,t)`, and `expired(w,t)`.

Trigger	Window
Event$(E.id, t)$	$w = (F.id, 1, n, N, t, I^-, I^+)$
Served(w, t)	$w' = (es(w), 0, n, N, t(w), I^-, I^+)$
Expired(w, t)	fail if $w = (F.id, 1, ...)$, ignore otherwise

Causal Age: `Age(E,F) within I` The process is triggered by events (E, t), `served(w,t)`, and `expired(w,t)`.

Trigger	Window
Event(E, t)	$w = (F.id, 0, n, Y, t, I^-, I^+)$
Served(w, t)	$w' = (es(w), 0, n, Y, t(w), I^-, I^+)$
Expired(w, t)	ignore

Patterns that include event sequences and/or event sets require an additional book keeping. For incoming events windows are not created immediately but only if the event streams are complete (cf. Chapter 6, and particularly Section 6.2). Similar holds for output event, for which sequences of windows have to be created. For example, for an event sequence (F, G) we would have:

Trigger	Window
Event(E, t)	$w = (F.*, 1, n, N, t, I^-, I^+)$

Served(w, t)	if `eventstore.complete()` then
	$w' = (F.*, 0, n, N, t(w), I^-, I^+)$ (start next round)
	else
	$w' = ($`eventstore.nextEvent()`$, 1, n, N, t(w), I^-, I^+)$

10 Error Types

Based on the process steps of the considered use cases detailed in Chapter 8, the relevant error types are described in the course of this chapter. While each error class usually corresponds to a single step, the error types assigned to their specific classes are summed up in Table 10.1 to Table 10.12.

Each table consists of five columns. While the *Error Type* is named in the first column, the second one contains the *Error Message* to be output. The subsequent columns specify the components via which the error message is forwarded to the user. In addition to a console output (*C*) and a decoration of the affected model elements (*M*), it is also possible to display an error dialog (*D*).

To enable referencing to the SysML model and to increase the traceability of the error types used, the error messages are supplemented with additional error attributes. They include

- the name of the affected block (**BlockName**), port (**PortName**), connection (**ConnName**), requirement (**ReqName**), relation (**RelName**), and/or flow property (**FloPName**),
- the text of the considered or violated pattern (**PatternText**),
- as well as a more detailed error message (**DetErrorMsg**) for debugging purposes.

10.1 Syntax Check

According to its process steps, the first use case uses error types of the classes STARTUP, LOAD_MODEL, VALIDATE_MODEL_PAPYRUS, VALIDATE_MODEL_MULTIC, GENERATE, BUILD_VIT, and ELABORATE.

10.1.1 Startup

Error Type	Error Message	C	M	D
SAVE_MODIFIED_PROJECTS	*“Failed to save modified projects!”*	✓		
OPEN_PAPYRUS_PERSPECTIVE	*“Failed to open Papyrus perspective!”*	✓		
OPEN_CONSOLE_VIEW	*“Failed to open status console!”*	✓		
ABORT_BY_USER	*“Failed to start up due to user termination!”*	✓		

Table 10.1: Error Types of Class STARTUP

10.1.2 Load Model

Error Type	Error Message	C	M	D
EMPTY_WORKSPACE	*“Failed to load SysML model due to empty an workspace!”*	✓		✓
ABORT_BY_USER	*“Failed to load SysML model due to user termination!”*	✓		

EMPTY_PROJECT	*“Failed to load SysML model due to an empty project!”*	✓		✓
MULTIPLE_MODELS	*“Failed to load project due to multiple existing SysML models!”*	✓		✓
OPEN_EDITOR	*“Failed to open SysML model in Papyrus!”*	✓		

Table 10.2: Error Types of Class LOAD_MODEL

10.1.3 Validate Model Papyrus

Error Type	Error Message	C	M	D
RETRIEVE_MODEL_SET	*“Failed to retrieve model set!”*	✓		
FIND_MODEL_ROOT	*“Failed to find model root!”*	✓		
EXECUTE_VALIDATION	*“Failed to execute Papyrus model validation!”*	✓		
REOPEN_CONSOLE	*“Failed to reopen status console!”*	✓		
FIND_MARKERS	*“Failed to find error markers!”*	✓		
EXISTING_ERRORS	*“Failed to validate SysML model using Papyrus model validation! Check ’Model Validation’ tab for further information.”*	✓		
ABORT_BY_USER	*“Failed to validate model due tu user termination!”*	✓		

Table 10.3: Error Types of Class VALIDATE_MODEL_PAPYRUS

10.1.4 Validate Model MULTIC

Error Type	Error Message	C	M	D
EXECUTE_VALIDATION	*“Failed to execute MULTIC model validation!”*	✓		
EMPTY_MODEL	*“Empty SysML model!”*	✓		
MISSING_INTERFACE_BLOCK	*“Missing interface block! Each SysML model must contain exactly one interface block named ’Event’.”*	✓		
MISSING_CONTEXT_BLOCK	*“Missing context block! Each SysML model must contain exactly one context block.”*	✓		
MULTIPLE_CONTEXT_BLOCKS	*“Multiple context blocks! Each SysML model must contain exactly one context block.”*	✓		
MISSING_SYSTEM_BLOCK	*“Missing system block! Each SysML model must contain exactly one system block.”*	✓		
MULTIPLE_SYSTEM_BLOCKS	*“Multiple system block! Each SysML model must contain exactly one system block.”*	✓		
MISSING_PORTS_ON_SYSTEM_BLOCK	*“Missing ports on system block ’**BlockName**’! The system block must have at least one output port.”*	✓	✓	
INVALID_PORT_TYPE	*“Invalid type of port ’**PortName**’ of block ’**BlockName**’! Each port must be of type ’Event’.”*	✓	✓	
INVALID_CONNECTION	*“Invalid connection ’**ConnName**’! At the same hierarchical level, output ports must be connected to input ports.”*	✓	✓	

INVALID_CONNECTION_HIERARCHICAL	*“Invalid connection '**ConnName**'! Between two hierarchical levels, input ports must be connected to input ports and output ports to output ports.”*	✓	✓	
MISSING_CONNECTION_HIERARCHICAL	*“Missing connection on block '**BlockName**'! Between two hierarchical levels, each input/output port of the superordinate block must be connected to at least one input/output port of the subordinate block.”*	✓	✓	
MISSING_CONNECTION	*“Missing connection on port '**PortName**' of block '**BlockName**'! At the same hierarchical level, each output port must be connected to at least one input port.”*	✓	✓	
MISSING_REQUIREMENT	*“Missing requirement for block '**BlockName**'! Each block must have exactly one requirement.”*	✓	✓	
MULTIPLE_REQUIREMENTS	*“Multiple requirements for block '**BlockName**'! Each block must have exactly one requirement.”*	✓	✓	
MISSING_SATISFY_RELATION	*“Missing satisfy relation for requirement '**ReqName**'! Each block must be linked to its requirement by exactly one satisfy relation.”*	✓	✓	
MULTIPLE_SATISFY_RELATIONS	*“Multiple satisfy relations for requirement '**ReqName**'! Each block must be linked to its requirement by exactly one satisfy relation.”*	✓	✓	
MISSING_REQUIREMENT_PROPERTIES	*“Missing properties for requirement '**ReqName**'! Each requirement must contain a 'Name', an 'ID' and a 'Text'.”*	✓	✓	
INVALID_ABSTRACTION_TYPE	*“Invalid abstraction type of relation '**RelName**'! Each block must be linked to its requirement by exactly one satisfy relation.”*	✓	✓	
MISSING_FLOW_PROPERTY	*“Missing flow property for interface block '**BlockName**'! Each interface block must contain exactly one flow property.”*	✓	✓	
MULTIPLE_FLOW_PROPERTIES	*“Multiple flow properties for interface block '**BlockName**'! Each interface block must contain exactly one flow property.”*	✓	✓	
INVALID_FLOW_PROPERTY_NAME	*“Invalid name of flow property '**FloPName**'! The flow property must be named 'EventFlow'.”*	✓	✓	
INVALID_FLOW_PROPERTY_DIRECTION	*“Invalid direction of flow property '**FloPName**'! The flow property direction must be set to 'out'.”*	✓	✓	
HYPHEN_IN_BLOCK_NAME	*“Invalid name of block '**BlockName**'! Block names must not contain '-'.”*	✓	✓	
SLASH_IN_BLOCK_NAME	*“Invalid name of block '**BlockName**'! Block names must not contain '/'.”*	✓	✓	
COLON_IN_BLOCK_NAME	*“Invalid name of block '**BlockName**'! Block names must not contain ':'.”*	✓	✓	
SPECIAL_CHARACTER_IN_PORT_NAME	*“Invalid name of port '**PortName**' of block '**BlockName**'! Port names must not contain any special characters.”*	✓	✓	
HYPHEN_IN_CONNECTION_NAME	*“Invalid name of connection '**ConnName**'! Connection names must not contain '-'.”*	✓	✓	
SLASH_IN_CONNECTION_NAME	*“Invalid name of connection '**ConnName**'! Connection names must not contain '/'.”*	✓	✓	

COLON_IN_CONNECTION_NAME	*“Invalid name of connection '**ConnName**'! Connection names must not contain ':'.”*	✓	✓	
ABORT_BY_USER	*“Failed to validate model with respect to the MULTIC Design Rules due to user termination!”*	✓		

Table 10.4: Error Types of Class VALIDATE_MODEL_MULTIC

10.1.5 Generate

Error Type	Error Message	C	M	D
EXECUTE_GENERATION	*“Failed to generate simulation model!”*	✓		
ABORT_BY_USER	*“Failed to generate simulation model due to user termination!”*	✓		

Table 10.5: Error Types of Class GENERATE

10.1.6 Build VIT

Error Type	Error Message	C	M	D
RESOLVE_SCRIPT_FILE_URL	*“Failed to resolve build VIT script file URL!”*	✓		
MISSING_SCRIPT_FILE	*“Missing build VIT script file!”*	✓		
REDIRECT_OUTPUT	*“Failed to redirect build VIT output stream!”*	✓		
CONFIGURE_MAKE	*“Failed to configure make!”*	✓		
ABORT_BY_USER	*“Failed to build VIT due to user termination!”*	✓		
CLOSE_REDIRECT	*“Failed to close build VIT output stream!”*	✓		
INVALID_COMMAND	*“Invalid build VIT command!”*	✓		
BUILD_BINARY	*“Failed to compile binary!”*	✓		

Table 10.6: Error Types of Class BUILD_VIT

10.1.7 Elaborate

Error Type	Error Message	C	M	D
INVALID_COMMAND	*“Invalid elaborate command!”*	✓		
REDIRECT_OUTPUT	*“Failed to redirect elaborate output stream!”*	✓		
EXECUTE_ELABORATION	*“Failed to elaborate simulation model!”*	✓		
ABORT_BY_USER	*“Failed to elaborate simulation model due to user termination!”*	✓		
CLOSE_REDIRECT	*“Failed to close elaborate output stream!”*	✓		
SYSTEMC_GENERAL	*“Failed to elaborate simulation model! **DetErrorMsg**.”*	✓		
SYNTAX_GENERAL	*“Failed to parse contract of block '**BlockName**' due to a syntax error! **DetErrorMsg**.”*	✓	✓	

SEMANTIC_GENERAL	*“Failed to parse contract of block '**BlockName**' due to a semantic error! **DetErrorMsg**.”*	✓	✓	
CONTRACT_NOT_FOUND	*“Missing contract in requirement of block '**BlockName**'! Each requirement must contain at least one guarantee statement.”*	✓	✓	
MISSING_PORT	*“Missing port '**PortName**' of block '**BlockName**' referenced in pattern '**PatternText**'! Each port used in a contract must be explicitly modeled in the SysML model.”*	✓	✓	
MISSING_SIGNAL	*“Missing signal 's_**PortName**' of block '**BlockName**' referenced in pattern '**PatternText**'! Each signal used in a contract must be explicitly modeled in the SysML model.”*	✓	✓	
OUTPUT_IN_ASSUMPTION	*“Invalid use of output port '**PortName**' of block '**BlockName**' in pattern '**PatternText**'! Assumption statements must only contain input ports.”*	✓	✓	

Table 10.7: Error Types of Class ELABORATE

10.2 Virtual Integration Test

In addition to the previously defined error types, the error classes SET_VIT_PARAMETERS, SIMULATE_VIT, and SHOW_VCD are of exclusive importance for the *Virtual Integration Test*.

10.2.1 Set VIT Parameters

Error Type	Error Message	C	M	D
ABORT_BY_USER	*“Failed to set VIT parameters due to user termination!”*	✓		

Table 10.8: Error Types of Class SET_VIT_PARAMETERS

10.2.2 Simulate VIT

Error Type	Error Message	C	M	D
INVALID_COMMAND	*“Invalid simulate VIT command!”*	✓		
REDIRECT_OUTPUT	*“Failed to redirect simulate VIT output stream!”*	✓		
EXECUTE_SIMULATION	*“Failed to simulate model!”*	✓		
ABORT_BY_USER	*“Failed to simulate VIT due to user termination!”*	✓		
CLOSE_REDIRECT	*“Failed to close simulate VIT output stream!”*	✓		
PATTERN_FAILED	*“Pattern '**PatternText**' of block '**BlockName**' failed! **Reason**.”*	✓	✓	
GENERATOR_INCONSISTENCY	*“Failed to find a consistent configuration of slices for pattern '**PatternText**'!”*	✓		

GENERATOR_WINDOW_ERROR	*“Failed to handle existing window for pattern '**PatternText**'! **Reason**.”*	✓		

Table 10.9: Error Types of Class SIMULATE_VIT

10.2.3 Show VCD

Error Type	Error Message	C	M	D
INVALID_COMMAND	*“Invalid show VCD command!”*	✓		
ABORT_BY_USER	*“Failed to show VCD file due to user termination!”*	✓		

Table 10.10: Error Types of Class SHOW_VCD

10.3 Functional Integration

The error types assigned to use case *Functional Integration* can be divided into the two error classes BUILD_FI and IMPORT_CDT. Moreover, the error types of the *Syntax Check* are still relevant.

10.3.1 Build FI

Error Type	Error Message	C	M	D
RESOLVE_SCRIPT_FILE_URL	*“Failed to resolve build FI script file URL!”*	✓		
MISSING_SCRIPT_FILE	*“Missing build FI script file!”*	✓		
REDIRECT_OUTPUT	*“Failed to redirect build FI output stream!”*	✓		
CREATE_CDT_PROJECT	*“Failed to create CDT project!”*	✓		
ABORT_BY_USER	*“Failed to build FI due to user termination!”*	✓		
CLOSE_REDIRECT	*“Failed to close build FI output stream!”*	✓		

Table 10.11: Error Types of Class BUILD_FI

10.3.2 Import CDT

Error Type	Error Message	C	M	D
LOAD_CDT_PROJECT	*“Failed to load CDT project in Eclipse!”*	✓		
ABORT_BY_USER	*“Failed to import CDT project due to user termination!”*	✓		

Table 10.12: Error Types of Class IMPORT_CDT

11 Test Models

In order to be able to check the correct functioning and the correct output of the error types presented in Chapter 10 during the individual development steps of the prototypical tool support, a series of test models were defined within the project. In this chapter, each of the used test cases is described briefly, followed by the relevant error types and the expected error handling behavior.

Since the *Functional Integration* use case only provides a developer with the files generated in the previous steps, no separate test models were created for it. All test models are built on the minimal example presented in Section 11.1.3 and are provided with this report.

11.1 Application Examples

The development was based on three application examples. In addition to the *Emergency Stop System* case study from the predecessor project [3, p. 80–126] at Functional Level A (see Section 11.1.1) and Functional Level B (see Section 11.1.2), the so-called *tinySysMLModel* (see Section 11.1.3) was used as a minimal example for the vertical breakthrough.

11.1.1 Emergency Stop System (Functional Level A)

<table>
<tr><th>AE-1</th><th><b><i>Emergency Stop System (Functional Level A)</i></b></th></tr>
<tr><td>Description</td><td><i>Case study of the predecessor project at Functional Level A. All under- and oversampling patterns are omitted for the sake of simplicity.</i></td></tr>
<tr><td>Error Types</td><td><i>Application example without any errors.</i></td></tr>
<tr><td colspan="2">Expected Console Output for UC1</td></tr>
<tr><td colspan="2">• <i>Syntax Check successfully completed!</i></td></tr>
<tr><td colspan="2">Expected Console Output for UC2</td></tr>
<tr><td colspan="2">• <i>Syntax Check successfully completed!</i>
• <i>Virtual Integration Test successfully completed!</i></td></tr>
<tr><td colspan="2">Expected Console Output for UC3</td></tr>
<tr><td colspan="2">• <i>Syntax Check successfully completed!</i>
• <i>Functional Integration successfully completed!</i></td></tr>
</table>

Table 11.1: Application Example `MULTIC-Tooling_ADAS_FuncA`

11.1.2 Emergency Stop System (Functional Level B)

AE-2	***Emergency Stop System (Functional Level B)***
Description	*Case study of the predecessor project at Functional Level A. All under- and oversampling patterns are omitted for the sake of simplicity.*
Error Types	*Application example without any errors.*
Expected Console Output for UC1	
• *Syntax Check successfully completed!*	
Expected Console Output for UC2	
• *Syntax Check successfully completed!* • *Virtual Integration Test successfully completed!*	
Expected Console Output for UC3	
• *Syntax Check successfully completed!* • *Functional Integration successfully completed!*	

Table 11.2: Application Example `MULTIC-Tooling_ADAS_FuncB`

11.1.3 tinySysMLModel Example

AE-3	***tinySysMLModel***
Description	*Minimal example used for the vertical breakthrough.*
Error Types	*Application example without any errors.*
Expected Console Output for UC1	
• *Syntax Check successfully completed!*	
Expected Console Output for UC2	
• *Syntax Check successfully completed!* • *Virtual Integration Test successfully completed!*	
Expected Console Output for UC3	
• *Syntax Check successfully completed!* • *Functional Integration successfully completed!*	

Table 11.3: Application Example `MULTIC-Tooling_tinySysMLModel`

11.1.4 tinyDemonstrator Example

AE-4	***tinyDemonstrator***
Description	*Minimal example with multiple generators per output port.*
Error Types	*Application example without any errors.*
Expected Console Output for UC1	
• *Syntax Check successfully completed!*	
Expected Console Output for UC2	
• *Syntax Check successfully completed!* • *Virtual Integration Test successfully completed!*	
Expected Console Output for UC3	
• *Syntax Check successfully completed!* • *Functional Integration successfully completed!*	

Table 11.4: Application Example `MULTIC-Tooling_tinyDemonstrator`

11.2 Syntax Check

During the development phase, test models for steps *Load Model*, *Validate Model Papyrus*, *Validate Model MULTIC*, and *Elaborate* were created for the *Syntax Check* use case.

11.2.1 Startup

No test models are available for this process step.

11.2.2 Load Model

TC1-01	***Empty Project***
Description	*A project without any SysML models.*
Error Types	`EMPTY_PROJECT`
Expected Console Output	
• *Failed to load SysML model due to an empty project!*	
Expected Dialog Output	
• *Failed to load SysML model due to an empty project!*	

Table 11.5: Test Model `TC1-01_EmptyProject`

TC1-03	***Multiple Models***
Description	*A project containing two SysML models.*
Error Types	`MULTIPLE_MODELS`
Expected Console Output	
- *Failed to load project due to multiple existing SysML models!*	
Expected Dialog Output	
- *Failed to load project due to multiple existing SysML models!*	

Table 11.6: Test Model `TC1-03_MultipleModels`

11.2.3 Validate Model Papyrus

TC2-24	***Invalid Satisfy Direction***
Description	*A SysML model which contains the satisfy relation 'SatisfySubSystemB' that connects the correct block with its corresponding requirement, but defines the link in the wrong direction.*
Error Types	`EXISTING_ERRORS`
Expected Console Output	
- *Failed to validate SysML model using Papyrus model validation! Check 'Model Validation' tab for further information.*	

Table 11.7: Test Model `TC2-24_InvalidSatisfyDirection`

TC4-05	***Empty Requirement Text***
Description	*A SysML model with a requirement which 'text' attribute is empty.*
Error Types	`EXISTING_ERRORS`
Expected Console Output	
- *Failed to validate SysML model using Papyrus model validation! Check 'Model Validation' tab for further information.*	

Table 11.8: Test Model `TC4-05_EmptyRequirementText`

TC5-07	***Missing Contract***
Description	*A SysML model in which no contract has been defined in requirement 'CONTRACT_System'.*
Error Types	`EXISTING_ERRORS`

Expected Console Output
• *Failed to validate SysML model using Papyrus model validation! Check 'Model Validation' tab for further information.*

Table 11.9: Test Model `TC5-07_MissingContract`

11.2.4 Validate Model MULTIC

TC1-02	***Empty Model***
Description	*A SysML model without any diagrams or model elements.*
Error Types	`EMPTY_MODEL, MISSING_INTERFACE_BLOCK, MISSING_CONTEXT_BLOCK`
Expected Console Output	
• *Empty SysML model!* • *Missing interface block! Each SysML model must contain exactly one interface block named 'Event'.* • *Missing context block! Each SysML model must contain exactly one context block.*	

Table 11.10: Test Model `TC1-02_EmptyModel`

TC2-01	***Missing Context Block***
Description	*A SysML model missing a context block.*
Error Types	`MISSING_CONTEXT_BLOCK`
Expected Console Output	
• *Missing context block! Each SysML model must contain exactly one context block.*	

Table 11.11: Test Model `TC2-01_MissingContextBlock`

TC2-02	***Context Block With Ports***
Description	*A SysML model containing a context block with ports.*
Error Types	`MISSING_CONTEXT_BLOCK, MISSING_CONNECTION_HIERARCHICAL, MISSING_CONNECTION, MISSING_REQUIREMENT`
Expected Console Output	

• *Missing context block! Each SysML model must contain exactly one context block.* • *Missing connection on block 'BLOCK_Context'! Between two hierarchical levels, each input/output port of the superordinate block must be connected to at least one input/output port of the subordinate block.* • *Missing connection on port 'Input' of block 'BLOCK_Context'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing connection on port 'Output' of block 'BLOCK_Context'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing requirement for block 'BLOCK_Context'! Each block must have exactly one requirement.*
Expected Model Decoration
• *Block 'BLOCK_System'.* • *Port 'Input' of block 'BLOCK_System'.* • *Port 'Output' of block 'BLOCK_System'.*

Table 11.12: Test Model TC2-02_ContextBlockWithPorts

TC2-03	***Missing System Block***
Description	*A SysML model without a system block.*
Error Types	MISSING_SYSTEM_BLOCK, MISSING_CONNECTION, MISSING_SATISFY_RELATION
Expected Console Output	
• *Missing system block! Each SysML model must contain exactly one system block.* • *Missing connection on port 'InputSubA' of block 'BLOCK_SubSystemA'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing connection on port 'InputSubB' of block 'BLOCK_SubSystemB'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing connection on port 'OutputSubB' of block 'BLOCK_SubSystemB'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing connection on port 'OutputSubA' of block 'BLOCK_SubSystemA'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing satisfy relation for requirement 'CONTRACT_System'! Each block must be linked to its requirement by exactly one satisfy relation.*	
Expected Model Decoration	

• *Port '`InputSubA`' of block '`BLOCK_SubSystemA`'.* • *Port '`OutputSubA`' of block '`BLOCK_SubSystemA`'.* • *Port '`InputSubB`' of block '`BLOCK_SubSystemB`'.* • *Port '`OutputSubB`' of block '`BLOCK_SubSystemB`'.* • *Requirement '`CONTRACT_System`'.*

Table 11.13: Test Model `TC2-03_MissingSystemBlock`

TC2-04	***Invalid Context System Link***
Description	*A SysML model in which the context block and the system block are not linked by an aggregation.*
Error Types	`MISSING_SYSTEM_BLOCK`
Expected Console Output	
• *Missing system block! Each SysML model must contain exactly one system block.*	

Table 11.14: Test Model `TC2-04_InvalidContextSystemLink`

TC2-05	***System Block Without Ports***
Description	*A SysML model containing a system block without input or output ports.*
Error Types	`MISSING_PORTS_ON_SYSTEM_BLOCK, MISSING_CONNECTION`
Expected Console Output	
• *Missing ports on system block '`BLOCK_System`'! The system block must have at least one output port.* • *Missing connection on port '`OutputSubB`' of block '`BLOCK_SubSystemB`'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing connection on port '`InputSubA`' of block '`BLOCK_SubSystemA`'! At the same hierarchical level, each output port must be connected to at least one input port.*	
Expected Model Decoration	
• *Block '`BLOCK_System`'.* • *Port '`InputSubA`' of block '`BLOCK_SubSystemA`'.* • *Port '`OutputSubB`' of block '`BLOCK_SubSystemB`'.*	

Table 11.15: Test Model `TC2-05_SystemBlockWithoutPorts`

TC2-06	***Missing Interface Block***
Description	*A SysML model without any interface block.*
Error Types	MISSING_INTERFACE_BLOCK, INVALID_PORT_TYPE
Expected Console Output	
• *Missing interface block! Each SysML model must contain exactly one interface block named 'Event'.* • *Invalid type of port 'InputSubB' of block 'BLOCK_SubSystemB'! Each port must be of type 'Event'.* • *Invalid type of port 'InputSubA' of block 'BLOCK_SubSystemA'! Each port must be of type 'Event'.* • *Invalid type of port 'OutputSubA' of block 'BLOCK_SubSystemA'! Each port must be of type 'Event'.* • *Invalid type of port 'OutputSubB' of block 'BLOCK_SubSystemB'! Each port must be of type 'Event'.* • *Invalid type of port 'Input' of block 'BLOCK_System'! Each port must be of type 'Event'.* • *Invalid type of port 'Output' of block 'BLOCK_System'! Each port must be of type 'Event'.*	
Expected Model Decoration	
• *Port 'Input' of block 'BLOCK_System'.* • *Port 'Output' of block 'BLOCK_System'.* • *Port 'InputSubA' of block 'BLOCK_SubSystemA'.* • *Port 'OutputSubA' of block 'BLOCK_SubSystemA'.* • *Port 'InputSubB' of block 'BLOCK_SubSystemB'.* • *Port 'OutputSubB' of block 'BLOCK_SubSystemB'.*	

Table 11.16: Test Model TC2-06_MissingInterfaceBlock

TC2-07	***Missing Flow Property***
Description	*A SysML model with an interface block for which no 'EventFlow' flow property is defined.*
Error Types	MISSING_FLOW_PROPERTY
Expected Console Output	
• *Missing flow property for interface block 'Event'! Each interface block must contain exactly one flow property.*	
Expected Model Decoration	

- *Interface block 'Event'.*

Table 11.17: Test Model TC2-07_MissingFlowProperty

TC2-08	***Invalid Flow Direction***
Description	*A SysML model with an 'EventFlow' flow property which 'Direction' attribute is not set to 'out'.*
Error Types	INVALID_FLOW_PROPERTY_DIRECTION
Expected Console Output	
- *Invalid direction of 'EventFlow'! The flow property direction must be set to 'out'.*	
Expected Model Decoration	
- *Interface block 'Event'.*	

Table 11.18: Test Model TC2-08_InvalidFlowDirection

TC2-09	***Hyphen In System Block Name***
Description	*A SysML model in which the name of the system block contains a hyphen.*
Error Types	HYPHEN_IN_BLOCK_NAME
Expected Console Output	
- *Invalid name of block 'BLOCK_System_Hyphen-Test'! Block names must not contain '-'.*	
Expected Model Decoration	
- *Block 'BLOCK_System_Hyphen-Test'.*	

Table 11.19: Test Model TC2-09_HyphenInSystemBlockName

TC2-10	***Slash In System Block Name***
Description	*A SysML model in which the name of the system block contains a slash.*
Error Types	SLASH_IN_BLOCK_NAME
Expected Console Output	
- *Invalid name of block 'BLOCK_System_Slash/Test'! Block names must not contain '/'.*	
Expected Model Decoration	

- *Block 'BLOCK_System_Slash/Test'.*

Table 11.20: Test Model TC2-10_SlashInSystemBlockName

TC2-11	***Colon In System Block Name***
Description	*A SysML model in which the name of the system block contains a colon.*
Error Types	COLON_IN_BLOCK_NAME
Expected Console Output	
- *Invalid name of block 'BLOCK_System_Colon:Test'! Block names must not contain ':'.*	
Expected Model Decoration	
- *Block 'BLOCK_System_Colon:Test'.*	

Table 11.21: Test Model TC2-11_ColonInSystemBlockName

TC2-12	***Hyphen In Top-Level Requirement Name***
Description	*A SysML model in which the name of a requirement contains a hyphen.*
Error Types	*Test model without any errors.*
- *Syntax Check successfully completed!*	
Expected Console Output for UC2	
- *Syntax Check successfully completed!* - *Virtual Integration Test successfully completed!*	
Expected Console Output for UC3	
- *Syntax Check successfully completed!* - *Functional Integration successfully completed!*	

Table 11.22: Test Model TC2-12_HyphenInTopLevelRequirementName

TC2-13	***Slash In Top-Level Requirement Name***
Description	*A SysML model in which the name of a requirement contains a slash.*
Error Types	*Test model without any errors.*

- *Syntax Check successfully completed!*
Expected Console Output for UC2
- *Syntax Check successfully completed!* - *Virtual Integration Test successfully completed!*
Expected Console Output for UC3
- *Syntax Check successfully completed!* - *Functional Integration successfully completed!*

Table 11.23: Test Model `TC2-13_SlashInTopLevelRequirementName`

TC2-14	***Colon In Top-Level Requirement Name***
Description	*A SysML model in which the name of a requirement contains a colon.*
Error Types	*Test model without any errors.*
- *Syntax Check successfully completed!*	
Expected Console Output for UC2	
- *Syntax Check successfully completed!* - *Virtual Integration Test successfully completed!*	
Expected Console Output for UC3	
- *Syntax Check successfully completed!* - *Functional Integration successfully completed!*	

Table 11.24: Test Model `TC2-14_ColonInTopLevelRequirementName`

TC2-15	***Hyphen In System Block Input Port Name***
Description	*A SysML model in which the name of the input port of the system block contains a hyphen.*
Error Types	`SPECIAL_CHARACTER_IN_PORT_NAME`
Expected Console Output	
- *Invalid name of port '`Input_Hyphen-Test`' of block '`BLOCK_System`'! Port names must not contain any special characters.*	

Expected Model Decoration
• *Port 'Input_Hyphen-Test' of block 'BLOCK_System'.*

Table 11.25: Test Model TC2-15_HyphenInSystemBlockInputPortName

TC2-16	***Slash In System Block Input Port Name***
Description	*A SysML model in which the name of the input port of the system block contains a slash.*
Error Types	SPECIAL_CHARACTER_IN_PORT_NAME
Expected Console Output	
• *Invalid name of port 'Input_Slash/Test' of block 'BLOCK_System'! Port names must not contain any special characters.*	
Expected Model Decoration	
• *Port 'Input_Slash/Test' of block 'BLOCK_System'.*	

Table 11.26: Test Model TC2-16_SlashInSystemBlockInputPortName

TC2-17	***Colon In System Block Input Port Name***
Description	*A SysML model in which the name of the input port of the system block contains a colon.*
Error Types	SPECIAL_CHARACTER_IN_PORT_NAME
Expected Console Output	
• *Invalid name of port 'Input_Colon:Test' of block 'BLOCK_System'! Port names must not contain any special characters.*	
Expected Model Decoration	
• *Port 'Input_Colon:Test' of block 'BLOCK_System'.*	

Table 11.27: Test Model TC2-17_ColonInSystemBlockInputPortName

TC2-18	***Hyphen In Flow Property Name***
Description	*A SysML model with a flow property which name contains a hyphen.*
Error Types	INVALID_FLOW_PROPERTY_NAME
Expected Console Output	

• *Invalid name of flow property '`EventFlow_Hyphen-Test`'! The flow property must be named '`EventFlow`'.*
Expected Model Decoration
• *Interface block '`Event`'.*

Table 11.28: Test Model `TC2-18_HyphenInFlowPropertyName`

TC2-19	***Slash In Flow Property Name***
Description	*A SysML model with a flow property which name contains a slash.*
Error Types	`INVALID_FLOW_PROPERTY_NAME`
Expected Console Output	
• *Invalid name of flow property '`EventFlow_Slash/Test`'! The flow property must be named '`EventFlow`'.*	
Expected Model Decoration	
• *Interface block '`Event`'.*	

Table 11.29: Test Model `TC2-19_SlashInFlowPropertyName`

TC2-20	***Colon In Flow Property Name***
Description	*A SysML model with a flow property which name contains a colon.*
Error Types	`INVALID_FLOW_PROPERTY_NAME`
Expected Console Output	
• *Invalid name of flow property '`EventFlow_Colon:Test`'! The flow property must be named '`EventFlow`'.*	
Expected Model Decoration	
• *Interface block '`Event`'.*	

Table 11.30: Test Model `TC2-20_ColonInFlowPropertyName`

TC2-21	***Hyphen In Connection Name***
Description	*A SysML model that specifies a connection whose name contains a hyphen.*
Error Types	`HYPHEN_IN_CONNECTION_NAME`
Expected Console Output	

- *Invalid name of connection '`BindingInputToInputSubA_Hyphen-Test`'! Connection names must not contain '`-`'.*
Expected Model Decoration
- *Connection '`BindingInputToInputSubA_Hyphen-Test`'.*

Table 11.31: Test Model `TC2-21_HyphenInConnectionName`

TC2-22	***Slash In Connection Name***
Description	*A SysML model that specifies a connection whose name contains a slash.*
Error Types	`SLASH_IN_CONNECTION_NAME`
Expected Console Output	
- *Invalid name of connection '`BindingInputToInputSubA_Slash/Test`'! Connection names must not contain '`/`'.*	
Expected Model Decoration	
- *Connection '`BindingInputToInputSubA_Slash/Test`'.*	

Table 11.32: Test Model `TC2-22_SlashInConnectionName`

TC2-23	***Colon In Connection Name***
Description	*A SysML model that specifies a connection whose name contains a colon.*
Error Types	`COLON_IN_CONNECTION_NAME`
Expected Console Output	
- *Invalid name of connection '`BindingInputToInputSubA_Colon:Test`'! Connection names must not contain '`:`'.*	
Expected Model Decoration	
- *Connection '`BindingInputToInputSubA_Colon:Test`'.*	

Table 11.33: Test Model `TC2-23_ColonInConnectionName`

TC2-25	***Hyphen In Property Name***
Description	*A SysML model that uses a flow property whose name contains a hyphen.*
Error Types	*Test model without any errors.*

<table>
<tr><td>• *Syntax Check successfully completed!*</td></tr>
<tr><td>Expected Console Output for UC2</td></tr>
<tr><td>• *Syntax Check successfully completed!*
• *Virtual Integration Test successfully completed!*</td></tr>
<tr><td>Expected Console Output for UC3</td></tr>
<tr><td>• *Syntax Check successfully completed!*
• *Functional Integration successfully completed!*</td></tr>
</table>

Table 11.34: Test Model `TC2-25_HyphenInPropertyName`

<table>
<tr><th>TC2-26</th><th>*Slash In Property Name*</th></tr>
<tr><td>Description</td><td>*A SysML model that uses a flow property whose name contains a slash.*</td></tr>
<tr><td>Error Types</td><td>*Test model without any errors.*</td></tr>
<tr><td colspan="2">• *Syntax Check successfully completed!*</td></tr>
<tr><td colspan="2">Expected Console Output for UC2</td></tr>
<tr><td colspan="2">• *Syntax Check successfully completed!*
• *Virtual Integration Test successfully completed!*</td></tr>
<tr><td colspan="2">Expected Console Output for UC3</td></tr>
<tr><td colspan="2">• *Syntax Check successfully completed!*
• *Functional Integration successfully completed!*</td></tr>
</table>

Table 11.35: Test Model `TC2-26_SlashInPropertyName`

<table>
<tr><th>TC2-27</th><th>*Colon In Property Name*</th></tr>
<tr><td>Description</td><td>*A SysML model that uses a flow property whose name contains a colon.*</td></tr>
<tr><td>Error Types</td><td>*Test model without any errors.*</td></tr>
<tr><td colspan="2">• *Syntax Check successfully completed!*</td></tr>
<tr><td colspan="2">Expected Console Output for UC2</td></tr>
</table>

• *Syntax Check successfully completed!* • *Virtual Integration Test successfully completed!*
Expected Console Output for UC3
• *Syntax Check successfully completed!* • *Functional Integration successfully completed!*

Table 11.36: Test Model TC2-27_ColonInPropertyName

TC3-01	***Unconnected System Input Port***
Description	*A SysML in which the input port of the system block is not connected to any input port of one subsystem block.*
Error Types	MISSING_CONNECTION_HIERARCHICAL, MISSING_CONNECTION
Expected Console Output	
• *Missing connection on block 'BLOCK_System'! Between two hierarchical levels, each input/output port of the superordinate block must be connected to at least one input/output port of the subordinate block.* • *Missing connection on port 'InputSubA' of block 'BLOCK_SubSystemA'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing connection on port 'Input' of block 'BLOCK_System'! At the same hierarchical level, each output port must be connected to at least one input port.*	
Expected Model Decoration	
• *Block 'BLOCK_System'.* • *Port 'Input' of block 'BLOCK_System'.* • *Port 'InputSubA' of block 'BLOCK_SubSystemA'.*	

Table 11.37: Test Model TC3-01_UnconnectedSystemInputPort

TC3-02	***Unconnected System Output Port***
Description	*A SysML in which the output port of the system block is not connected to any output port of one subsystem block.*
Error Types	MISSING_CONNECTION_HIERARCHICAL, MISSING_CONNECTION
Expected Console Output	

• *Missing connection on block '`BLOCK_System`'! Between two hierarchical levels, each input/output port of the superordinate block must be connected to at least one input/output port of the subordinate block.* • *Missing connection on port '`Output`' of block '`BLOCK_System`'! At the same hierarchical level, each output port must be connected to at least one input port.* • *Missing connection on port '`OutputSubB`' of block '`BLOCK_SubSystemB`'! At the same hierarchical level, each output port must be connected to at least one input port.*
Expected Model Decoration
• *Block '`BLOCK_System`'.* • *Port '`Output`' of block '`BLOCK_System`'.* • *Port '`OutputSubB`' of block '`BLOCK_SubSystemB`'.*

Table 11.38: Test Model `TC3-02_UnconnectedSystemOutputPort`

TC3-03	***Invalid Input Configuration***
Description	*A SysML model containing an input port of which the attribute '`isConjugated`' is not set to '`true`'.*
Error Types	`INVALID_CONNECTION_HIERARCHICAL`
Expected Console Output	
• *Invalid connection '`InputToInputSubA`'! Between two hierarchical levels, input ports must be connected to input ports and output ports to output ports.*	
Expected Model Decoration	
• *Connection '`InputToInputSubA`'.*	

Table 11.39: Test Model `TC3-03_InvalidInputConfiguration`

TC3-04	***Invalid Output Configuration***
Description	*A SysML model containing an output port of which the attribute '`isConjugated`' is not set to '`false`'.*
Error Types	`INVALID_CONNECTION`
Expected Console Output	
• *Invalid connection '`OutputSubAtoInputSubB`'! At the same hierarchical level, output ports must be connected to input ports.*	
Expected Model Decoration	

• *Connection ’OutputSubAtoInputSubB’.*

Table 11.40: Test Model TC3-04_InvalidOutputConfiguration

TC4-01	***Missing Top-Level Requirement***
Description	*A SysML model in which the system block is missing a requirement block.*
Error Types	MISSING_REQUIREMENT
Expected Console Output	
• *Missing requirement for block ’BLOCK_System’! Each block must have exactly one requirement.*	
Expected Model Decoration	
• *Block ’BLOCK_System’.*	

Table 11.41: Test Model TC4-01_MissingTopLevelRequirement

TC4-02	***Missing Requirement***
Description	*A SysML model with another block that is missing a requirement block.*
Error Types	MISSING_REQUIREMENT
Expected Console Output	
• *Missing requirement for block ’BLOCK_SubSystemA’! Each block must have exactly one requirement.*	
Expected Model Decoration	
• *Block ’BLOCK_SubSystemA’.*	

Table 11.42: Test Model TC4-02_MissingRequirement

TC4-03	***Multiple Requirements***
Description	*A SysML model with a block that is linked to multiple requirements.*
Error Types	MULTIPLE_REQUIREMENTS
Expected Console Output	
• *Multiple requirements for block ’BLOCK_SubSystemB’! Each block must have exactly one requirement.*	
Expected Model Decoration	

- *Block 'BLOCK_SubSystemB'.*

Table 11.43: Test Model TC4-03_MultipleRequirements

TC4-04	***Invalid Requirement Link***
Description	*A SysML model that contains a block which is not connected to its requirement block using a satisfy relationship.*
Error Types	MISSING_REQUIREMENT, MISSING_SATISFY_RELATION, INVALID_ABSTRACTION_TYPE
Expected Console Output	
- *Missing requirement for block 'BLOCK_SubSystemA'! Each block must have exactly one requirement.* - *Missing satisfy relation for requirement 'CONTRACT_SubSystemA'! Each block must be linked to its requirement by exactly one satisfy relation.* - *Invalid abstraction type of relation 'RefineSubSystemA'! Each block must be linked to its requirement by exactly one satisfy relation.*	
Expected Model Decoration	
- *Block 'BLOCK_SubSystemA'.* - *Requirement 'CONTRACT_SubSystemA'.*	

Table 11.44: Test Model TC4-04_InvalidRequirementLink

11.2.5 Generate

No test models are available for this process step.

11.2.6 Build VIT

No test models are available for this process step.

11.2.7 Elaborate

TC5-01	***Missing Delimiter In Sentence***
Description	*A SysML model that contains the requirement 'CONTRACT_System' in which a point is missing as a delimiter at the end of the assumption.*
Error Types	SYNTAX_GENERAL, EXECUTE_ELABORATION
Expected Console Output	

• *Failed to parse contract of block 'BLOCK_System' due to a syntax error! syntax error, unexpected IDENTIFIER, expecting UNIT.* • *Failed to elaborate simulation model!*
Expected Model Decoration
• *Block 'BLOCK_System'.* • *Requirement 'CONTRACT_System'.*

Table 11.45: Test Model TC5-01_MissingDelimiterInSentence

TC5-02	***Missing Time Unit***
Description	*A SysML model that contains the requirement 'CONTRACT_System' in which a time unit is missing.*
Error Types	SYNTAX_GENERAL, EXECUTE_ELABORATION
Expected Console Output	
• *Failed to parse contract of block 'BLOCK_System' due to a syntax error! syntax error, unexpected '.', expecting UNIT.* • *Failed to elaborate simulation model!*	
Expected Model Decoration	
• *Block 'BLOCK_System'.* • *Requirement 'CONTRACT_System'.*	

Table 11.46: Test Model TC5-02_MissingTimeUnit

TC5-03	***Mistake In Occurs Statement***
Description	*A SysML model that contains the requirement 'CONTRACT_System' in which an 's' is missing in one of its 'occurs' clauses.*
Error Types	SYNTAX_GENERAL, EXECUTE_ELABORATION
Expected Console Output	
• *Failed to parse contract of block 'BLOCK_System' due to a syntax error! syntax error, unexpected IDENTIFIER, expecting occurs every or occurs within or ','.* • *Failed to elaborate simulation model!*	
Expected Model Decoration	

• *Block 'BLOCK_System'.* • *Requirement 'CONTRACT_System'.*

Table 11.47: Test Model TC5-03_MistakeInOccursStatement

TC5-04	***Missing Square Bracket For Time Interval***
Description	*A SysML model that contains the requirement 'CONTRACT_System' in which a square bracket is missing in a time interval.*
Error Types	SYNTAX_GENERAL, EXECUTE_ELABORATION
Expected Console Output	
• *Failed to parse contract of block 'BLOCK_System' due to a syntax error! syntax error, unexpected UNIT, expecting '[' or ']'.* • *Failed to elaborate simulation model!*	
Expected Model Decoration	
• *Block 'BLOCK_System'.* • *Requirement 'CONTRACT_System'.*	

Table 11.48: Test Model TC5-04_MissingSquareBracketForTimeInterval

TC5-05	***Missing Bracket In Reaction Constraint***
Description	*A SysML model containing the requirement 'CONTRACT_System' in which a paranthesis is missing in its reaction constraint.*
Error Types	SYNTAX_GENERAL, EXECUTE_ELABORATION
Expected Console Output	
• *Failed to parse contract of block 'BLOCK_System' due to a syntax error! syntax error, unexpected within, expecting ')'.* • *Failed to elaborate simulation model!*	
Expected Model Decoration	
• *Block 'BLOCK_System'.* • *Requirement 'CONTRACT_System'.*	

Table 11.49: Test Model TC5-05_MissingBracketInReactionConstraint

TC5-06	***Missing Port Referenced In Contract***
Description	*A SysML model containing the requirement 'CONTRACT_System' in which an input port is referenced not defined in the model.*
Error Types	MISSING_PORT, EXECUTE_ELABORATION
Expected Console Output	
• *Missing port 'InSubB' of block 'BLOCK_SubSystemB' referenced in pattern 'InSubB occurs every 60 ms with offset [0 s, 60 ms] and jitter 5 ms.'! Each port used in a contract must be explicitly modeled in the SysML model.* • *Failed to elaborate simulation model!*	
Expected Model Decoration	
• *Block 'BLOCK_SubSystemB'.* • *Requirement 'CONTRACT_SubSystemB'.*	

Table 11.50: Test Model TC5-06_MissingPortReferencedInContract

TC5-08	***Output Port Referenced In Assumption***
Description	*A SysML model whose requirement 'CONTRACT_System' includes an output port in the assumption.*
Error Types	OUTPUT_IN_ASSUMPTION, EXECUTE_ELABORATION
Expected Console Output	
• *Invalid use of output port 'Output' of block 'BLOCK_System' in pattern 'Output occurs every 33 ms with offset [0 s, 33 ms] and jitter 5 ms.'! Assumption statements must only contain input ports.* • *Failed to elaborate simulation model!*	
Expected Model Decoration	
• *Block 'BLOCK_System'.* • *Requirement 'CONTRACT_System'.*	

Table 11.51: Test Model TC5-08_OutputPortReferencedInAssumption

11.3 Virtual Integration Test

In addition to the test models for the *Syntax Check* use case, the test suite contains test models for the step *Simulate VIT* of the *Virtual Integration Test* use case.

11.3.1 Set VIT Parameters

No test models are available for this process step.

11.3.2 Simulate VIT

TC6-01	***Inconsistent Assumption***
Description	*A SysML model with an inconsistent assumption regarding the input 'InputSubB' due to a period chosen too small.*
Error Types	PATTERN_FAILED
Expected Console Output	
• *Pattern 'InputSubB occurs every 33 ms with offset [0 s, 33 ms] and jitter 5 ms.' of block 'BLOCK_SubSystemB' failed! First event cannot occur at 50400313570 ps.*	
Expected Model Decoration	
• *Block 'BLOCK_SubSystemB'.* • *Requirement 'CONTRACT_SubSystemB'.*	

Table 11.52: Test Model TC6-01_InconsistentAssumption

TC6-02	***Missing Offset Definitions***
Description	*A SysML model with missing offset definitions.*
Error Types	PATTERN_FAILED
Expected Console Output	
• *Pattern 'InputSubB occurs every 60 ms with offset 0 s and jitter 5 ms.' of block 'BLOCK_SubSystemB' failed! First event cannot occur at 24048640492 ps.*	
Expected Model Decoration	
• *Block 'BLOCK_SubSystemB'.* • *Requirement 'CONTRACT_SubSystemB'.*	

Table 11.53: Test Model TC6-02_MissingOffsetDefinitions

TC6-03	***VIT Screencast Example***
Description	*A SysML model with a syntax error in contract 'CONTRACT_SubSystemA' and an inconsistent offset definition in contract 'CONTRACT_SubSystemB'.*
Error Types	SYNTAX_GENERAL, EXECUTE_ELABORATION, PATTERN_FAILED
Expected Console Output	

• *Failed to parse contract of block 'BLOCK_SubSystemA' due to a syntax error! syntax error, unexpected within, expecting ')'.* • *Failed to elaborate simulation model!* • *Pattern 'InputSubB occurs every 60 ms with offset [0 s, 20 ms] and jitter 5 ms.' of block 'BLOCK_SubSystemB' failed! First event cannot occur at 50400313570 ps.*
Expected Model Decoration
• *Block 'BLOCK_SubSystemA'.* • *Requirement 'CONTRACT_SubSystemA'.* • *Block 'BLOCK_SubSystemB'.* • *Requirement 'CONTRACT_SubSystemB'.*

Table 11.54: Test Model `TC6-03_VITScreencastExample`

TC6-04	***Inconsistent Slice Configuration***
Description	*A SysML model whose annotated timing specifications cause the generation of an inconsistent slice configuration.*
Error Types	EXECUTE_ELABORATION, GENERATOR_INCONSISTENCY
Expected Console Output	
• *Failed to find a consistent configuration of slices for pattern 'A: InputSubB occurs every 33ms with offset [10,43]ms and jitter 15ms. OtherInputSubB occurs every 33ms with offset [0,33]ms and jitter 10ms. G: OutputSubB occurs every 33ms with offset [0,33]ms and jitter 10ms. Age(InputSubB,OutputSubB) within [0,60]ms. Age(OtherInputSubB,OutputSubB) with in [0,60]ms.'!* • *Failed to elaborate simulation model!*	

Table 11.55: Test Model `TC6-04_InconsistentSliceConfiguration`

11.3.3 Show VCD

No test models are available for this process step.

12 Conclusion

The present report documents the results of the MULTIC Tooling project, where a prototypical but feature rich design and analysis tool for the MULTIC design methodology has been developed.

The report contains two parts. Firstly, it discusses the concepts for the construction of generator and monitor automata from timing specifications. It defines syntax and semantics of the so-called MULTIC Timing Specification Language (MTSL), which allows expressing timing properties in terms of contracts. While semantics is defined in terms of languages, the construction of generators and monitors for these specifications calls for an operational semantics. To this end, we rephrase the formal definition of a class of hybrid automata. For implementation purposes, we also present an implementation oriented language for specifying a sub class of these automata. Based on this intermediate language, we finally discuss generators and monitors for the elements of MTSL.

The second part discusses architecture and implementation details of the developed MULTIC-Tool. The MULTIC-Tool integrates the paradigms that have been elaborated in the predecessor project MULTIC into a tool that can be applied also on large models in an industrial environment. It is based on Eclipse Papyrus for capturing or importing of system specification models in SysML and allows equipping them with timing contracts. The tool furthermore performs an automatic timing specification consistency check through execution of a VIT. In the final step, the system specification can be exported to SystemC, where arbitrary behavior can be added into the generated modules and checked against the timing contracts. Potential timing contract violations are visualized as a trace and provide (in the form or a counterexample) support for localizing the behavior or behavior chain that caused this violation.

This second part is complemented by useful information for users of the tool about the supported set of input models, and lists possible errors that may occur when the underlying analyses are executed. Finally, it provides information about the test models that are delivered together with the tool, and lists the expected results and errors that will occur when analysis is performed on these models.

The tool prototype is provided pre-installed in a virtual box appliance and ready to use. It also contains many sample models. The reader is encouraged to have a look at the two video tutorials in German and English. The first deals wit the correct SysML modeling and the second video explains how to perform a *Virtual Integration Test*.

References

[1] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*, 109:43–56, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[2] Andreas Baumgart, Philipp Reinkemeier, Achim Rettberg, Ingo Stierand, Eike Thaden, and Raphael Weber. A model-based design methodology with contracts to enhance the development process of safety-critical systems. In *Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems*, SEUS'10, pages 59–70, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Eckard Böde, Mathias Büker, Werner Damm, Günter Ehmen, Martin Fränzle, Sebastian Gerwinn, Thomas Goodfellow, Kim Grüttner, Bernhard Josko, Björn Koopmann, Thomas Peikenkamp, Frank Poppen, Philipp Reinkemeier, Michael Siegel, and Ingo Stierand. Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC). Technical Report 302, October 2017.

[4] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde. Boosting Re-use of Embedded Automotive Applications Through Rich Components. In *Proceedings of Foundations of Interface Technologies (FIT'05)*, 2005.

[5] Henning Dierks, Alexander Metzner, and Ingo Stierand. Efficient model-checking for real-time task networks. In *6th International Conference on Embedded Software and Systems*, May 2009.

[6] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08)*. IEEE Computer Society, 2008.

[7] Thomas A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 278–292, Washington, DC, USA, 1996. IEEE Computer Society.

[8] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Implementation of End-to-End Latency Analysis for Component-Based Multi-Rate Real-Time Systems in Rubus-ICE. In *Proceedings of the 9th IEEE International Workshop on Factory Communication Systems (WFCS'12)*, pages 165–168, May 2012.

[9] Object Management Group. *OMG Systems Modeling Language^{TM}(OMG SysML)*, September 2015. Version 1.4.

[10] TIMMO-2-USE Partners. Language Syntax, Semantics, Metamodel V2. TIMMO-2-USE Deliverable D11, TIMMO-2-USE Project, August 2012.

[11] SPEEDS Project. `http://www.accellera.org`.

[12] Thomas Strathmann and Jens Oehlerking. Verifying properties of an electro-mechanical braking system. In *2nd Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH 2015)*, April 2015.

[13] Homepage of the Accellera Systems Initiative. `http://www.accellera.org`.

Bisher in der FAT-Schriftenreihe erschienen (ab 2014)

Nr.	Titel
263	Laserstrahlschweißen von Stahl an Aluminium mittels spektroskopischer Kontrolle der Einschweißtiefe und erhöhter Anbindungsbreite durch zweidimensional ausgeprägte Schweißnähte, 2014
264	Entwicklung von Methoden zur zuverlässigen Metamodellierung von CAE Simulations-Modellen, 2014
265	Auswirkungen alternativer Antriebskonzepte auf die Fahrdynamik von PKW, 2014
266	Entwicklung einer numerischen Methode zur Berücksichtigung stochastischer Effekte für die Crashsimulation von Punktschweißverbindungen, 2014
267	Bewegungsverhalten von Fußgängern im Straßenverkehr - Teil 1, 2014
268	Bewegungsverhalten von Fußgängern im Straßenverkehr - Teil 2, 2014
269	Schwingfestigkeitsbewertung von Schweißnahtenden MSG-geschweißter Feinblechstrukturen aus Aluminium, 2014
270	Physiologische Effekte bei PWM-gesteuerter LED-Beleuchtung im Automobil, 2015
271	Auskunft über verfügbare Parkplätze in Städten, 2015
272	Zusammenhang zwischen lokalem und globalem Behaglichkeitsempfinden: Untersuchung des Kombinationseffektes von Sitzheizung und Strahlungswärmeübertragung zur energieeffizienten Fahrzeugklimatisierung, 2015
273	UmCra - Werkstoffmodelle und Kennwertermittlung für die industrielle Anwendung der Umform- und Crash-Simulation unter Berücksichtigung der mechanischen und thermischen Vorgeschichte bei hochfesten Stählen, 2015
274	Exemplary development & validation of a practical specification language for semantic interfaces of automotive software components, 2015
275	Hochrechnung von GIDAS auf das Unfallgeschehen in Deutschland, 2015
276	Literaturanalyse und Methodenauswahl zur Gestaltung von Systemen zum hochautomatisierten Fahren, 2015
277	Modellierung der Einflüsse von Porenmorphologie auf das Versagensverhalten von Al-Druckgussteilen mit stochastischem Aspekt für durchgängige Simulation von Gießen bis Crash, 2015
278	Wahrnehmung und Bewertung von Fahrzeugaußengeräuschen durch Fußgänger in verschiedenen Verkehrssituationen und unterschiedlichen Betriebszuständen, 2015
279	Sensitivitätsanalyse rollwiderstandsrelevanter Einflussgrößen bei Nutzfahrzeugen – Teil 3, 2015
280	PCM from iGLAD database, 2015
281	Schwere Nutzfahrzeugkonfigurationen unter Einfluss realitätsnaher Anströmbedingungen, 2015
282	Studie zur Wirkung niederfrequenter magnetischer Felder in der Umwelt auf medizinische Implantate, 2015
283	Verformungs- und Versagensverhalten von Stählen für den Automobilbau unter crashartiger mehrachsiger Belastung, 2016
284	Entwicklung einer Methode zur Crashsimulation von langfaserverstärkten Thermoplast (LFT) Bauteilen auf Basis der Faserorientierung aus der Formfüllsimulation, 2016
285	Untersuchung des Rollwiderstands von Nutzfahrzeugreifen auf realer Fahrbahn, 2016

286 χMCF - A Standard for Describing Connections and Joints in the Automotive Industry, 2016

287 Future Programming Paradigms in the Automotive Industry, 2016

288 Laserstrahlschweißen von anwendungsnahen Stahl-Aluminium-Mischverbindungen für den automobilen Leichtbau, 2016

289 Untersuchung der Bewältigungsleistung des Fahrers von kurzfristig auftretenden Wiederübernahmesituationen nach teilautomatischem, freihändigem Fahren, 2016

290 Auslegung von geklebten Stahlblechstrukturen im Automobilbau für schwingende Last bei wechselnden Temperaturen unter Berücksichtigung des Versagensverhaltens, 2016

291 Analyse, Messung und Optimierung des Ventilationswiderstands von Pkw-Rädern, 2016

292 Innenhochdruckumformen laserstrahlgelöteter Tailored Hybrid Tubes aus Stahl-Aluminium-Mischverbindungen für den automobilen Leichtbau, 2017

293 Filterung an Stelle von Schirmung für Hochvolt-Komponenten in Elektrofahrzeugen, 2017

294 Schwingfestigkeitsbewertung von Nahtenden MSG-geschweißter Feinbleche aus Stahl unter kombinierter Beanspruchung, 2017

295 Wechselwirkungen zwischen zyklisch-mechanischen Beanspruchungen und Korrosion: Bewertung der Schädigungsäquivalenz von Kollektiv- und Signalformen unter mechanisch-korrosiven Beanspruchungsbedingungen, 2017

296 Auswirkungen des teil- und hochautomatisierten Fahrens auf die Kapazität der Fernstraßeninfrastruktur, 2017

297 Analyse zum Stand und Aufzeigen von Handlungsfeldern beim vernetzten und automatisierten Fahren von Nutzfahrzeugen, 2017

298 Bestimmung des Luftwiderstandsbeiwertes von realen Nutzfahrzeugen im Fahrversuch und Vergleich verschiedener Verfahren zur numerischen Simulation, 2017

299 Unfallvermeidung durch Reibwertprognosen, 2017

300 Thermisches Rollwiderstandsmodell für Nutzfahrzeugreifen zur Prognose fahrprofilspezifischer Energieverbräuche, 2017

301 The Contribution of Brake Wear Emissions to Particulate Matter in Ambient Air, 2017

302 Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC), 2017

303 Experimentelle Untersuchung des Einflusses der Oberflächenbeschaffenheit von Scheiben auf die Kondensatbildung, 2017

304 Der Rollwiderstand von Nutzfahrzeugreifen unter realen Umgebungsbedingungen, 2018

305 Simulationsgestützte Methodik zum Entwurf intelligenter Energiesteuerung in zukünftigen Kfz-Bordnetzen, 2018

306 Einfluss der Kantenbearbeitung auf die Festigkeitseigenschaften von Stahl-Feinblechen unter quasistatiscl und schwingender Beanspruchung, 2018

307 Fahrerspezifische Aspekte beim hochautomatisierten Fahren, 2018

308 Der Rollwiderstand von Nutzfahrzeugreifen unter zeitvarianten Betriebsbedingungen, 2018

309 Bewertung der Ermüdungsfestigkeit von Schraubverbindungen mit gefurchtem Gewinde, 2018

310 Konzept zur Auslegungsmethodik zur Verhinderung des selbsttätigen Losdrehens bei Bauteilsystemen im Leichtbau, 2018

311 Experimentelle und numerische Identifikation der Schraubenkopfverschiebung als Eingangsgröße für eine Bewertung des selbsttätigen Losdrehens von Schraubenverbindungen, 2018

312 Analyse der Randbedingungen und Voraussetzungen für einen automatisierten Betrieb von Nutzfahrzeuge im innerbetrieblichen Verkehr, 2018

313 Charakterisierung und Modellierung des anisotropen Versagensverhaltens von Aluminiumwerkstoffen für die Crashsimulation, 2018

314 Definition einer „Äquivalenten Kontakttemperatur“ als Bezugsgröße zur Bewertung der ergonomischen Qualität von kontaktbasierten Klimatisierungssystemen in Fahrzeugen, 2018

315 Anforderungen und Chancen für Wirtschaftsverkehre in der Stadt mit automatisiert fahrenden E-Fahrzeugen (Fokus Deutschland), 2018

316 MULTIC-Tooling, 2019

Impressum

Herausgeber	FAT Forschungsvereinigung Automobiltechnik e.V. Behrenstraße 35 10117 Berlin Telefon +49 30 897842-0 Fax +49 30 897842-600 www.vda-fat.de
ISSN	2192-7863

Verband der Automobilindustrie e.V. (VDA)
Behrenstraße 35, 10117 Berlin
www.vda.de
Twitter @VDA_online

Forschungsvereinigung Automobiltechnik e.V. (FAT)
Behrenstraße 35, 10117 Berlin
www.vda.de/fat

VDA Verband der Automobilindustrie

FAT Forschungsvereinigung Automobiltechnik