Modelling and Analysis of Real-Time Coordination Patterns

Stephanie Kemper

Modelling and Analysis of Real-Time Coordination Patterns

Proefschrift

ter verkrijging van

de graad van Doctor aan de Universiteit Leiden

op gezag van Rector Magnificus prof. mr. P.F. van der Heijden,

volgens besluit van het College voor Promoties

te verdedigen op dinsdag 20 december 2011

klokke 11:15 uur

door

Stephanie Kemper

geboren te Bremerhaven, Duitsland, in 1979

The work in this thesis has been carried out at the Centrum Wiskunde & Informatica (CWI), and under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was partially funded by the Netherlands Organisation for Scientific Research (NWO) under the BRICKS project (Basic Research in Informatics for Creating the Knowledge Society).

Cover design by Nils Kemper.
Printed and published by Boxpress BV ||Proefschriftmaken.nl

ISBN: 978-90-8891-360-0
IPA Dissertation Series 2011-24

Contents

Chapter 1

Introduction

Embedded software systems are nowadays found everywhere: in smartphones, in cars and aeroplanes, even in coffee machines. Naturally, with an increased number of software systems comes an increased number of observable system failures. While for some systems, a system failure is not dangerous at all but merely annoying—for example, if no coffee can be prepared—failure of other systems has safety critical or even life threatening consequences—for example, an airbag failure or an aeroplane crash. For this reason, it is important to verify that the system works correctly and safely before it is being put into operation.

Verification of such systems is a difficult task by itself already, but two features of present-day software systems complicate this task even more. First, due to the increasing size of software systems, they are developed in a modular, *component-based* way, such that the final system can be distributed over several (logical and/or physical) locations. Second, to be able to truthfully express real-life situations, and to specify lower and upper safety bounds on the occurrence of events—to ensure that things happen at the right moment—systems involve handling of *real-time*. We now look at these two aspects in a little more detail.

Component-based software engineering and system design amounts to constructing large systems by composing individual components. That is, (typically at beginning of the design phase) the behaviour of the system to be developed is split into logical units, each encapsulating a certain behaviour or aspect of the system. These units can then be developed in parallel and independently of each other. In the end, the final system is obtained by composing these individual components. Naturally, the correctness and safety of concurrent systems developed in this way not only depends on the correctness and safety of the individual components, but also (even more) on correct inter-component communication actions: even if the components are correct—that means, they correctly implement the intended behaviour of the respective logical unit—the final system cannot work as intended if the components exchange inappropriate information or communicate with the wrong partner. For example, consider an aeroplane controller that intends to request the status of the fuel level sensor, but instead communicates with the cabin pressure sensor. In case the cabin pressure is fine, but the fuel level is low, this wrong connection of communication partners can have lethal consequences.

Another problem arises from the fact that when designing a new system, not all components are/need to be developed from scratch. Instead, component-based software engineering allows for reusing components from other systems, and for simple or recurring tasks, even off-the-shelf components exist. Unfortunately, such reused components are often only available as black boxes. As a consequence, to correctly interlink the components, component connectors are required that provide exogenous coordination, i.e., coordination from without [Arb98]. As such, these component connectors require true concurrency in time, which combines synchrony and asynchrony, to express complex coordination patterns.

The major advantages of the component-based approach are the possibility to develop the system parts in parallel and deal with larger systems (scalability), and the increased flexibility to exchange or modify components of the system, without having to modify or even inform/touch the other components (modularity). The major challenge for the verification of component-based systems is to adapt and scale the verification techniques to large and distributed systems.

To guarantee correct behaviour of the system, in addition to the above, the communication between components not only needs to happen correctly with respect to exchanged information and involved components, but equally important needs to happen *at the right time*. For example, a warning message indicating a low fuel level is useless if it is displayed only after the fuel tank is completely empty. Yet, “at the right time” does typically not mean “as fast as possible”, but involves both upper and lower time bounds. For example, an airbag must not deploy immediately, but with the correct delay (a couple of milliseconds), to properly slow down the forward movement of the occupant at the right moment.

To develop component-based real-time systems which are correct—that means, behave as expected—and safe—that means, nothing bad can ever happen—essentially two things are needed. First, a formal model to describe the system components. This formal model should be powerful enough to faithfully describe all aspects of the system, and yet simple enough to allow being used in industrial practice by non-experts in formal methods . Second, formal methods are needed to analyse the system model before it is being put into operation, and to verify that it satisfies certain requirements.

In this thesis, we aim at solving the problem described above. To this end, we present formal models to specify real-time coordination patterns, and define formal methods to analyse and verify these formal models, as well as formalisms to further increase the manageable system size. We present our tool implementation featuring a graphical editor, that supports users in the development and design process.

1.1 Contents and Structure of this Thesis

The contents of this thesis are structured as follows.

Chapter 2 (System Models) We start with the formal models. In this Chapter, we define a series of three automata-based system models. We start with the (among the three) simplest model of Timed Automata [AD94, Alu99], followed by Timed Constraint Automata [ABdBR07, Kem11] and Timed Network Automata [Kem10]. The formal models are presented based on increasing modelling power with respect to communication. For each model, we discuss advantages and disadvantages, including for example what kind of constraints on (inter-component) communication can be imposed with the respective model, and what kind of constraints cannot be imposed. Moreover, for each of the formal models, we define a formal semantics, and present the product construction needed to compose larger systems.

Chapter 3 (SAT-based Verification) In this Chapter, we show how to use formal methods to analyse and verify properties of the system models from Chapter 2. We start by defining a translation into propositional logic with linear arithmetic for each of the system models. This formula representation allows to apply well-established model checkers to examine the systems and verify certain properties. In particular, we show how to apply the technique of Bounded Model Checking [CBRZ01, BCC$^+$03] to our representation, show how to express certain common-type properties, and we discuss correctness and completeness issues for Bounded Model Checking. In the end of Chapter 3, we provide an in-depth discussion about other possible translations, what influence these (would) have on the efficiency of verification, and in this way motivate and justify our design decisions.

Chapter 4 (Abstraction Refinement) In this Chapter, we adapt the principles of Abstraction Refinement [CGJ$^+$03, HJMM04], such that they can be used in combination with the formal methods presented in Chapter 3. These techniques allow to increase the size of systems to be analysed, by removing system parts which are considered irrelevant for the current examination or property under test. In particular, we consider a variant of Counterexample Guided Abstraction Refinement [CGJ$^+$03], where information from so-called spurious counterexamples (counterexamples to the property under test that only exist in the reduced system) is used to identify the reason why verification has failed. Such information is extracted from spurious counterexamples with the help of Craig interpolants [Cra57]. We briefly refresh the principles of Craig interpolants, show what kind of information can be obtained from these, how to modify the input to maximise the information, and in the end of the Chapter present techniques to prevent the spurious counterexample from happening again.

Chapter 5 (Tool Development and Application to Case Studies) The theoretical results obtained in this thesis have been implemented and integrated into an existing tool suit for component-based systems. Chapter 5 is devoted to tool development. We start by presenting the details of our implementation in the context of the existing tool suite. We then show how to actually use the tool, by giving a little workflow example, which can also be seen as a “getting started” tutorial. Finally, we show the applicability and performance of our approach and tool on two small case studies.

Chapter 6 (Conclusions) In this Chapter, we wrap up the results, and discuss some directions for future research.

Appendix In the Appendix, we give correctness results for the formula representation presented in Chapter 3, and the abstraction function presented in Chapter 4.

1.2 Origin of Material and Main Contributions

Some parts of the research presented in this thesis have been published previously, or have been included here for completeness. Other parts contain new, original research which has not been presented elsewhere. We now give an overview for each of the Chapters, and sum up the main contributions.

Chapter 2 (System Models)

Timed Automata have first been introduced in [AD94, Alu99]. Since they come in many different variants (most of them with only subtle differences), in Section 2.2 we have outlined the exact nature of Timed Automata that we use in this thesis, without changing the formal model itself.

In Section 2.3, we extend the formal model of Timed Constraint Automata from [ABdBR07] (which has also been used in [Kem11]) with location memory. Location memory for (Untimed) Constraint Automata [ABRS04] has been defined in [PSHA09], but this is the first work on defining a formal syntax (including a product construction) and semantics for the combination of Constraint Automata with time *and* data. The new Timed Constraint Automaton model is an extension of the model in [ABdBR07] and therefore extends the set of coordination patterns that can be expressed, in that coordination patterns can not only reason about data values received in the current step, but also about data values received in previous steps. Timed Constraint Automata can—to a certain extend—be seen as an extension of Timed Automata [AD94, Alu99] with data.

In Section 2.4, we extend the formal model of Timed Network Automata, as presented in [Kem10], with memory cells and concrete data values, and define a formal syntax and semantics. In particular, we define a product construction that uses data variables to preserve information about data values contained in memory cells. Timed Network Automata were inspired by the three-colouring idea for $\mathcal{R}$eo networks [CCA07], but lift the idea of considering environmental constraints to the automaton level. Since the extended Timed Network Automata defined in this thesis features the same concepts for data handling as Timed Constraint Automata (that is, ports and memory cells), they can be seen as an extension of Timed Constraint Automata with environmental constraints.

Summarising, this Chapter contains the following main contributions:

Contribution 1: memory cell extension of Timed Constraint Automata, with corresponding product construction and formal semantics

Contribution 2: definition of Timed Network Automata, with formal syntax, semantics and product construction extension of the basic Timed Network Automaton model with memory cells and data-awareness, with corresponding formal syntax, semantics and product construction

Chapter 3 (SAT-based Verification)

The formula representation of Timed Automata presented in Section 3.1.2 has been published in [KP07]. Subsequent to the extensions of Timed Constraint Automata in Section 2.3 and Timed Network Automata in Section 2.4, in Sections 3.1.3 and 3.1.4 we extend the formula representations (in propositional logic with linear arithmetic) of Timed Constraint Automata (presented in [Kem11]) and Timed Network Automata (presented in [Kem10]) accordingly. After that, we discuss completeness issues of Bounded Model Checking in Section 3.2, which have only been briefly sketched in [Kem11], and extend the correctness proof to the extended representation (note that the actual proofs are contained in the appendix). Finally, we give an in-depth discussion about the quality of the representation with respect to speed of verification, which has not been published before.

Summarising, this Chapter contains the following main contributions:

Contribution 1: application of SAT-based verification in the process of component-based software engineering and to formal models of component-based software

Contribution 2: adaptation of the formula representation of Timed Constraint Automata and Timed Network Automata to the extended models of Chapter 2

Contribution 3: in-depth discussion of the quality of representation, including advantages and disadvantages of different translation options

Chapter 4 (Abstraction Refinement)

The abstraction function we present in Section 4.1 is essentially equivalent to the abstraction function presented in [Kem11], but we show that it works uniformly for both Timed Automata and Timed Constraint Automata, and that correctness is preserved for both formal models (again, the actual proof is contained in the appendix).

We restate the principles of Craig interpolation [Cra57] for the sake of completeness. After that, we provide a detailed discussion about the expressiveness of interpolants, the possibilities to influence expressiveness, and refinement options and heuristics, all of which has only been briefly and vaguely sketched in [KP07, Kem11].

Summarising, this Chapter contains the following main contributions:

Contribution 1: application of abstraction refinement in the process of component-based software engineering and to formal models of component-based software

Contribution 2: generalisation of the abstraction function and correctness results

Contribution 3: in-depth discussion of interpolation and refinement, to provide a deep understanding of the underlying mechanisms

Chapter 5 (Tool Development and Application to Case Studies)

We present the implementation that, in the context of this thesis, has been developed as part of the Extensible Coordination Tools [ECT],[1] a tool suit for component-based systems. In Section 5.1, we present the details of the implementation in a comprehensible way, which makes it easy to understand the overall structure and functioning, and provides the basis for future extensions of the tool. Section 5.2 demonstrates the applicability of the approach, by giving a workflow example/"getting started" tutorial. We finish with two little case studies.

Apart from Section 5.1.1 (which provides the overall context for the description of our implementation) and Section 5.3.1 (which has been published as part of [Kem11]), all results presented in this Chapter describe original work that has not been published before.

This Chapter contains the following main contribution:

Contribution: extensive tool-support for modelling and analysis of real-time coordination patterns implemented with Timed Constraint Automata

[1]See `http://reo.project.cwi.nl/`

Chapter 2

System Models

When designing real-time systems, one—if not *the*—major design decision is the choice of time domain. In [AD94], the authors phrased this problem as finding the answer to the question "What is the nature of time?".

Reasoning on a *discrete* time scale allows time values in the domain of natural numbers $\mathbb{N}$, that means events can happen at equidistant points in time, or multiples thereof. Reasoning on a *continuous* (or *dense*) time scale allows time values in the domain of real numbers $\mathbb{R}_{\geq 0}$, that means events can happen at any positive (to express that time "starts at 0" and does not "run backwards") real-valued point in time. While discrete time is very close to physical hardware implementation of real-time systems (for example, a discrete time step can be taken with every trailing edge), continuous time is a more realistic approach, in that events do not always happen at integer-valued times, even if the base value for "one time unit" is small.

To use a discrete time scale, the occurrence times of events would either have to be restricted to integer values, or be approximated with the "closest" integer value. The former is unrealistic, the latter comprises a serious information loss with respect to sequential order and (a)synchrony of events. Therefore, in this thesis, we choose continuous time as the time domain `Time` of real-time systems, that means,

$$\texttt{Time} = \mathbb{R}_{\geq 0}.$$

In the remainder of this chapter, we introduce the state-based system models that we use to model real-time systems. We begin by introducing some general notions of real-time systems in Section 2.1. In subsequent Sections, we introduce Timed Automata (TA, Section 2.2), Timed Constraint Automata (TCA, Section 2.3), and Timed Network Automata (TNA, Section 2.4). We conclude each Section with a discussion about the advantages and disadvantages of the respective system model. We conclude the Chapter in Section 2.5 with a clear identification of the parts that represent original work in this thesis, and parts that are based on previous results.

2.1 Preliminaries

In this section, we introduce some general notions and concepts for modelling real-time systems.

Notation 2.1.1 (Operator Precedence). To reduce the number of parenthesis, we establish the following precedence rules. Among logical operators, $\neg$ has a higher precedence than $\{\wedge, \vee\}$, which have a higher precedence than $\{\rightarrow, \leftrightarrow\}$. Among arithmetical operators, - (unary minus) has a higher precedence than $\{+, -\}$ (binary minus), which have a higher precedence than $\{<, \leq, =, \geq, >\}$.

For operators with the same precedence, we may still have to add parenthesis, for example, we need to make clear whether $a{\wedge}b{\vee}c$ means $(a{\wedge}b){\vee}c$ or $a{\wedge}(b{\vee}c)$. We do not need to define precedence rules between arithmetical and logical operators though, since it is clear from the context which variables or constants the operators bind to. For example, for integer variables x_1, x_2, x_3, x_4, it is clear that $x_1{=}x_2{\wedge}x_3{\geq}x_4$ means $(x_1{=}x_2){\wedge}(x_3{\geq}x_4)$.

2.1.1 Time

As explained above, we work with a dense time domain $\texttt{Time}{=}\mathbb{R}_{\geq 0}$. To measure the passage of time, each model of a real-time systems is equipped with a finite set of real-valued clocks $\mathcal{X}$. All clocks evolve with the same slope 1 (they all "run with the same speed"), i.e., after d time units have passed, the value of each clock has advanced by d. Components of the system may be associated with clock constraints, which restrict the behaviour of the system. The syntax of clock constraints is defined as follows

Definition 2.1.2 (Clock Constraint). Let $\mathcal{X}$ be a finite set of real-valued variables, called clocks. *Clock constraints* $cc{\in}\mathcal{CC}(\mathcal{X})$ *over* $\mathcal{X}$ are defined as follows:

$$cc ::= \texttt{true} \mid x{\sim}c \mid x{-}y{\sim}c \mid cc_1 \wedge cc_2,$$
$$\text{with } x, y{\in}\mathcal{X}, c{\in}\mathbb{Z}, \text{ and } \sim\, \in\{<, \leq, =, \geq, >\}$$

We write $\mathcal{X}|_{cc}{\subseteq}\mathcal{X}$ to denote the set of clocks that occur in a clock constraint cc.

Remark 2.1.3 (Time Domain in Clock Constraints). Though clocks are real-valued, we need to restrict the domain of constants in clock constraints, in order to obtain/preserve decidability results (cf. [AD94]). For example, with real-valued constants, reachability would become undecidable.

To preserve decidability results, it would be enough to restrict the domain to $\mathbb{Q}$ (rational numbers) though. Yet, for simplicity, we further restrict it to integer numbers. This does not reduce expressiveness (with respect to rational numbers): if clock constraints involve rational constants, we can multiply all constants by the least common multiple of their denominators to obtain clock constraints with integer constants only.

The validity ("semantics") of clock constraints is evaluated under a certain valuation of the clock variables.

Definition 2.1.4 (Clock Valuation). Let $\mathcal{X}$ be a finite set of clocks, $X{\subseteq}\mathcal{X}$, $x{\in}\mathcal{X}$ and $t{\in}\texttt{Time}$. A *clock valuation* $\nu{\in}\mathcal{V}(\mathcal{X})$ *over* $\mathcal{X}$ is a mapping $\nu{:}\mathcal{X}{\rightarrow}\texttt{Time}$, assigning to each clock $x{\in}\mathcal{X}$ an element from the time domain $\texttt{Time}$, its current value. The *restriction* $\nu|_X$ *of* ν *from* $\mathcal{X}$ *to* X is a valuation that agrees with ν on clocks $x{\in}X$, and is undefined otherwise, that means $\nu|_X(x){=}\nu(x)$ iff $x{\in}X$, and $\nu|_X(x){=}$undefined otherwise.

We may write $\mathcal{V}$ if $\mathcal{X}$ is clear from the context.

We use $\models$ for the standard satisfaction relation on clock constraints. For example, $\nu{\models}(x{\sim}c)$ iff $\nu(x){\sim}c$, ${\sim}{\in}\{<,\leq,=,\geq,>\}$.

To model the semantics of real-time systems, we need the following operations on valuations.

Definition 2.1.5 (Timeshift, Update). Let $\mathcal{X}$ be a finite set of clocks, $x,y{\in}\mathcal{X}$, $t{\in}\texttt{Time}$ and $\nu{\in}\mathcal{V}(\mathcal{X})$.

The *timeshift operation* $\nu{+}t$ (or simply *timeshift*) increases (with respect to ν) the values of all clocks simultaneously by the same amount of time t, that means, $(\nu{+}t)(x){=}\nu(x){+}t$ for all $x{\in}\mathcal{X}$.

An *update map* $\lambda{\in}\Lambda(\mathcal{X})$ *over* $\mathcal{X}$ [AM04] is a mapping $\lambda{:}\mathcal{X}{\rightarrow}(\mathcal{X}{\cup}\mathbb{N})$. The *update operation* $\nu[\lambda]$ (or simply *update*) modifies the values of clocks under valuation ν, by either setting them to the value of another clock or to a natural number, according to the update map λ (λ is the identity for clocks not meant to be modified). That is, $\nu[\lambda](x){=}\nu(y)$ iff $\lambda(x){=}y$,[1] and $\nu[\lambda](x){=}n$ iff $\lambda(x){=}n{\in}\mathbb{N}$.

We do not allow negation on clock constraints (cf. [Alu99, AM04]), which results in clock constraints being *convex*.

Remark 2.1.6 (Convexity of Clock Constraints). Clock constraints defined according to Definition 2.1.2 are *convex* under timeshift and update (Definition 2.1.5), for any clock valuation $\nu{\in}\mathcal{V}(\mathcal{X})$ (Definition 2.1.4).

Intuitively, convexity of clock constraints means that if the value of a clock satisfies the same clock constraint under two different valuations, then it also satisfies the clock constraint for all valuations "in between". Formally: for a clock constraint cc and two valuations ν and ν' over clock $x{\in}\mathcal{X}|_{cc}$, such that $\nu(x){<}\nu'(x)$, if $\nu{\models}cc$ and $\nu'{\models}cc$, then $\nu''{\models}cc$ for all ν'' with $\nu(x){<}\nu''(x){<}\nu'(x)$.

Convexity of clock constraints is an important property used for efficient representation (and verification) of real-time systems in Chapter 3.

Note that we do not impose any semantic constraints on clock constraints. For example, they may "overlap" on a clock, like $(x{\leq}4)\wedge(x{\geq}3)$. Yet, due to convexity, this simply reduces the number of satisfying valuations.

[1]In case of an update $\lambda(x){=}y$, $\lambda(y){=}n$, the equation $\nu[\lambda](x){=}\nu(y)$ ensures that x is assigned the value of y *before* y is updated.

2.1.2 Data

Communication in real-time systems may involve exchange of data values. We assume a global (possibly infinite but) countable data domain $\mathcal{D}ata$, with a special element $\bot{\in}\mathcal{D}ata$ representing "no data", which we use in Chapter 3 to explicitly represent *absence of data*. Real-time systems use a finite set of ports $\mathcal{P}$, through which they exchange data values with the environment. Further, they can make use of a finite set of data variables $\mathcal{D}$. The intended idea is that ports are used to exchange data values with the environment, while data variables are used to store and exchange data values within the real-time system.

To restrict the admissible data values to be exchanged, components of real-time systems may be associated with data constraints. These restrict the behaviour of the system, by reasoning about the data values exchanged through ports, or stored in data variables. The syntax of data constraints is defined as follows.

Definition 2.1.7 (Data Constraint). Let $\mathcal{P}$ be a finite, nonempty set of ports, $\mathcal{D}$ a finite, nonempty set of data variables, $\mathcal{D}ata$ a data domain. *Data constraints* $dc{\in}DC(\mathcal{P}{\cup}\mathcal{D})$ *over* $\mathcal{P}$ *and* $\mathcal{D}$ are defined as

$$dc ::= \texttt{true} \mid \mathbf{d}{=}\mathbf{d}' \mid dc_1{\wedge}dc_2 \mid \neg dc, \text{ with } \mathbf{d},\mathbf{d}'{\in}\mathcal{P}{\cup}\mathcal{D}{\cup}\mathcal{D}ata$$

If there exists a total order $\leqslant$ on $\mathcal{D}ata\backslash\bot$, we also allow data constraints of the form

$$dc ::= \mathbf{d}{\leqslant}\mathbf{d}', \text{ with } \mathbf{d},\mathbf{d}'{\in}\mathcal{P}{\cup}\mathcal{D}{\cup}\mathcal{D}ata\backslash\bot$$

If on top of $\leqslant$ there exists an operation $+$ (addition) on $\mathcal{D}ata\backslash\bot$,[2] we also allow data constraints of the form

$$\begin{aligned}&dc ::= \mathbf{D}{\sim}\mathbf{D}', \text{ with } \sim\in\{=,\leqslant\}, \text{ and}\\ &\mathbf{D} ::= \mathbf{d} \mid \mathbf{D_1}{+}\mathbf{D_2} \mid (\mathbf{D_1}{-}\mathbf{D_2}), \text{ with } \mathbf{d},\mathbf{d}'{\in}\mathcal{P}{\cup}\mathcal{D}{\cup}\mathcal{D}ata\backslash\bot\end{aligned}$$

We write $\mathcal{D}|_{dc}{\subseteq}\mathcal{D}$ to denote the set of data variables that occur in a data constraint dc. Equivalently, $\mathcal{P}|_{dc}{\subseteq}\mathcal{P}$ denotes the set of ports that occur in dc, and $\mathcal{D}ata|_{dc}{\subseteq}\mathcal{D}ata$ denotes the set of data values that occur in dc.

We may write $DC(\mathcal{P}{,}\mathcal{D})$ instead of $DC(\mathcal{P}{\cup}\mathcal{D})$, and we use $DC(\mathcal{P})$ as a shorthand for $DC(\mathcal{P},\emptyset)$, equivalently we use $DC(\mathcal{D})$ as a shorthand for $DC(\emptyset,\mathcal{D})$.

Other data constraints, like for example $p{\in}A$ (for some set $A{\subseteq}\mathcal{D}ata$), $dc_1{\vee}dc_2$, or $dc_1{\rightarrow}dc_2$, are defined as abbreviations ("syntactic sugar") in the standard way.

The validity ("semantics") of data constraints is evaluated under a certain data assignment. Data assignments describe the data values which are pending at ports, or stored in data variables.

[2]Formally, $+$ is required to be an operation such that $(\mathcal{D}ata\backslash\bot,+)$ is an abelian group, i.e. a group which satisfies the abelian group axioms (1) closure, (2) associativity, (3) existence of identity element, (4) existence of inverse element, and (5) commutativity. As usual, the operation "$-$" (negation) is a shorthand for addition of the inverse.

Definition 2.1.8 (Data Assignment). Let $\mathcal{P}$ and $\mathcal{D}$ be as in Definition 2.1.7, and $\mathcal{D}ata$ a data domain. A *data assignment* $\delta{\in}DA(\mathcal{P}{\cup}\mathcal{D})$ *over* $\mathcal{P}$ *and* $\mathcal{D}$ is a mapping $\delta{:}(\mathcal{P}{\cup}\mathcal{D}){\to}\mathcal{D}ata$, assigning to each port $p{\in}\mathcal{P}$ the data value which is currently pending at p, and to each data variable $d{\in}\mathcal{D}$ the data value which is currently contained in d.

If $\delta(p){=}\bot$ ("no dataflow through p"), p is called *inactive*. Otherwise, p is called *active*. If $\delta(d){=}\bot$, d is called *empty*.

The *restriction* $\delta|_{\mathcal{A}}$ *of* δ *from* $(\mathcal{P}{\cup}\mathcal{D})$ *to any subset* $\mathcal{A}{\subseteq}\mathcal{P}{\cup}\mathcal{D}$ is a data assignment that agrees with δ on elements $a{\in}\mathcal{A}$, and is undefined otherwise.

We may write $DA(\mathcal{P}{,}\mathcal{D})$ instead of $DA(\mathcal{P}{\cup}\mathcal{D})$, and we use $DA(\mathcal{P})$ as a shorthand for $DA(\mathcal{P},\emptyset)$, equivalently we use $DA(\mathcal{D})$ as a shorthand for $DA(\emptyset,\mathcal{D})$.

Definition 2.1.7 allows for trivial data constraints involving only data constants (i.e., elements from $\mathcal{D}ata$), for example $\mathbb{d}_{\mathbb{1}}{=}\mathbb{d}_{\mathbb{2}}$, with $\mathbb{d}_{\mathbb{1}},\mathbb{d}_{\mathbb{2}}{\in}\mathcal{D}ata$. Since the validity of such data constraints does not depend on a specific data assignment δ, they can be evaluated statically (to either `true` or $\texttt{false}\overset{\text{def}}{=}\neg\texttt{true}$). Therefore, we assume that every data constraint involves at least one port or one data variable. We use $\models$ for the standard satisfaction relation of data assignments on data constraints. For example, $\delta{\models}(p{=}q)$ iff $\delta(p){=}\delta(q)$, and $\delta{\models}(p{\leqslant}q)$ iff $\delta(p){\leqslant}\delta(q)$.

Remark 2.1.9 (Use of $\bot$ in Data Constraints). Notice that we only allow the special value $\bot$ in simple data constraints of the form $(\mathbf{d}{=}\mathbf{d}')$. The idea is that a data constraint $(p{=}\bot)$, with $p{\in}\mathcal{P}$, represents a "check" whether port p is inactive. Equivalently, a data constraint $(d{=}\bot)$, with $d{\in}\mathcal{D}$, represents a check whether data variable d is empty.

We do not allow $\bot$ to be used in combination with $\leqslant$, since it is not clear how to define the result of such a comparison. One possible solution would be to define $\bot$ as supremum or infimum of $\mathcal{D}ata$; yet, the constraints involving $\bot$ could be simplified to `true` or `false` in this case. Apart from that, many countably infinite sets have neither a supremum nor an infimum (take for example $\mathbb{Z}$), and actually we do not consider a comparisons of the form $(\mathbb{d}{\leqslant}\bot)$, with $\mathbb{d}{\in}\mathcal{D}ata$, useful at all.

A similar argumentation holds for the use of $\bot$ in combination with $+$.

2.2 Timed Automata

In this section, we present the first and most basic system model for modelling real-time systems: Timed Automata.

Timed Automata (TA) were introduced in the seminal paper of Alur and Dill in 1994 [AD94], and have been studied intensively since. TA are finite automata, extended with real-valued clock variables, that can measure the passing of time. Their behaviour consists of a sequence of events (or actions) happening over time. Conceptually, this is represented by an infinite sequence of events, which is paired with an infinite sequence of time instants, with the intended meaning that a specific events takes place at the specific time.

To model this behaviour, TA comprise two kinds of events: visible (external) and invisible (internal) events. The former are used for synchronisation with other automata, while the latter are used for internal activities of a single automaton, independent from others.

The underlying idea is that transitions (location changes) are instantaneous, time may only elapse while the automaton remains in one of its locations. The firing of transitions, and the dwell time in locations, are restricted by constraints on the clocks—called clock guards and clock invariants, respectively—which the current clock values have to satisfy. That means, a TA is only allowed to fire a transition if the associated clock guard is satisfied, and is only allowed to stay in a location as long as the associated clock invariant is satisfied. In addition, transitions may update the values of (a subset of the) clocks to a natural number or to the value of another clock [AM04].

In the literature, many slightly different variants of TA can be found. Our definitions are essentially based on [AD94]. Please refer to Section 2.2.4 for a discussion of other variants.

2.2.1 Syntax of Timed Automata

Each transition of a TA is labelled with a distinct event, which captures "what is being performed" when the transition is fired. In this work, we distinguish between two types of events: visible (external) and invisible (internal) actions, cf. [BK08, KP07, AM04]. The former are used to synchronise with other automata when considering systems of TA, while the latter are used for internal steps of a single automaton, independent from other automata. Since internal actions can be regarded as "being of no further interest" [BK08], they are commonly denoted by the single distinguished action symbol τ. We denote the set of visible events of a TA by Σ_v, and the set of all events (visible and invisible) by Σ, i.e., $\Sigma{=}\Sigma_v\dot{\cup}\tau$. For a discussion of other possibilities to define the set of admissible events, please refer to Section 2.2.3.

Recalling Definitions 2.1.2, 2.1.4 and 2.1.5, we define the syntax of TA as follows.

Definition 2.2.1 (Timed Automaton). A *TA* is a tuple $\mathfrak{A}{=}(S, s_0, \Sigma, \mathcal{X}, I, E)$, with S a finite set of locations, $s_0{\in}S$ the initial location, Σ a finite set of visible events, $\mathcal{X}$ a finite set of real-valued clocks, $I{:}S{\to}\mathcal{CC}(\mathcal{X})$ a function assigning a clock constraint (*location invariant*) to every location, and $E{\subseteq}(S{\times}\Sigma{\times}\mathcal{CC}(\mathcal{X}){\times}\Lambda(\mathcal{X}){\times}S)$ the finite set of transitions.

The idea of transitions of TA is as follows: an element $e{=}(s, \mathfrak{a}, cc, \lambda, s'){\in}E$ describes a transition from the *source location* s to the *target location* s' on occurrence ("execution") of *action* $\mathfrak{a}$. The firing of the transition is restricted by the *(clock) guard* cc, and updates clocks according to the *update map* λ. For every such transition, we require cc to be satisfiable. If $\mathfrak{a}{\in}\Sigma_v$, we call e an *external (or visible)* transition, otherwise (i.e., if $\mathfrak{a}{=}\tau$), e is called *internal (or invisible)*.

Example 2.2.2 (Timed Automaton). Two examples for TA are given in Figures 2.1 and 2.2. The TA in Figure 2.1 models an "intelligent light switch": it consists of three locations *off*, *light* and *bright*, representing the corresponding states of the light, and a clock x. In the initial location (marked by the incoming arrow), the light is off. If the switch is pressed (modelled by action *press*), the light turns on (location *light*), and the clock x is reset to measure the temporal difference to the next *press* action. If the switch is pressed again before the value of x reaches 3 (modelled by the guard $x{\leq}3$), the light becomes bright, otherwise (i.e., if the value of x is greater than 3, modelled by the guard $x{>}3$), it switches off again.

The automaton in Figure 2.2 shows a "user" of the light switch. The automaton consists of a single location and a clock y. The location has an invariant $y{\leq}4$, which forces the automaton to leave the location after having delayed there for at most 4 time units. Together with the guard $y{\geq}2$ and the update $y{:=}0$ on the transition, this models a user which executes the *press* action every 2 to 4 time units.

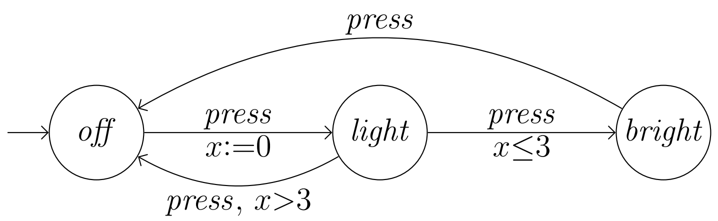


Figure 2.1: Intelligent Light Controller

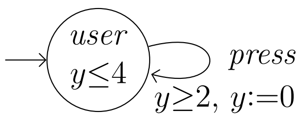


Figure 2.2: User pressing the Light Switch

In the graphical representation of TA, we use assignment rather than functional notation for the updates, and we omit guards and invariants equal to `true` as well as identity updates of the form $\lambda(x){=}x$.

Notation 2.2.3 (TA). If not stated otherwise, we shall assume the constituents of a TA $\mathfrak{A}$ to be denoted as $\mathfrak{A}{=}(S, s_0, \Sigma, \mathcal{X}, I, E)$, and of a TA $\mathfrak{A}_i$ to be denoted as $\mathfrak{A}_i{=}(S_i, s_{0,i}, \Sigma_i, \mathcal{X}_i, I_i, E_i)$, for $i{\in}\mathbb{N}$.

By $CC(\mathcal{X})|_{\mathfrak{A}}$, we denote the set of clock constraints (over clock set $\mathcal{X}$) that occur in a TA $\mathfrak{A}$ (invariants or guards).

2.2.2 Semantics of Timed Automata

As mentioned above, the idea of TA is that transitions are instantaneous, time only elapses while the automaton remains in one of its locations. The semantics of a TA $\mathfrak{A}$ is defined as the set of runs of the associated labelled transition system (LTS) $\mathfrak{S}_{\mathfrak{A}}$, cf. for example [BK08].

Definition 2.2.4 (Associated Labelled Transition System). Let $\mathfrak{A}$ be a TA. The *associated LTS* $\mathfrak{S}_{\mathfrak{A}}$ is a tuple $\mathfrak{S}_{\mathfrak{A}}{=}(\mathcal{Q}, q_0, \rightarrow)$, with $\mathcal{Q}{\subseteq}(S{\times}\mathcal{V}(\mathcal{X}))$ the set of *configurations*, such that $\nu{\models}I(s)$ for every $(s, \nu){\in}\mathcal{Q}$, $q_0{=}\langle s_0, \mathbf{0}\rangle$ the *initial configuration*, with $\mathbf{0}(x){=}0$ for all $x{\in}\mathcal{X}$, and the transition relation $\rightarrow\ \subseteq(\mathcal{Q}{\times}(\mathtt{Time}{\cup}\Sigma){\times}\mathcal{Q})$ is given in (2.1) and (2.2).

$$\frac{(s, \mathfrak{a}, cc, \lambda, s')\in E,\quad \nu\models cc, \nu[\lambda]\models I(s')}{\langle s,\nu\rangle \xrightarrow{\mathfrak{a}} \langle s', \nu[\lambda]\rangle} \qquad (2.1) \qquad\qquad \frac{s\in S, t\in\mathtt{Time}, t>0,\quad \forall t', t\geq t'\geq 0 : \nu+t'\models I(s)}{\langle s,\nu\rangle \xrightarrow{t} \langle s, \nu+t\rangle} \qquad (2.2)$$

A *run of* $\mathfrak{S}_{\mathfrak{A}}$ *(starting in configuration q)* is an infinite sequence of transitions $q_0 \xrightarrow{\mathfrak{a}_0} q_1 \xrightarrow{\mathfrak{a}_1} \ldots$, with $\mathfrak{a}_i\in(\Sigma\cup\mathtt{Time})$ for all $i>0$. A run is called *initial* if it starts in the initial configuration q_0, it is called *loop-free* if all configurations are different.

Rule (2.1) describes an *action transition* (with action $\mathfrak{a}$) of $\mathfrak{S}_{\mathfrak{A}}$, based on a transition $e\in E$ of $\mathfrak{A}$. The valuation ν of the source configuration $\langle s, \nu\rangle$ needs to satisfy ("enable") the clock guard cc, and the updated valuation $\nu[\lambda]$ after the execution of the transition needs to satisfy the invariant of location s' (otherwise, the automaton could not enter location s'). On execution of the transition, the values of the clocks of $\mathfrak{A}$ are updated according to the update map λ. Rule (2.2) describes a *delay transition* (in location s) of $\mathfrak{S}_{\mathfrak{A}}$: the invariant $I(s)$ of the location needs to be satisfied at all times (for all t'), and the clock values of all clocks increase by the same amount of time t.

Definition 2.2.5 (Semantics of Timed Automata). Let $\mathfrak{A}$ be a TA, $\mathfrak{S}_{\mathfrak{A}}$ the associated LTS as defined in Definition 2.2.4. The *trace semantics of* $\mathfrak{A}$ is given by the set $Run_{\mathfrak{A}}$ of initial runs of $\mathfrak{S}_{\mathfrak{A}}$. With $Run_{\mathfrak{A},k}$, we denote the set of finite prefixes of elements of $Run_{\mathfrak{A}}$ of (at most) length k.

Note that we do not require clock guards of outgoing transitions of a location to be complete, i.e., cover every possible valuation. This may lead to so-called *timelocks* [Tri99]: consider a location with invariant $x\leq 3$ and a single outgoing transition with clock guard $x\leq 2$. The location cannot be left once $\nu(x)>2$, and as soon as $\nu(x)=3$, from a theoretical point of view, time is not allowed to progress anymore, since the automaton is neither allowed stay in the location nor allowed to leave it. However, using the above definitions, such behaviour is excluded from the semantics.

Example 2.2.6 (Run of a Timed Automaton). In (2.3), we show a run of the intelligent light switch from Figure 2.1 of length 10.

$$\begin{aligned}&\langle \mathit{off}, x{=}0\rangle \xrightarrow{2} \langle \mathit{off}, x{=}2\rangle \xrightarrow{\mathit{press}} \langle \mathit{light}, x{=}0\rangle \xrightarrow{2.5} \langle \mathit{light}, x{=}2.5\rangle \xrightarrow{\mathit{press}}\\ &\langle \mathit{bright}, x{=}2.5\rangle \xrightarrow{1} \langle \mathit{bright}, x{=}3.5\rangle \xrightarrow{\mathit{press}} \langle \mathit{off}, x{=}3.5\rangle \xrightarrow{\mathit{press}}\\ &\langle \mathit{light}, x{=}0\rangle \xrightarrow{6} \langle \mathit{light}, x{=}6\rangle \xrightarrow{3} \langle \mathit{light}, x{=}9\rangle \xrightarrow{\mathit{press}} \langle \mathit{off}, x{=}9\rangle \qquad (2.3)\end{aligned}$$

Convexity of clock constraints (cf. Remark 2.1.6) gives rise to the following property for sequences of delay transitions, cf. [Alu99].

Remark 2.2.7 (Time-Additivity). For two consecutive delay transitions in a run, *time-additivity* holds. That means, for a TA $\mathfrak{A}$ and associated LTS $\mathfrak{S}_{\mathfrak{A}}=(\mathcal{Q}, q_0, \rightarrow)$, with configurations $q_1, q_2, q_3\in\mathcal{Q}$, and $t_1, t_2\in\mathtt{Time}$, if $q_1 \xrightarrow{t_1} q_2 \in \rightarrow$ and $q_2 \xrightarrow{t_2} q_3 \in \rightarrow$, then also $q_1 \xrightarrow{t_1+t_2} q_3 \in \rightarrow$.

2.2.3 Systems of Timed Automata

In this section, we present a product construction for TA which is *compositional*, and therefore allows to build complex and/or distributed systems by first designing the individual components separately, and then combining them with the product operation.

The intended idea of a system of TA is that the automata work in parallel, while synchronising via transitions labelled with the same event. In this work, we assume TA to perform *joint broadcast synchronisation* on visible events (cf. [Alu99]). That means, if some visible event $\mathfrak{a}$ occurs, every automaton $\mathfrak{A}_i$ in the system, with $\mathfrak{a}\in\Sigma_i$ ("knowing about $\mathfrak{a}$") must execute a transition labelled with $\mathfrak{a}$, while an automaton $\mathfrak{A}_i$ with $\mathfrak{a}\notin\Sigma_i$ performs a zero-delay step ("nothing"). If, instead, event τ occurs, automata may decide to either execute a transition labelled with τ or do a zero-delay step. Delay steps with delay $t>0$ have to be executed synchronously by all automata. The product automaton for two TA $\mathfrak{A}_1$ and $\mathfrak{A}_2$ is defined as follows.

Definition 2.2.8 (Product of TA). Let $\mathfrak{A}_1$, $\mathfrak{A}_2$ be TA, with $\mathcal{X}_1\cap\mathcal{X}_2=S_1\cap S_2=\emptyset$ (can be achieved by renaming the constituents in one of the TA). The *product of* $\mathfrak{A}_1$ *and* $\mathfrak{A}_2$ is a new TA $\mathfrak{A}_1\bowtie\mathfrak{A}_2=(S, s_0, \Sigma, \mathcal{X}, I, E)$, with $S=S_1\times S_2$, $s_0=(s_{0,1}, s_{0,2})$, $\Sigma=\Sigma_1\cup\Sigma_2$, $\mathcal{X}=\mathcal{X}_1\cup\mathcal{X}_2$, $I{:}S_1\times S_2\to CC(\mathcal{X})$ such that for $s=(s_1, s_2)\in S$, $I(s)=I(s_1)\wedge I(s_2)$, and E is defined in (2.4) and (2.5), and the symmetric rule of the latter.

$$\frac{(s_1, \mathfrak{a}, cc_1, \lambda_1, s_1')\in E_1,\quad (s_2, \mathfrak{a}, cc_2, \lambda_2, s_2')\in E_2,}{((s_1, s_2), \mathfrak{a}, cc_1\wedge cc_2, \lambda_1\circ\lambda_2, (s_1', s_2'))} \quad (2.4) \qquad \frac{(s_1, \mathfrak{a}, cc_1, \lambda_1, s_1')\in E_1,\quad (\mathfrak{a}\notin\Sigma_2) \text{ or } (\mathfrak{a}=\tau), s_2\in S_2}{((s_1, s_2), \mathfrak{a}, cc_1, \lambda_1, (s_1', s_2))} \quad (2.5)$$

Rule (2.4) describes synchronisation, that means the automata execute a transition labelled with the same event (note that this can be a visible event as well as the internal action τ) in parallel. The resulting transition in the product involves a location change in both underlying TA, the transition is guarded by the combined guard $cc_1\wedge cc_2$, and clocks are updated according to the combined update maps $\lambda_1\circ\lambda_2$. Note that since $\mathcal{X}_1\cap\mathcal{X}_2=\emptyset$, we have $\lambda_1\circ\lambda_2=\lambda_2\circ\lambda_1$, see also Proposition 2.2.9. Rule (2.5) describes the execution of a local transition (visible or invisible) in one of the TA, while the other automaton remains in its current location.

Proposition 2.2.9 (Product of TA). The product of TA is commutative and associative, up to isomorphy of location names.

Proof.

1. Commutativity follows from the commutativity of $\wedge$ on clock constraints, and the commutativity of $\circ$ (function composition) on update maps over disjoint clock sets.

2. Associativity follows from the associativity of $\wedge$ on clock constraints, and the associativity of $\circ$ on update maps over disjoint clock sets. □

Example 2.2.10 (Product of TA). An example for the product construction can be found in Figure 2.3. The automaton shows the product TA for the intelligent light switch and user, as presented in Example 2.2.2.

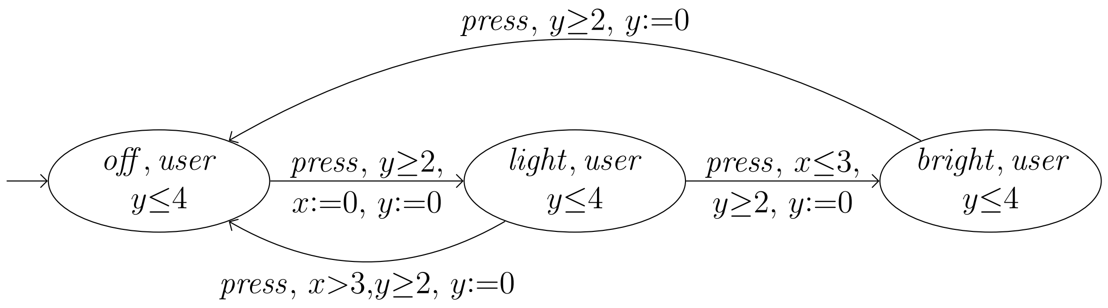


Figure 2.3: Product of Light Switch and User

The semantics of a system of two TA $\mathfrak{A}_1$ and $\mathfrak{A}_2$ (with $\mathcal{X}_1{\cap}\mathcal{X}_2{=}\emptyset$ and $S_1{\cap}S_2{=}\emptyset$, as required in Definition 2.2.8) is defined as the semantics (Definition 2.2.5) of the corresponding product automaton $\mathfrak{A}_1{\bowtie}\mathfrak{A}_2$ (Definition 2.2.8), i.e., the set of runs of the associated LTS $\mathfrak{S}_{\mathfrak{A}_1\bowtie\mathfrak{A}_2}$ (Definition 2.2.4). In (2.6), we show a run of the product automaton from Figure 2.3 of length 9.

$$\begin{aligned}
&\langle(\mathit{off},\mathit{user}),\ {\substack{x=0\\y=0}}\rangle \xrightarrow{2} \langle(\mathit{off},\mathit{user}),\ {\substack{x=2\\y=2}}\rangle \xrightarrow{\mathit{press}} \langle(\mathit{light},\mathit{user}),\ {\substack{x=0\\y=0}}\rangle \xrightarrow{2.5}\\
&\quad\langle(\mathit{light},\mathit{user}),\ {\substack{x=2.5\\y=2.5}}\rangle \xrightarrow{\mathit{press}} \langle(\mathit{bright},\mathit{user}),\ {\substack{x=2.5\\y=0}}\rangle \xrightarrow{1}\\
&\quad\langle(\mathit{bright},\mathit{user}),\ {\substack{x=3.5\\y=1}}\rangle \xrightarrow{\mathit{press}} \langle(\mathit{off},\mathit{user}),\ {\substack{x=3.5\\y=0}}\rangle \xrightarrow{\mathit{press}} \langle(\mathit{light},\mathit{user}),\ {\substack{x=0\\y=0}}\rangle \xrightarrow{4}\\
&\quad\langle(\mathit{light},\mathit{user}),\ {\substack{x=4\\y=4}}\rangle \xrightarrow{\mathit{press}} \langle(\mathit{off},\mathit{user}),\ {\substack{x=4\\y=0}}\rangle
\end{aligned}\tag{2.6}$$

Remark 2.2.11 (Size of the Product). The product automaton is exponential in the size of the underlying TA in the worst case. Therefore, for model checking/verification, we provide a technique to avoid the explicit construction of the exponential cross product in Section 3.1.2.

2.2.4 Discussion

TA were introduced a long time ago, and are by now well-studied. As a result, TA have been extended and adapted in many ways and for many purposes, like for example Timed Automata with Deadlines [BS00, GS05], Task Automata [FKPY07], or Probabilistic Timed Automata [Bea03, KNSS02]. Yet, even for the "basic" version of TA, slightly different variants of how to define them can be found in the literature, each of which has advantages and disadvantages. We now discuss the major distinctions, and motivate our design decisions.

2.2.4.1 Internal Actions

The original work [AD94, Alu99] did not consider internal actions, but only considered visible actions in the set of admissible events. As a consequence, every transition of a TA could synchronise with a transition in another TA, if a transition labelled with the same event was enabled. In this work, in addition to visible events, we also allow for invisible internal actions. We consider this more realistic: typically, the behaviour of a real-time system not only depends on the environment (modelled by synchronisation via visible actions), but also on internal state changes, which cannot be influenced by other TA in any way. Consider for example a deadline expiration, after which the behaviour of the system changes.

It has been shown in [AM04] that internal actions add to the expressive power of TA, i.e., a TA with internal actions is more expressive than a TA without internal actions. As an example, consider the TA in Figure 2.4. The automaton consists of a single location s, a clock x, and can synchronise with other automata via action $\mathfrak{a}$. The automaton requires all actions to occur at integer times, and no two actions occur at the same time: both transitions become enabled exactly one time unit after the system started, modelled by the guard $x{=}1$. At this time, the invariant $x{\le}1$ forces the automaton to leave location s, by performing either of the transitions. If no synchronisation via action $\mathfrak{a}$ (left transition) is possible, the automaton executes the internal action τ (right transition). Both transitions reset clock x to zero, such that the automaton can reenter location s after the execution of the transition.

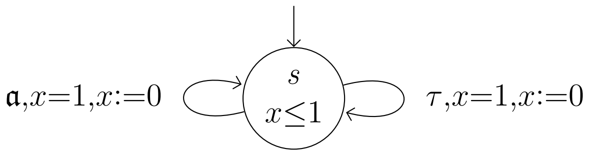


Figure 2.4: Use of Invisible Transitions

This behaviour cannot be modelled with a TA which does not allow for τ-transitions (cf. [AM04]): if the largest constant occurring in guards and invariants was c,[3] the TA could not distinguish between occurrences of action $\mathfrak{a}$ which happen at times $c{+}1$ and $c{+}1.1$ (after the start of the computation).

2.2.4.2 Synchronisation

Within systems of TA, we assume *joint broadcast synchronisation* (cf. Section 2.2.3), that means, on occurrence of a visible event $\mathfrak{a}$, all TA "knowing about" $\mathfrak{a}$ have to synchronise on this event. In particular, there is no restriction on how many automata can participate in a single synchronisation, the number is determined only by the fact whether $\mathfrak{a}$ is contained in the event set of an automaton. Another possibility of defining synchronisation (which is closer to the approach taken in process algebras) is to assume the synchronisation to be *binary*: under this assumption, *exactly two* automata synchronise, i.e., execute a transition labelled with some event $\mathfrak{a}$ in parallel. If more than two automata are ready to synchronise on $\mathfrak{a}$, the choice which pair—i.e., which two automata—executes the synchronisation transitions is made nondeterministically.

The expressiveness of these two ways of modelling synchronisation is the same. We have chosen for joint broadcast synchronisation, since we consider this closer

[3] Such largest constant always exists, since both the set of locations as well as the set of transitions of TA are finite, cf. Definition 2.2.1

to the nature of component-based systems. The ideas of modelling each approach with the respective other can roughly be sketched as follows: using TA with joint broadcast synchronisation to model binary synchronisation is straightforward, by using each action in exactly two automata only. Using TA with binary synchronisation to model joint broadcast synchronisation uses the fact that transitions are instantaneous, therefore, multiple transitions (all labelled with the same action) can take place at the same time instant.

2.2.4.3 Input- and Output Actions

A special case of binary synchronisation is the distinction between input and output actions. While we consider two types of actions (visible/external and invisible/internal), some authors prefer to further divide the set of visible actions into input and output actions, cf. for example [BDL04]. Usually, input and output actions are used to distinguish between actions that are controlled by the TA, and actions that are controlled by the environment. Since we do not make this distinction here, we regard two types of actions (external and internal) to be sufficient.

2.2.4.4 Clock Constraints and Updates

In the original work presented in [AD94, Alu99], so-called *diagonal* clock constraints of the form $x - y \sim c$ (i.e., involving clock differences, cf. Definition 2.1.2) were not allowed. However, it has been shown by Alur and Madhusudan [AM04] that adding these constraints does not add to the expressive power of TA. Therefore, we allow diagonal clock constraints, since they may be used for more concise modelling.

Moreover, [AD94, Alu99] only allowed transitions to reset a (sub)set of clocks to zero. It has been shown in [AM04] that this restriction can be relaxed in favour of more general *update maps*, without adding to the expressiveness. We follow their approach and use the update maps presented in [AM04], cf. Definition 2.1.5.

In contrast to [AD94], but following [Alu99], we do not allow negation in clock constraints, cf. Definition 2.1.2. This is done to obtain convex clock constraints (cf. Remark 2.1.6), which are used for efficient representation in Chapter 3. Yet, non-convex clock constraints can be simulated by splitting locations (for non-convex invariants) and transitions (for non-convex guards).

These considerations on clock constraints and update maps directly carry over to TCA (Section 2.3) and TNA (Section 2.4).

2.3 Timed Constraint Automata

In this section, we define the second system model for modelling real-time systems: Timed Constraint Automata (TCA). TCA [ABdBR07, Kem11] arise from combining the concepts of constraint automata [ABRS04] (CA) and timed automata (TA, Section 2.2). CA were originally defined as a semantical model for the channel-based coordination language $\mathcal{R}$eo [Arb04], and consequently, TCA were intended to serve as a semantical model of a timed variant of $\mathcal{R}$eo. Yet, TCA offer a powerful coordination mechanism for channel-based coordination languages in general. They are specially tailored for implementing coordinating connectors in networks where timed components communicate by exchanging data through multiple channels. The behaviour of the network is given by synchronisation between channel ends (ports).

For this, TCA allow for two types of transitions: internal (invisible) location changes caused by some timing constraints, and external (visible) transitions representing data flow at some of the ports. Transitions in TCA are labelled with sets of actions (ports). The underlying general idea is that all actions which happen at the same time (i.e., atomically) collapse into a single transition. As a consequence, a positive amount of time elapses before every visible transitions.

The major conceptual difference to TA and other action-based coordination models, like e.g. finite state machines, or timed I/O automata [KLSV03a, KLSV03b], is thus the "non-instantaneous" handling of transitions. While in TA, every transition can be fired immediately (provided that the guard is satisfied), TCA require a positive delay before every visible transition. Moreover, TA permit only a single action per transition, such that synchrony—and concurrent execution in the parallel composition—of different actions is reduced to arbitrary interleavings plus nondeterminism. This is unintuitive, since it imposes a sequential order on actions which conceptually happen at the same time. Moreover, from a technical point of view, the presence of all possible interleavings amplifies the state explosion problem. TCA permit *true concurrency*, as they directly model truly atomic synchronous communication through different ports.

2.3.1 Syntax of Timed Constraint Automata

The syntactic concepts for handling real-time (i.e., clocks, guards, invariants), as defined in the previous section, directly carry over from TA to TCA. For handling data values, each TCA is equipped with a finite set of ports, through which it exchanges data values with other TCA. Transition of TCA are labelled with a subset of these ports, with the intended meaning that data flows through these ports (the ports are active) when the transition is fired. Transitions can be labelled with a data constraint (cf. Definition 2.1.7), restricting the admissible data values flowing through the active ports. In addition, the coordination pattern implemented by a TCA may depend on data values which have been exchanged *before* (i.e., through ports that were active in previous steps). To keep track of this, we extend the basic definition of TCA, presented in [Kem11], with *location memory*, cf. also [PSHA09]: each TCA is equipped with a finite set of data variables, which can be used by locations to store data values, cf. Section 2.1.2. We call these data variables associated to locations *memory cells*.[4]

The underlying idea is that memory cells enable the *current* location to store data values. The same memory cell can be used by both source and target location of the same transition, in this case, its contents carry over unchanged, unless they are updated by the transition. The contents of memory cells which are used by the source locations but not by the target location are discarded, i.e., unused memory

[4]In the remainder of this Section, we may use the terms *memory cell* and *data variable* interchangeably.

cells are empty by definition. Consequently, data constraints on transitions may only refer to memory cells used by the source or target location.

Recalling Definitions 2.1.2, 2.1.4, 2.1.5, 2.1.7 and 2.1.8, TCA are defined as follows.

Definition 2.3.1 (Timed Constraint Automaton). A *TCA over data domain $\mathcal{D}ata$* is a tuple $\mathfrak{T}=(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, with S a finite set of locations, $s_0 \in S$ the initial location, $\mathcal{P}$ a finite set of ports, $\mathcal{X}$ a finite set of clocks, $I{:}S \to \mathcal{CC}(\mathcal{X})$ a function assigning a clock constraint (*location invariant*) to every location, $\mathcal{D}$ a finite set of data variables, called *memory cells*, $\#{:}S \to 2^{\mathcal{D}}$ a function assigning to each location the set of memory cells it may use, and $E \subseteq (S \times 2^{\mathcal{P}} \times \mathcal{DC}(\mathcal{P},\mathcal{D}) \times \mathcal{CC}(\mathcal{X}) \times \Lambda(\mathcal{X}) \times S)$ the finite set of transitions.

For every element $e=(s, P, dc, cc, \lambda, s') \in E$, we require that the *data guard dc* only reasons about active ports, and only about memory cells of the *source location s* and the *target location s'*, i.e., $dc \in \mathcal{DC}(P, \#(s) \cup \#(s'))$. We require both dc and cc (*clock guard*) to be satisfiable. If $P=\emptyset$, transition e is called *invisible*, otherwise, it is called *visible*.

The idea of invisible transitions is that they do not represent observable data flow: the port set P is empty, so no ports are active. The data constraint dc of an invisible transition may thus only reason about memory cells, i.e., $dc \in \mathcal{DC}(\emptyset, \#(s) \cup \#(s'))$. In this way, invisible transitions only serve for internal synchronisation purposes, for example by changing to another location, or updating clocks. Visible transitions, on the other hand, correspond to observable behaviour: on location change from s to s', data flows through all ports in the *port set P*. After the TCA has delayed in location s for a positive amount of time,[5] during which the invariant $I(s)$ of s needs to be satisfied, it executes the transition and moves to location s', provided that the data values pending at the active ports (in P) and contained in memory cells of the source location s satisfy the data guard dc, and the clock values satisfy the *clock guard cc* and the invariant $I(s')$ of the target location s'. The firing of the transition, i.e., the location change from s to s', is considered to be instantaneous. On execution of the transition, all clocks are updated according to the update map λ. These informally described timing constraints will be made explicit in the definition of semantics (Definition 2.3.6).

Remark 2.3.2 (Use of Memory Cells). We do not impose any restrictions on the set of memory cells used by locations. In particular, the same memory cell m may be used by both source and target location of a transition. This may lead to an ambiguous data constraint dc in case dc reasons about m. Yet, such behaviour is necessary for example when it comes to product definition (cf. Definition 2.3.9). To avoid ambiguities, we indicate in the data constraint to which location a memory cell m belongs, i.e., whether the occurrence of m refers to its value before or after the execution of the transition. We add to m the prefix "$s.$" if it refers to the value used

[5]As explained in the beginning of the section, the general idea of TCA is that all data flow actions which happen at the same time atomically collapse to a single transition, therefor, every such transition is preceded by a positive time delay.

by the source location, and the prefix "*t.*" if it refers to the value used by the target location. For example, instead of $(m{=}\mathbb{d}_{\mathbb{1}})\wedge(m{=}\mathbb{d}_{\mathbb{2}})$ (which would obviously evaluate to `false`), we write $(s.m{=}\mathbb{d}_{\mathbb{1}})\wedge(t.m{=}\mathbb{d}_{\mathbb{2}})$. Any data assignment δ will consider $s.m$ and $t.m$ to be different elements.

Our definition allows to leave the value of a memory cell m used by the target location unspecified, i.e., neither does the source location use m, nor does m occur in the data constraint. In this case, we assume m to take a random value.

While unused memory cells are empty be definition (cf. the intuition at the beginning of this section, and also Definition 2.3.5), we may explicitly require memory cells used by the current location to be empty as well, using data constraints of the form $(m{=}\bot)$ (cf. Definition 2.1.7).

Example 2.3.3 (Timed Constraint Automaton). An example for a TCA is given in Figure 2.5. It models a FIFO buffer with capacity 1 and expiration: the buffer consists of two locations *empty* and *full*, uses clock x to measure the time until the data expires, and two ports p and q for exchanging data values. The *full* location has an associated memory cell m, which is denoted $[m]$ in the graphical representation. In the initial location, the buffer is empty. On receiving a data value through port p, the TCA moves to location *full*, updates (resets) clock x to 0, and stores the data value in the location memory of *full*, indicated by the data guard $p{=}t.m$. Since we do not impose any further data constraints, this means that any value can be received through p. If data flow through port q is possible before 3 time units have elapsed (clock guard $x{<}3$), the TCA sends the data item from the memory cell through port q, and moves to back to the initial location. If data flow through q is not possible, at time 3, the invariant of location *full* forces the TCA to leave that location, and execute the invisible transition back to the initial location. No data is transmitted, which models "loosing" the stored data item.

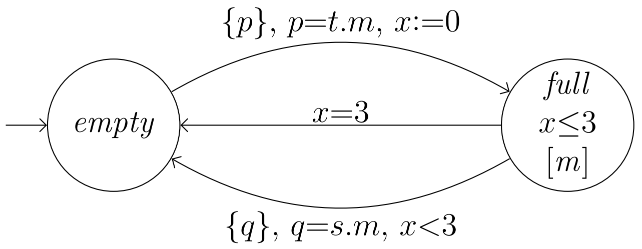


Figure 2.5: 1-bounded FIFO Buffer with Expiration

Notation 2.3.4 (TCA). If not stated otherwise, we shall assume the constituents of a TCA $\mathfrak{T}$ to be denoted as $\mathfrak{T}{=}(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, and of a TCA $\mathfrak{T}_i$ to be denoted as $\mathfrak{T}_i{=}(S_i, s_{0,i}, \mathcal{P}_i, \mathcal{X}_i, I_i, \mathcal{D}_i, \#_i, E_i)$, for $i{\in}\mathbb{N}$. We lift $\#$ to reason about sets of locations, and we may omit curly braces: $\#(s, s') \overset{\text{def}}{=} \#(s){\cup}\#(s')$.

By $\mathcal{CC}(\mathcal{X})|_{\mathfrak{T}}$, we denote the set of clock constraints (over clock set $\mathcal{X}$) that occur in a TCA $\mathfrak{T}$ (invariants or guards). Equivalently, by $\mathcal{DC}(\mathcal{P}{,}\mathcal{D})|_{\mathfrak{T}}$, we denote the set of data constraints over port set $\mathcal{P}$ and memory cells $\mathcal{D}$ that occur in a TCA $\mathfrak{T}$.

As for TA, in the graphical notation of TCA, we use assignment rather than functional notation for updates, and we omit guards and invariants equal to `true`, as well as identity updates. We denote a set of memory cells $\{m_1, \dots, m_n\}$ associated to a location as a bracketed list $[m_1, \dots, m_n]$.

2.3.2 Semantics of Timed Constraint Automata

TCA model true concurrency, by allowing sets of ports on each transition. As a consequence, a positive amount of time has to elapse before every visible transition (while invisible transitions may be instantaneous). The underlying idea is that all actions which happen at the same time are truly atomic and thus collapse to a single transition. The semantics of a TCA $\mathfrak{T}$ is defined as the set of runs of the associated labelled transition system (LTS) $\mathfrak{S}_{\mathfrak{T}}$.

Definition 2.3.5 (Associated LTS). Let $\mathfrak{T}$ be a TCA. The *associated LTS* $\mathfrak{S}_{\mathfrak{T}}$ is a tuple $\mathfrak{S}_{\mathfrak{T}}=(\mathcal{Q}, q_0, \rightarrow)$, with $\mathcal{Q}\subseteq(S\times DA(\mathcal{D})\times\mathcal{V}(\mathcal{X}))$ the set of configurations, where for every $(s,\delta,\nu)\in\mathcal{Q}$, $\delta\in DA(\mathcal{D})$, with $\delta(m)=\bot$ if $m\notin\#(s)$, and $\nu\models I(s)$. The initial configuration is $q_0=\langle s_0, \mathbf{0}, \mathbf{0}\rangle$, with $\mathbf{0}(x)=0$ for all $x\in\mathcal{X}$ and $\mathbf{0}(m)=\bot$ for all $m\in\#(s_0)$. The transition relation $\rightarrow\ \subseteq(\mathcal{Q}\times 2^{\mathcal{P}}\times DA(\mathcal{P},\mathcal{D})\times\texttt{Time}\times\mathcal{Q})$ is given in (2.7) and (2.8).

$$\frac{\begin{array}{l}(s,P,dc,cc,\lambda,s')\in E\\ t>0, t\geq t'\geq 0 : \nu+t'\models I(s), \nu+t\models cc, (\nu+t)[\lambda]\models I(s')\\ \bar{\delta}\in DA(\mathcal{P},\mathcal{D}) : \bar{\delta}\models dc\\ \bar{\delta}(s.m)=\begin{cases}\delta(m), & \text{if } m\in\#(s)\\ \bot, & \text{otherwise}\end{cases}\\ \bar{\delta}(t.m)=\delta(m) \text{ if } m\in\#(s), m\in\#(s'), t.m\notin\mathcal{D}|_{dc}\\ \bar{\delta}(m)=\bot \text{ if } m\in\mathcal{D}\backslash\#(s,s'),\\ \bar{\delta}(p)=\bot \text{ iff } p\in\mathcal{P}\backslash P,\\ \delta'(m)=\bar{\delta}(t.m) \text{ for all } m\in\#(s')\end{array}}{\langle s,\delta,\nu\rangle\xrightarrow{P,\bar{\delta},t}\langle s',\delta',\nu+t[\lambda]\rangle} \tag{2.7}$$

$$\frac{\begin{array}{l}(s,\emptyset,dc,cc,\lambda,s')\in E\\ \nu\models cc, \nu[\lambda]\models I(s')\\ \bar{\delta}\in DA(\mathcal{P},\mathcal{D}) : \bar{\delta}\models dc\\ \bar{\delta}(s.m)=\begin{cases}\delta(m), & \text{if } m\in\#(s)\\ \bot, & \text{otherwise}\end{cases}\\ \bar{\delta}(t.m)=\delta(m) \text{ if } m\in\#(s), m\in\#(s'), t.m\notin\mathcal{D}|_{dc}\\ \bar{\delta}(m)=\bot \text{ if } m\in\mathcal{D}\backslash\#(s,s'),\\ \bar{\delta}(p)=\bot \text{ iff } p\in\mathcal{P},\\ \delta'(m)=\bar{\delta}(t.m) \text{ for all } m\in\#(s')\end{array}}{\langle s,\delta,\nu\rangle\xrightarrow{\emptyset,\bar{\delta},0}\langle s',\delta',\nu[\lambda]\rangle} \tag{2.8}$$

A *run of* $\mathfrak{S}_{\mathfrak{T}}$ *(starting in configuration q)* is a sequence of transitions $q \xrightarrow{P,\delta,t} q_1 \xrightarrow{P',\delta',t'} \ldots$ which is either time divergent (i.e. infinite, and $t{+}t'{+}\ldots{=}\infty$) or finite and ends in a *terminal configuration* $\langle s, \delta, \nu\rangle$ (i.e. without outgoing transitions, allowing for infinite passage of time: $\forall t{>}0{:}\nu{+}t{\models}I(s)$). A run is called *initial* if it starts in the initial configuration q_0, it is called *loop-free* if all configurations are different.

Rule (2.7) captures the constraints described after Definition 2.3.1, for both visible and invisible transitions: before the transition can be fired, a positive amount of time ($t{>}0$) has to elapse, during which the invariant $I(s)$ needs to be satisfied at all times.[6] At time t, the transition is fired, and updates all clocks according to the update map λ, provided that the clock guard cc is satisfied *before* the resetting of clocks, and the invariant of the target location is satisfied (after the time delay and) *after* the resetting of clocks (all second row). The data values have to satisfy the data guard (third row), the values of memory cells before the execution of the transition remain unchanged if used by the source location s, otherwise, they are empty (fourth row). The values of memory cells which are used in both source and target location carry over to s' if they are not modified in the data constraint (fifth row). Note that if a memory cell is not used in the target location, its value $t.m$ after the execution of the transition is unspecified, cf. also Remark 2.3.2. All other memory cells are empty (sixth row), data is only pending at active ports (seventh row), and the data assignment on the transition determines the data values of the memory cells of the target location (eighth row).

Rule (2.8) captures the fact that invisible transitions may be instantaneous; it can be seen as a simplification of (2.7) for $P{=}\emptyset$ and $t{=}0$. Note that data may not be pending at any port (seventh row), yet data constraints of invisible transitions may still reason about memory cells of the involved locations, for example to copy values from one memory cell to another, or to model data loss.

As mentioned in Remark 2.3.2, memory cells which are used by the target location, but are neither used by the source location nor occur in the data constraint, can take a random value. As can be seen from the fifth rows in (2.7) and (2.8), we do not impose any constraints on the values ($t.m$) of these memory cells, which indeed allows $\bar{\delta}$ to assign a random (up to the fact that $\bar{\delta}{\models}dc$ has to hold) value to them.

Definition 2.3.6 (Semantics of Timed Constraint Automata). Let $\mathfrak{T}$ be a TCA, $\mathfrak{S}_{\mathfrak{T}}$ the associated LTS as defined in Definition 2.3.5. The *trace semantics of* $\mathfrak{T}$ is given by the set $Run_{\mathfrak{T}}$ of initial runs of $\mathfrak{S}_{\mathfrak{T}}$. With $Run_{\mathfrak{T}k}$, we denote the set of finite prefixes of elements of $Run_{\mathfrak{T}}$ of (at most) length k.

Note that as for TA, we do not require clock guards to be complete, nor do we require this for data guards, cf. Page 14.

Example 2.3.7 (Run of a Timed Constraint Automaton). In (2.9), we show a run of the 1-bounded FIFO buffer from Figure 2.5 of length 4.

[6] Due to convexity, these constraints can be relaxed in the representation, since it is enough to check the invariant at the beginning and at the end of the time delay.

$$\begin{aligned}\langle \mathit{empty}, \bot, x{=}0\rangle &\xrightarrow{\{p\},\,\substack{p=2\\ t.m=2},4} \langle \mathit{full}, m{=}2, x{=}0\rangle \xrightarrow{\{q\},\,\substack{q=2\\ s.m=2},2.5} \\ &\langle \mathit{empty}, \bot, x{=}2.5\rangle \xrightarrow{\{p\},\,\substack{p=5\\ t.m=5},10} \langle \mathit{full}, m{=}5, x{=}0\rangle \xrightarrow{\emptyset,\bot,3} \\ &\langle \mathit{empty}, \bot, x{=}3\rangle \end{aligned} \tag{2.9}$$

Here, we omit data assignments on transitions which evaluate to $\bot$, and we abuse the single symbol $\bot$ to denote the data assignment assigning $\bot$ to all elements.

Remark 2.3.8 (Maximal Progress). The semantics of TCA, as defined above, requires *maximal progress with respect to active ports*: on execution of a (visible) transition, data is pending on all active ports, and on none of the non-active ports, ensured by the constraints $\bar{\delta}(p){=}\bot$ iff $p{\in}\mathcal{P}\backslash P$ in (2.7) and $\bar{\delta}(p){=}\bot$ iff $p{\in}\mathcal{P}$ in (2.8).

For example, if data is pending at ports p and q, and two transitions with port sets $\{p\}$ and $\{p, q\}$, respectively, are ready to fire, then only the latter transition is enabled.

2.3.3 Systems of Timed Constraint Automata

In this section, we present a *compositional* product construction for TCA, which allows to easily build complex connectors by composing simpler ones. We also present a hiding operation on ports, which allows to hide ports on which two (or more) TCA synchronise from the environment. The idea is that the ports become "internal" to the new (composed) connector.

The idea of TCA synchronisation is as follows: within a system of TCA, two automata synchronise (communicate, exchange data values) if the port sets of the involved transitions coincide on common ports. Invisible transitions, and local visible transitions (i.e., transitions involving ports known to only one automaton) can be executed independently of other automata. This gives rise to the following definition.

Definition 2.3.9 (Product of TCA). Let $\mathfrak{T}_1$, $\mathfrak{T}_2$ be TCA over $\mathcal{D}ata_1$ and $\mathcal{D}ata_2$, respectively, with $\mathcal{X}_1{\cap}\mathcal{X}_2{=}\emptyset$, $\mathcal{D}_1{\cap}\mathcal{D}_2{=}\emptyset$, and $S_1{\cap}S_2{=}\emptyset$ (can be achieved by renaming the constituents in one of the TCA). The *product of* $\mathfrak{T}_1$ *and* $\mathfrak{T}_2$ is a new TCA $\mathfrak{T}_1{\bowtie}\mathfrak{T}_2{=}(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$ over data domain $\mathcal{D}ata_1{\cup}\mathcal{D}ata_2$, with $S{=}S_1{\times}S_2$, $s_0{=}(s_{0,1}{,}s_{0,2})$, $\mathcal{P}{=}\mathcal{P}_1{\cup}\mathcal{P}_2$, $\mathcal{X}{=}\mathcal{X}_1{\cup}\mathcal{X}_2$, $I{:}S{\to}\mathcal{CC}(\mathcal{X}_1{,}\mathcal{X}_2)$, with $I(s){=}I_1(s_1){\wedge}I_2(s_2)$ for $s{=}(s_1, s_2){\in}S$, $\mathcal{D}{=}\mathcal{D}_1{\cup}\mathcal{D}_2$, $\#{:}S{\to}2^{\mathcal{D}}$, with $\#((s_1, s_2)){=}\#_1(s_1){\cup}\#_2(s_2)$, and E is defined in (2.10) and (2.11), and the symmetric rule of the latter.

$$\frac{\begin{array}{c}(s_1, P_1, dc_1, cc_1, \lambda_1, s_1'){\in}E_1\\ (s_2, P_2, dc_2, cc_2, \lambda_2, s_2'){\in}E_2\\ P_1{\cap}\mathcal{P}_2 = P_2{\cap}\mathcal{P}_1, P_1{\neq}\emptyset, P_2{\neq}\emptyset, dc_1{\wedge}dc_2{\neq}\texttt{false}\end{array}}{((s_1, s_2), P_1{\cup}P_2, dc_1{\wedge}dc_2, cc_1{\wedge}cc_2, \lambda_1{\circ}\lambda_2, (s_1', s_2')){\in}E} \tag{2.10}$$

$$\frac{(s_1, P_1, dc_1, cc_1, \lambda_1, s_1'){\in}E_1, P_1{\cap}\mathcal{P}_2 = \emptyset, s_2{\in}S_2}{((s_1, s_2), P_1, dc_1, cc_1, \lambda_1, (s_1', s_2)){\in}E}, \tag{2.11}$$

Rule (2.10) captures the synchronisation of visible transitions: the nonempty port sets have to coincide on common ports, i.e. data flows through the same set of shared ports on both transitions. The case where $P_1 \cap \mathcal{P}_2 = P_2 \cap \mathcal{P}_1 = \emptyset$ (i.e., the set of shared ports is empty) represents a system step where each automaton performs a *local* visible transition (concurrent execution of independent actions). Rule (2.11) describes the execution of a local transition (visible or invisible) in one automaton, while the other automaton remains in its current location (is idle). The semantics of TCA, in particular the fifth and sixth row of (2.7), ensures that the values stored in memory cells of the idle automaton correctly carry over to the next location. Note that in case such a local transition is preceded by a time delay, the idle automaton actually performs a delay transition.

Example 2.3.10 (Product of TCA). An example for the product construction can be found in Figure 2.7: it shows the product TCA of two instances of the 1-bounded FIFO buffer (cf. Example 2.3.3), as show in Figures 2.5 and 2.6 (the two instances are identical except for renaming). The resulting TCA models a FIFO buffer with capacity 2, where each of the buffer cells has its own expiration timer (clocks x and x', respectively). The buffer accepts data through port p, and releases it through port r. The synchronisation on port q models the step where the data from the first buffer cell is transferred to the second buffer cell.

For readability, we have abbreviated the location names in Figure 2.7 with e, e', f and f', for *empty*, *empty'*, *full* and *full'*, respectively.

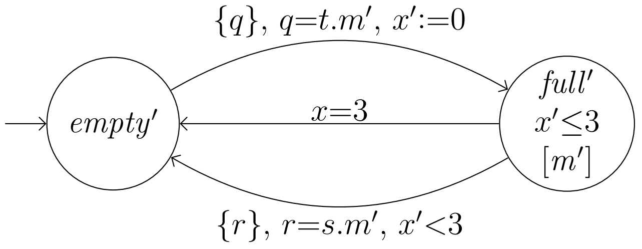


Figure 2.6: 1-bounded FIFO Buffer with Expiration, Second Instance

Proposition 2.3.11 (Product of TCA). The product of TCA is commutative and associative, up to isomorphy of location names.

Proof.

1. Commutativity follows from the commutativity of $\cup$ on port sets, the commutativity of $\circ$ on update maps over disjoint clock sets, and the commutativity of $\wedge$ on data and clock constraints.

2. Associativity follows from the associativity of $\cup$ on port sets, the associativity of $\circ$ on update maps over disjoint clock sets, the associativity of $\wedge$ on data and clock constraints, and the fact that for $P_i \subseteq \mathcal{P}_i$, $i{=}1,2,3$, if we have $P_2 \cap \mathcal{P}_3 = P_3 \cap \mathcal{P}_2$, $P_1 \cap (\mathcal{P}_2 \cup \mathcal{P}_3) = \mathcal{P}_1 \cap (P_2 \cup P_3)$ and $P_1 \cap \mathcal{P}_2 = P_2 \cap \mathcal{P}_1$, then holds $P_3 \cap (\mathcal{P}_1 \cup \mathcal{P}_2) = \mathcal{P}_3 \cap (P_1 \cup P_2)$.

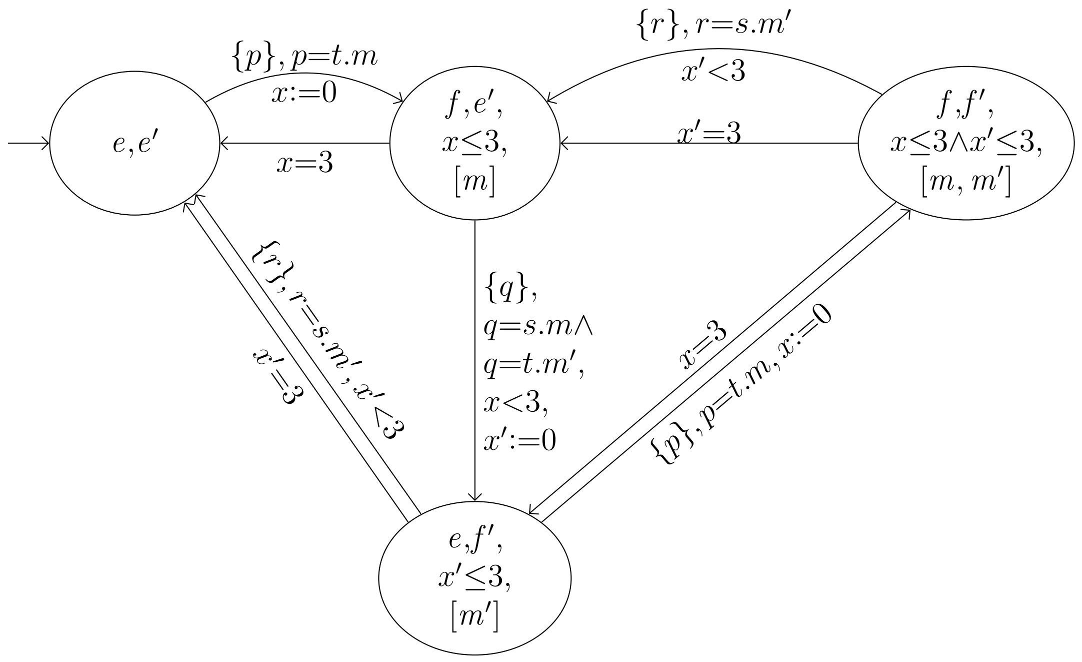


Figure 2.7: Product of two 1-bounded FIFO Buffers

□

It may be required to hide some ports of a TCA from the environment. For example, in the product construction, the common ports (on which the TCA synchronise) could be considered to become "internal" ports, and thus not be visible from the outside anymore. Consider for example port q in Example 2.3.10. The hiding operation removes all information about a set of ports $O{\subseteq}\mathcal{P}$ from a TCA. To ensure correct timed behaviour of transitions with port sets $P{\subseteq}O$—namely that such transitions may only be taken after a positive amount of time—we need to introduce an additional clock.

Definition 2.3.12 (Hiding in TCA). Let $\mathfrak{T}$ be a TCA, $x{\notin}\mathcal{X}$ a fresh clock, and $O{\subseteq}\mathcal{P}$. The *hiding of O in $\mathfrak{T}$* yields a new TCA $\mathfrak{T}\backslash_O{=}(S, s_0, \mathcal{P}\backslash O, \mathcal{X}{\cup}x, I, \mathcal{D}, \#, E')$, where E' is given in (2.12) and (2.13).

$$\frac{(s, P, dc, cc, \lambda, s'){\in}E, ((P{=}\emptyset) \vee (P\backslash O{\neq}\emptyset))}{(s, P\backslash O, dc\backslash_O, cc, \lambda_x, s'){\in}E} \tag{2.12}$$

$$\frac{(s, P, dc, cc, \lambda, s'){\in}E, \emptyset{\neq}P{\subseteq}O}{(s, \emptyset, \mathtt{true}, cc{\wedge}(x{>}0), \lambda_x, s'){\in}E'} \tag{2.13}$$

Here, $dc\backslash_O$ denotes the data constraint which is derived from dc by replacing all literals dc', with $dc'{=}(\mathbf{D}{\sim}\mathbf{D}')$ or $dc'{=}\neg(\mathbf{D}{\sim}\mathbf{D}')$, by $\mathtt{true}$ iff $p{\in}\mathcal{P}|_{dc'}$ (cf. Definition 2.1.7), for all $p{\in}O$. The update map λ_x updates the new clock x to zero, and agrees with λ on other clocks: $\lambda_x(x'){=}\lambda(x')$ if $x'{\in}\mathcal{X}$, and $\lambda_x(x){=}0$ otherwise.

The basic idea is to obtain transitions of $\mathfrak{T}\backslash_O$ from transitions of $\mathfrak{T}$ by reducing the port set (and data constraints) to ports not contained in O (2.12). If the resulting transition in $\mathfrak{T}\backslash_O$ is invisible, while the underlying transition in $\mathfrak{T}$ is visible (2.13), the new clock x is used to ensure correct timed behaviour: since x is updated to zero on all transitions, the additional constraint $(x{>}0)$ ensures the elapse of a positive amount of time before the (now invisible) transition can be taken.

The semantics of a system of two TCA $\mathfrak{T}_1$ and $\mathfrak{T}_2$ (with disjoint sets of clocks, locations and memory cells, as required in Definition 2.3.9) is defined as the semantics (Definition 2.3.6) of the corresponding product automaton $\mathfrak{T}_1 \bowtie \mathfrak{T}_2$ (Definition 2.3.9), i.e., the set of runs of the associated LTS $\mathfrak{S}_{\mathfrak{T}}$ (Definition 2.3.5). In (2.14), we show a run of the product automaton from Figure 2.7 of length 5.

$$\begin{aligned}
&\left\langle (e,e'), \bot, \substack{x=0\\x'=0} \right\rangle \xrightarrow{\{p\},\substack{p=3\\t.m=3},4} \left\langle (f,e'), m{=}3, \substack{x=0\\x'=4} \right\rangle \xrightarrow{\{q\},\substack{q=3\\s.m=3\\t.m'=3},2.5} \\
&\left\langle (e,f'), m'{=}3, \substack{x=2.5\\x'=0} \right\rangle \xrightarrow{\{p\},\substack{p=5\\t.m=5\\s.m'=3\\t.m'=3},0.5} \\
&\left\langle (f,f'), \substack{m=5\\m'=3}, \substack{x=0\\x'=0.5} \right\rangle \xrightarrow{\emptyset,\substack{s.m=5\\t.m=5},2.5} \\
&\left\langle (f,e'), m{=}5, \substack{x=2.5\\x'=3} \right\rangle \xrightarrow{\emptyset,\bot,0.5} \left\langle (e,e'), \bot, \substack{x=3\\x'=3.5} \right\rangle
\end{aligned} \tag{2.14}$$

This run correspond to an execution of the buffer as follows: first, after a delay of 4, it receives data element 3 through port p, and stores it in memory cell m of target location (f,e'). Next, it transfers the data item to memory cell m' of (e,f') through port q, then receives data item 5 through port p and stores it in the memory cell m of target location (f,f'). The buffer is now completely full. In location (f,f'), the buffer delays for 2.5, which—together with previous delays—increases the value of clock x' to 3, such that the buffer takes the invisible transition to location (f,e'), loosing data item 3. After delaying for another 0.5, the value of clock x reaches the threshold 3 as well, the second data item is lost as well, and the buffer returns to the initial location (e,e').

Remark 2.3.13 (Size of the Product). The result of the product construction for TCA, as presented in Definition 2.3.9, is exponential in the worst case. For model checking/verification, we present a linear technique to avoid the explicit construction of the product automaton in Section 3.1.3.

2.3.4 Discussion

TCA arise from combining the real-time concepts of TA, as presented in Section 2.2, with the coordination concepts of constraint automata (CA) [ABRS04]. We further extend the basic definition from [ABdBR07, Kem11] with location memory, which allows to reason about and base the coordination pattern on data values which have been exchanged prior to the current step.

In this way, TCA are specially tailored for implementing coordinating connectors in networks where timed components communicate by exchanging data values through multiple channels. The behaviour of the network is given by synchronisation between channel ends (ports). While the functionality of channels is often limited to synchrony and (FIFO) buffering, TCA allow connectors with arbitrary behaviour. These connectors provide exogenous coordination, by imposing a certain communication pattern—for example reordering or delays—on associated components. TCA are compositional, which allows to easily build complex connectors out of simpler ones.

One of the major advantages of TCA (which is also one of the major distinctions to TA) is the fact that they provide *true concurrency.* Most action-based (coordination) models, like e.g. finite state machines, timed I/O automata, but also TA, permit only a single action per transition. As a consequence, synchrony, and concurrent execution of actions in the parallel composition, is reduced to arbitrary interleavings plus nondeterminism. Especially for timed systems involving exchange of data values—aside from being unintuitive—this does not correctly capture the nature of distributed systems, since it imposes a sequential order on actions which conceptually happen at the same time. Furthermore, the sequential order might influence the availability of data items, depending on the execution order of transitions. What is more, from a technical point of view, the presence of all possible interleavings amplifies the state explosion problem. In contrast, TCA allow sets of actions on each transition, which permits *true concurrency*, as this directly models (truly atomic) synchronous communication through different ports.

Under true concurrency, all actions which happen at the same time (atomically) collapse into a single transition. As a consequence, a positive amount of time has to elapse before every visible transition (i.e., transitions involving visible data flow). This allows for a more "concise" semantics compared to TA: transitions in the associated LTS $\mathfrak{S}_{\mathfrak{A}}$ of a TA $\mathfrak{A}$ correspond to *either* the execution of a transition of $\mathfrak{A}$, *or* to a system delay,[7] cf. Definition 2.2.4. In contrast, *every* transition in the associated LTS $\mathfrak{S}_{\mathfrak{T}}$ of a TCA $\mathfrak{T}$ corresponds to the execution of a transition in $\mathfrak{T}$, possibly preceded by a time delay, cf. Definition 2.3.5. Thus, on average, runs of $\mathfrak{S}_{\mathfrak{T}}$ are shorter than runs of $\mathfrak{S}_{\mathfrak{A}}$ while containing the same number of visible events, i.e., providing the same "information content".

2.4 Timed Network Automata

In this section, we define the third system model: Timed Network Automata (TNA). TNA [Kem10] can be seen as an extension of TCA with *environmental constraints* (see for example [CCA07, Cos10]). Such constraints are imposed on the TNA by the surrounding network (hence the name), and capture information about whether the environment is ready to communicate. Thus, presence and absence of dataflow in the connector (which is modelled by the TNA) no longer depend on the internal state of the automaton only, but also take into account whether the environment is ready to communicate. Thereby, the behaviour of the environment is represented through constraints on the transitions of the TNA, i.e., there is *no need* to specify the environment explicitly. In this way, TNA provide a modular framework for compositional construction of a real-time generalisation of dataflow networks.

[7]Remember that subsequent delays can be combined into a single transition, cf. Remark 2.2.7)

The underlying idea of TNA is that absence of dataflow needs a reason. For example, a simple empty buffer (without any constraints on the input) should always be ready to accept data, communication can only be delayed if the reason comes from the outside the connector (if the environment does not provide a data item, i.e., is not ready to communicate). To capture where the reason for delaying the communication comes from, TNA distinguish between input ports and output ports.

2.4.1 Syntax of Timed Network Automata

The (externally visible) behaviour of a TNA is given by the possible dataflow through its (externally visible) ports, which not only depends on the TNA itself, but also on the environment it occurs in. In particular, there needs to be a reason for the absence of dataflow, from either the TNA or the environment; if both are ready to communicate, dataflow cannot be delayed. Consequently, we define the environmental constraints over the ports of a TNA. Essentially following the three-colouring idea presented in [CCA07], we define three different states of ports—called colours[8]—which not only capture presence and absence of dataflow, but in case of no dataflow also describe where the reason for delaying the communication comes from.

Definition 2.4.1 (Colours, Colourings). Let $\mathcal{P}$ be a finite set of ports, $\mathcal{Q}\subseteq\mathcal{P}$, and $p\in\mathcal{P}$. A *colouring* $\mathbb{c}\in\mathbb{C}(\mathcal{P})$ *over* $\mathcal{P}$ is a mapping $\mathbb{c}:\mathcal{P}\to\mathbb{C}lr$, assigning to each port $p\in\mathcal{P}$ a colour from the *set of colours* $\mathbb{C}lr=\{$——, -!-, -?-$\}$. We denote the colouring of a port p (i.e., the colour assigned to that port) by p:——, p:-!- and p:-?-, respectively. Port p is called *active (under colouring* $\mathbb{c}$*)* iff $\mathbb{c}(p)=$——, and *inactive (under colouring* $\mathbb{c}$*)* otherwise.

The *restriction* $\mathbb{c}|_{\mathcal{Q}}$ *of colouring* $\mathbb{c}$ *from* $\mathcal{P}$ *to* $\mathcal{Q}$ is a colouring that agrees with $\mathbb{c}$ on ports in $\mathcal{Q}$, and is undefined otherwise, i.e., $\mathbb{c}|_{\mathcal{Q}}:\mathcal{Q}\to\mathbb{C}lr$, $\mathbb{c}|_{\mathcal{Q}}(p)=\mathbb{c}(p)$, $p\in\mathcal{Q}$.

We may write colourings in either orientation (i.e., p:—— or ——:p), and we may omit the port name p if it is clear from the context. Further, we may write $\mathbb{C}$ if $\mathcal{P}$ is clear from the context.

The intended idea of the colourings of a port p is to denote *dataflow through* p (p:——) and *delay on* p, with the underlying TNA to which the port belongs either providing (p:-!-) or getting (p:-?-) a reason for the delay on its port p. Intuitively, p:-?- means that the TNA cannot actively delay dataflow through p, instead, delay requires a reason from the *outside*. On the other hand, p:-!- denotes that the TNA *itself* delays the communication, for example because no data is available to be transmitted.

A TNA consists of the externally visible ports, plus the internal behaviour. We specify this internal behaviour by means of finite automata, which are equipped with real-valued clocks. As for TA (cf. Section 2.2), we assume location changes to be instantaneous, time may only elapse while the automaton remains in one of

[8]We here adopt the term *colour* and related notions, as introduced in [CCA07], to avoid confusion with other uses of the word *state*.

its locations.[9] The delays may depend on environmental constraints. Therefore, we specify the admissible delays as explicit transitions of the automaton modelling the internal behaviour of the TNA.

Similar to TCA, TNA are equipped with a finite set of data variables $\mathcal{D}ata$. These can be used by locations, to store data values for use in subsequent steps. In addition, data constraints on transitions may reason about data variables which are not used by source or target location of the transition (this was not allowed for TCA, cf. Definition 2.3.1). This feature is primarily used for conciseness of the definition of TNA composition, but also to retain information when hiding ports from the environment (cf. Definitions 2.4.13 and 2.4.14).

Recalling Definitions 2.1.2, 2.1.4, 2.1.5, 2.1.7 and 2.1.8, we define TNA as follows.

Definition 2.4.2 (Timed Network Automaton). A *TNA* $\mathfrak{N}$ *over data domain* $\mathcal{D}ata$ is a tuple $\mathfrak{N}{=}(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, with S a finite set of locations, $s_0{\in}S$ the initial location, $\mathcal{P}{=}\mathcal{P}^r\dot{\cup}\mathcal{P}^w$ a finite set of ports, with $\mathcal{P}^r$ and $\mathcal{P}^w$ disjoint sets of read respectively write ports, $\mathcal{X}$ a finite set of real-valued clocks, $I{:}S{\to}\mathcal{CC}(\mathcal{X})$ a function assigning a clock constraint (*location invariant*) to every location, $\mathcal{D}$ a finite set of data variables, $\#{:}S{\to}2^{\mathcal{D}}$ a function assigning to each location the set of data variables it may use, and $E{\subseteq}(S{\times}\mathbb{C}(\mathcal{P}){\times}\mathcal{DC}(\mathcal{P}{,}\mathcal{D}){\times}\mathcal{CC}(\mathcal{X}){\times}\Lambda(\mathcal{X}){\times}S)$ the finite transition relation. The set $\mathcal{P}$ of ports is also called the *external interface of* $\mathfrak{N}$.

An element $e{=}(s, \mathbb{c}, dc, cc, \lambda, s'){\in}E$ describes a transition from *source location* s to *target location* s', with dataflow/delay according to colouring $\mathbb{c}$, enabled under *data guard* dc and *clock guard* cc, and updating all clocks according to the update map λ. For every such transition, we require that dc only reasons about active ports, i.e., $dc{\in}\mathcal{DC}(\mathcal{Q}, \mathcal{D})$, with $\mathcal{Q}{=}\{p{\in}\mathcal{P} \mid \mathbb{c}(p){=}\!\!-\!\!-\!\!-\}$. We require both dc and cc to be satisfiable. Transition e is called *delay* iff $s'{=}s$, $\lambda{=}id$ (identity mapping), and $\mathbb{c}(p){\neq}\!-\!\!-\!\!-$ for all $p{\in}\mathcal{P}$, and *communication* otherwise. Two TNA are called *disjoint* if the respective constituents (i.e., locations, ports, clocks, data variables) are disjoint.

A communication $(s, \mathbb{c}, dc, cc, \lambda, s')$ describes the conditions for a location change from s to s', while a delay $(s, \mathbb{c}, dc, cc, id, s)$ describes the conditions under which $\mathfrak{N}$ may delay in location s—namely, as long as guard cc is satisfied, and a reason for delay exists which satisfies colouring $\mathbb{c}$. Note that the data constraint dc on delays may not involve ports (since no dataflow is allowed), but only data variables (the duration of the delay or whether the delay is possible at all may still depend on the contents of the memory cells).

Remark 2.4.3 (Use of Data Variables). As for TCA, we not impose any restrictions on the set of data variables used by locations, cf. Remark 2.3.2. This may lead to the same ambiguities as described in the Remark, if for a transition $(s, \mathbb{c}, dc, cc, \lambda, s')$, both locations use the same data variable d, and d occurs in dc. We use the same conventions as introduced for TCA (cf. Remark 2.3.2 again):

[9]Yet, it is straightforward to model duration of data flow in a TCA like style, for example by adding a fresh clock and appropriate clock guards >0 on transitions, which forces the automaton to delay in every location.

we prefix the occurrence of a data variable d in dc by "s." if it belongs to the source location, and by "t." if it belongs to the target location. That means, instead of $(d{=}\mathbb{d}_1) \wedge (d{=}\mathbb{d}_2)$ (which would obviously evaluate to `false`), we write $(s.d{=}\mathbb{d}_1) \wedge (t.d{=}\mathbb{d}_2)$, and any data assignment δ will consider $s.d$ and $t.d$ to be different elements. A data variable without prefix is used in the data constraint only and does not correspond to a memory cell of source or target location.

As for TCA before, unconstrained memory cells of the target location can take random values, and we may explicitly require data variables to be empty.

Example 2.4.4. An example for a TNA can be found in Figure 2.8. We model again the 1-bounded FIFO buffer with expiration from Example 2.3.3, but now in addition take into account environmental constraints. To model dataflow and environmental constraints, the TNA has a read port r, through which it receives the data item, a write port w, through which it releases the data item to the environment again, and uses a data variable m, to model the memory cell in location *full*. We denote communications by solid lines, and delays by dashed lines.

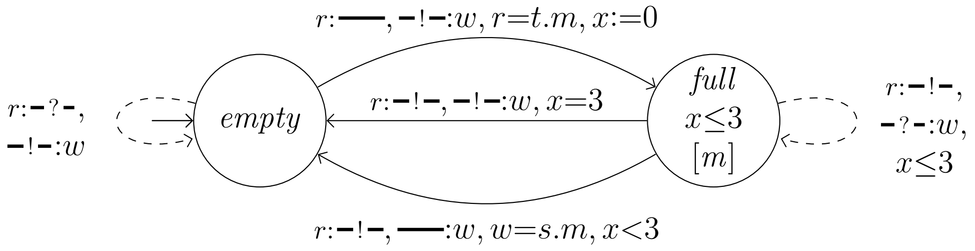


Figure 2.8: 1-bounded FIFO Buffer with Expiration and Environmental Constraints

The general idea of the TNA in Figure 2.8 is identical to the TCA presented in Figure 2.5. The three communications correspond almost directly to the three transitions in the TCA. On each communication, the inactive port always provides (and never requires) a reason to delay. On the upper communication (from *empty* to *full*), for example, this is due to the fact that the buffer is empty, so no data can flow out of it (through w), so the TNA itself provides a reason to delay on write port w. The reason can be read as "no data available". The explanation for the lower transition is similar. On the middle transition, both ports provide a reason to delay: port r cannot be active, since the buffer is full and no (more) data can be accepted through r. The fact that port w is inactive models the loss of the data item: once the deadline of 3 time units is reached, the data item is lost and cannot be sent through w anymore.

To handle environmental constraints, we add two delays, one for each location. The delay in location *empty* models the fact that no data item can be written to the environment (since there is no data, i.e., the buffer is empty), therefore, w provides a reason to delay. Since the TNA itself is ready to accept data through r, the read port requires a reason for delay. Stated differently: if the buffer is empty, it is always ready to accept data. The explanation for the delay in *full* is symmetrical, the additional clock constraint $x{\leq}3$ is used to enforce the expiration threshold of 3 time units.

Notation 2.4.5 (TNA). If not state otherwise, we shall assume the constituents of a TNA $\mathfrak{N}$ to be denoted as $\mathfrak{N}{=}(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$, with port set $\mathcal{P}{=}\mathcal{P}^r{\cup}\mathcal{P}^w$, and of a TNA $\mathfrak{N}_i$ to be denoted as $\mathfrak{N}_i{=}(S_i, s_{0,i}, \mathcal{P}_i, \mathcal{X}_i, I_i, \mathcal{D}, \#_i, E_i)$, with port set $\mathcal{P}_i{=}\mathcal{P}^r_i{\cup}\mathcal{P}^w_i$, for $i{\in}\mathbb{N}$. We lift $\#$ to reason about sets of locations, and we may omit curly braces: $\#(s, s'){=}\#(s){\cup}\#(s')$.

By $CC(\mathcal{X})|_{\mathfrak{N}}$, we denote the set of clock constraints (over clock set $\mathcal{X}$) that occur in a TNA $\mathfrak{N}$ (invariants or guards). Equivalently, by $DC(\mathcal{P},\mathcal{D})|_{\mathfrak{N}}$, we denote the set of data constraints over port set $\mathcal{P}$ and data variables $\mathcal{D}$ that occur in a TNA $\mathfrak{N}$.

In the graphical representation of TNA, we use (as before) assignment rather than functional notation for updates, and we omit guards equal to `true` as well as identity updates. We denote a set of memory cells $\{m_1, \ldots, m_n\}$ associated to a location as a bracketed list $[m_1, \ldots, m_n]$.

2.4.2 Semantics of Timed Network Automata

The semantics of a TNA is given by the set of runs of the associated LTS $\mathfrak{S}_{\mathfrak{N}}$.

Definition 2.4.6 (Associated LTS). Let $\mathfrak{N}$ be a TNA. The *associated LTS* $\mathfrak{S}_{\mathfrak{N}}$ is a tuple $\mathfrak{S}_{\mathfrak{N}}{=}(\mathcal{Q}, q_0, \rightarrow)$, with $\mathcal{Q}{\subseteq}(S{\times}DA(\mathcal{D}){\times}\mathcal{V}(\mathcal{X}))$ the set of configurations, such that for every $(s, \delta, \nu){\in}\mathcal{Q}$, $\delta{\in}DA(\mathcal{D})$, with $\delta(m){=}\bot$ if $m{\notin}\#(s)$, and $\nu{\models}I(s)$. The initial configuration is $q_0{=}\langle s_0, \mathbf{0}, \mathbf{0}\rangle$, with $\mathbf{0}(x){=}0$ for all $x{\in}\mathcal{X}$ and $\mathbf{0}(m){=}\bot$ for all $m{\in}\#(s_0)$, and the transition relation $\rightarrow\ {\subseteq}(\mathcal{Q}{\times}\mathbb{C}{\times}(DA(\mathcal{P},\mathcal{D}){\cup}\texttt{Time}){\times}\mathcal{Q})$ is given by

$$\frac{\begin{array}{l} (s, \mathbb{c}, dc, cc, \lambda, s'){\in}E, \\ \nu{\models}cc, \nu[\lambda]{\models}I(s') \\ \bar{\delta}{\in}DA(\mathcal{P},\mathcal{D}) : \bar{\delta}{\models}dc \\ \bar{\delta}(s.m){=}\begin{cases} \delta(m), & \text{if } m{\in}\#(s) \\ \bot, & \text{otherwise} \end{cases} \\ \bar{\delta}(t.m){=}\delta(m) \text{ if } m{\in}\#(s), m{\in}\#(s'), t.m{\notin}\mathcal{D}|_{dc} \\ \bar{\delta}(d){=}\bot \text{ if } d{\in}\mathcal{D}\backslash(\mathcal{D}|_{dc}{\cup}\#(s, s')) \\ \bar{\delta}(p){=}\bot \text{ iff } \mathbb{c}(p){\neq}\text{---} \\ \delta'(m){=}\bar{\delta}(t.m) \text{ for all } m{\in}\#(s') \end{array}}{\langle s, \delta, \nu\rangle \xrightarrow{\mathbb{c},\bar{\delta}} \langle s', \delta', \nu[\lambda]\rangle} \tag{2.15}$$

$$\frac{\begin{array}{l} (s, \mathbb{c}, dc, cc, id, s){\in}E, \\ t{>}0, t{\geq}t'{\geq}0 : \nu{+}t'{\models}cc, \nu{+}t'{\models}I(s), \\ \delta{\models}dc \end{array}}{\langle s, \delta, \nu\rangle \xrightarrow{\mathbb{c},t} \langle s, \delta, \nu{+}t\rangle} \tag{2.16}$$

A *run of* $\mathfrak{S}_{\mathfrak{N}}$ *(starting in configuration* q_0*)* is a sequence of transitions $q_0 \xrightarrow{\gamma_0} q_1 \xrightarrow{\gamma_1} \ldots$, with $\gamma_i{\in}(\mathbb{C}{\times}DA(\mathcal{P},\mathcal{D})){\cup}(\mathbb{C}{\times}\texttt{Time})$. A run is called *initial* if it starts in the initial configuration q_0, it is called *loop-free* if all configurations are different.

Transitions of $\mathfrak{S}_{\mathfrak{N}}$ directly correspond to the two types of transitions of TNA, cf. Definition 2.4.2: an *action transition* (2.15) describes the firing of an instantaneous communication in $\mathfrak{N}$, with dataflow according to colouring $\mathbb{c}$. The clock guard cc is satisfied before the execution of the transition, and the invariant of the target location is satisfied after the execution, i.e., after updating the clocks (second row). The data assignment satisfies the data guard dc (third row). As for TCA (cf. (2.7)), the values of memory cells before the execution of the transition remain unchanged if used by the source location s, otherwise, they are empty (fourth row). The values of memory cells which are used in both source and target location carry over to s' if they are not modified in the data constraint, if a memory cell is not used in the target location, its value after the execution of the transition is unspecified (fifth row). Data variables which are not used in source or target location, nor in the data constraint, are empty (sixth row), and data may only be pending at active ports (sixth row). Data variables used by the target location s' obtain their values according to the data assignment on the transition. A *delayed action transition* (2.16) describes the firing of a delay of $\mathfrak{N}$: if a reason for delay exists that satisfies colouring $\mathbb{c}$, the TNA can delay for a positive amount of time t, during which the invariant $I(s)$ and the clock guard cc need to be satisfied at all points (second row),[10] and the data guard dc has to be satisfied. Since by definition, all ports are inactive (i.e., have a colouring $\neq$——) during the execution of a delay, the data values stored in data variables do not change (new data values can only be received through active ports), i.e., the data assignment δ is identical in both configurations, it can only reason about memory cells that are used by s.

Definition 2.4.7 (Semantics of Timed Network Automata). Let $\mathfrak{N}$ be a TNA, $\mathfrak{S}_{\mathfrak{N}}$ the associated LTS. The *trace semantics of* $\mathfrak{N}$ is given by the set $Run_{\mathfrak{N}}$ of initial runs (also called *executions*) of $\mathfrak{S}_{\mathfrak{N}}$. With $Run_{\mathfrak{N}k}$, we denote the set of finite prefixes of elements of $Run_{\mathfrak{N}}$ of (at most) length k.

Example 2.4.8 (Execution of a Timed Network Automaton). In (2.17), we show an execution of the TNA from Example 2.4.4 of length 8. Again (cf. Example 2.3.7), we omit data assignments on transitions which evaluate to $\perp$, and we abuse the single symbol $\perp$ to denote the empty data assignment, which assigns $\perp$ to all elements. In order not to clutter up the illustration, we further omit port names; the colourings correspond to port r on top, and port w below.

2.4.3 Systems of Timed Network Automata

In this section, we present a compositional product construction for TNA, which allows to build complex TNA out or simpler ones. The basic idea of the composition operator is to join sets of (read and write) ports, which conceptually yields invisible *internal ports.* Note that internal ports are theoretical constructs, in that they do not actually appear in the composed TNA. The notion of internal ports is used for explanatory purposes, and to reason about the validity of the composition.

[10]Due to convexity, it is actually enough to check the clock constraints at the beginning and at the end of the time delay only. We will use this fact in the representation, cf. Section 3.1.4.

$$
\begin{aligned}
&\langle \mathit{empty}, \bot, x{=}0\rangle \xrightarrow{\substack{\text{-?-}\\ \text{-!-}}^{,4}} \langle \mathit{empty}, \bot, x{=}4\rangle \xrightarrow{\substack{\text{——}\\ \text{-!-}},\substack{r=2\\ t.m=2}} \langle \mathit{full}, m{=}2, x{=}0\rangle \xrightarrow{\substack{\text{-!-}\\ \text{-?-}}^{,2.5}}\\
&\qquad\langle \mathit{full}, m{=}2, x{=}2.5\rangle \xrightarrow{\substack{\text{-!-}\\ \text{——}},\substack{w=2\\ s.m=2}} \langle \mathit{empty}, \bot, x{=}2.5\rangle \xrightarrow{\substack{\text{-?-}\\ \text{-!-}}^{,10}}\\
&\qquad\langle \mathit{empty}, \bot, x{=}10\rangle \xrightarrow{\substack{\text{——}\\ \text{-!-}},\substack{r=5\\ t.m=5}} \langle \mathit{full}, m{=}5, x{=}0\rangle \xrightarrow{\substack{\text{-!-}\\ \text{-?-}}^{,3}}\\
&\qquad\langle \mathit{full}, m{=}5, x{=}3\rangle \xrightarrow{\substack{\text{-!-}\\ \text{-!-}},\substack{w=\bot\\ s.m=5}} \langle \mathit{empty}, \bot, x{=}3\rangle
\end{aligned}
\tag{2.17}
$$

The intended behaviour of internal ports is to act as self-contained, stateless "pumping stations" [BSAR06], *merging* data from write ports, and *replicating* data to read ports. If data flows through an internal port, then it flows through *exactly one* underlying write port and through *all* underlying read ports. Absence of dataflow is subject to environmental constraints on the involved ports: if there is a reason for delay (-!-) on at least one read port (i.e., the TNA contributing the port provides a reason to delay on that port) or on all write ports, data cannot flow. Stated differently, a valid colouring of an internal port must not involve the colour -?- only. We do not restrict composition to one-to-one relations (as is done in [Arb04, CCA07, CPLA09], for example). On the contrary, we do not impose any restrictions on the number, type (read/write) or origin (same or different TNA) of ports to be merged; the only condition is that a port cannot be merged more than once. Though the composition of colourings would be slightly simpler in a one-to-one approach, our many-to-many composition provides a direct and more intuitive way of specifying compositions, for example for mergers, replicators or multi-synchronisations.

We now formalise these ideas.

Definition 2.4.9 (Merge Set, Validity of Colouring). Let $\mathcal{P}$ be a set of ports, $\mathcal{Q}{\subseteq}\mathcal{P}$ a subset, $\mathcal{Q}^r{\subseteq}\mathcal{P}^r$ and $\mathcal{Q}^w{\subseteq}\mathcal{P}^w$ the sets of read respectively write ports in $\mathcal{Q}$, and $\mathbb{c}{\in}\mathbb{C}(\mathcal{P})$ a colouring. If ports in $\mathcal{Q}$ are intended to be joined (merged), we call $\mathcal{Q}$ a *merge set (over $\mathcal{P}$)*, the resulting internal port is denoted as $p_{\prec\mathcal{Q}}$.

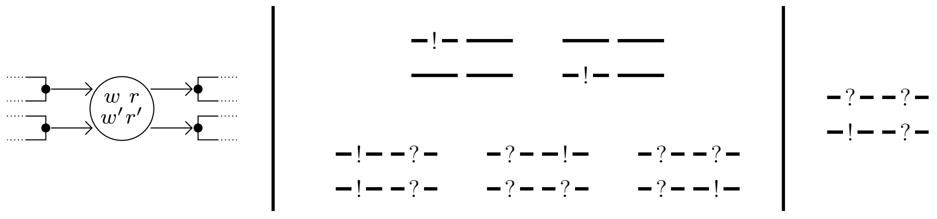
Colouring $\mathbb{c}$ is *valid over merge set $\mathcal{Q}$* (or *valid over $p_{\prec\mathcal{Q}}$*), if it satisfies the following conditions for all ports $w, w', r{\in}\mathcal{Q}$:

1. If $\exists w{\in}\mathcal{Q}^w{:}\mathbb{c}(w){=}\text{——}$, then $\forall r{\in}\mathcal{Q}^r{:}\mathbb{c}(r){=}\text{——}$, and $\forall w'{\in}\mathcal{Q}^w, w'{\neq}w{:}\mathbb{c}(w'){\neq}\text{——}$
2. If $\exists r{\in}\mathcal{Q}^r{:}\mathbb{c}(r){=}\text{——}$, then $\exists w{\in}\mathcal{Q}^w{:}\mathbb{c}(w){=}\text{——}$
3. If $\nexists w{\in}\mathcal{Q}{:}\mathbb{c}(w){=}\text{——}$, then $((\forall w'{\in}\mathcal{Q}^w{:}\mathbb{c}(w'){=}\text{-!-})$ or $(\exists r{\in}\mathcal{Q}^r{:}\mathbb{c}(r){=}\text{-!-}))$

Colouring $\mathbb{c}$ is valid over a set $\mathcal{Q}'$ of disjoint merge sets $\mathcal{Q}'{=}\{\mathcal{Q}_1, \ldots, \mathcal{Q}_n\}$, $n{\geq}1$, if it is valid over each $\mathcal{Q}_i$.

Only valid colourings correctly reflect/model the aforementioned behaviour of internal ports: conditions 1 and 2 in Definition 2.4.9 describe simultaneous dataflow through exactly one write port and all read ports of $p_{\prec\mathcal{Q}}$. Condition 3 describes the propagation of environmental constraints (delays): no dataflow is possible only if either all write ports or at least one read port in $\mathcal{Q}$ provide a reason to delay.

Example 2.4.10 (Validity of Colourings). To illustrate validity of colourings over internal ports, consider a merge set which contains two write ports w, w', and two read ports r, r'. The internal port resulting from this merge set is conceptually depicted on the left side of the illustration below (note that the ports do not need to come from different TNA, as suggested in the picture). Some of the valid colourings of the internal port are given in the middle (the layout of colourings reflects the layout of the ports on the left), there are 17 valid colourings in total.



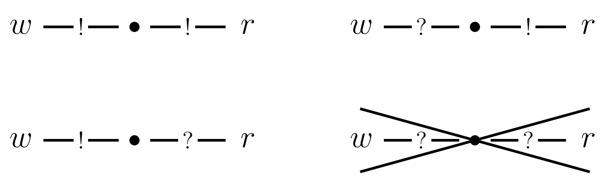
The colouring in the lower right, for example, can be read as follows: if read port r' provides a reason to delay (–!–), while the other ports can only delay if they get a reason (–?–), then the reason from r' is enough to delay dataflow through the internal port.

The colouring on the right side of the illustration is an example for an invalid colouring. It corresponds to a situation where there is no actual reason for delay: neither of the read ports can delay the communication, both read ports require a reason for delay. Write port w' provides a reason for delay, but write port w requires a reason, that means, w is actually ready to communicate. Thus, data could flow through ports w, r and r', since they are all ready to communicate, so this "no flow" colouring is not valid.

The *flip rule*, introduced in [CCA07], is used to reduce the size of composed TNA, by identifying redundant (with respect to compositionality) colourings.

Remark 2.4.11 (Flip Rule). Let $\mathcal{P}$ be a set of ports, $p \in \mathcal{P}$, and $\mathbb{c}_1, \mathbb{c}_2 \in \mathbb{C}(\mathcal{P})$. If $\mathbb{c}_1$ and $\mathbb{c}_2$ are identical except for $\mathbb{c}_1(p)=$–!– and $\mathbb{c}_2(p)=$–?–, then $\mathbb{c}_2$ is redundant and can be removed: the set of colourings with which $\mathbb{c}_2$ can compose over p is a strict subset of the set of colourings with which $\mathbb{c}_1$ can compose over p.

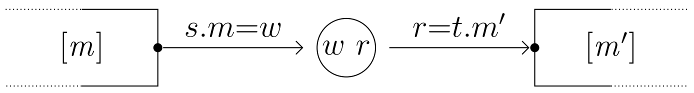
Example 2.4.12 (Flip Rule). To illustrate the flip rule, consider the following simple example.



A write port w with colouring w:-!- can compose with a read port r under both possible “no flow” colourings of r (left side). But the colouring w:-?- can compose with the colouring -!-:r only (right side), the colouring in the lower right is not valid, since it corresponds to a situation where both ports delay, without actually having a reason to delay (cf. also Example 2.4.10). Therefore, the colouring w:-?- is redundant and can be removed.

As explained above, internal ports are theoretical constructs, and do not actually appear in the composed TNA. In particular, the ports in the merge set are removed from the composed TNA. Consequently, we would have to remove the data constraints on these ports as well, since data constraints may only reason about (active) ports. If we want to ensure that data values are transmitted correctly over internal ports, we need to preserve the information from data constraints on ports in the merge set; simply removing such data constraints would not correctly reflect the intended behaviour.

To illustrate this, consider the following example: a TNA $\mathfrak{N}_1$ has a data value stored in a memory cell m, then writes it to the environment through a write port w. The corresponding transition contains a data constraint of the form $s.m{=}w$, to ensure the value that is written through w is indeed that value that was contained in m. A second TNA $\mathfrak{N}_2$ reads a data value from the environment into a memory cell m', the corresponding transition contains a data constraint of the form $r{=}t.m'$. Ports w are r are to be merged, i.e., there exists a merge set $\mathcal{Q}{=}\{w, r\}$. This scenario is conceptually depicted as



The expected result of merging ports w and r is to create a permanent link, with the expected behaviour that the data value that arrives in m' is always identical to the one that was contained in m. Yet, if we simply remove the two data constraints, we cannot guarantee this behaviour, since the correlation between m and m' is completely lost. So instead, we would like to obtain a data constraint (that is equivalent to) $s.m{=}t.m'$. For this, we define reduced (with respect to a merge set) data constraints.

Definition 2.4.13 (Reduced Data Constraint). Let $\mathcal{P}$ be a set of ports, $\mathcal{Q}{\subseteq}\mathcal{P}$ a merge set over $\mathcal{P}$, $\mathcal{Q}'{=}\{\mathcal{Q}_1, \ldots, \mathcal{Q}_n\}$, $n{\geq}1$, a set of disjoint merge sets over $\mathcal{P}$, $dc{\in}\mathcal{DC}(\mathcal{P})$ a data constraint, $d, d_1, \ldots, d_n{\in}\mathcal{D}$ distinct data variables not occurring in dc, i.e., $d, d_1, \ldots, d_n{\notin}\mathcal{D}|_{dc}$. The *reduced data constraint $dc|_{\mathcal{Q}}$ of dc (with respect to $\mathcal{Q}$)* is obtained by replacing every occurrence of a port $q{\in}\mathcal{Q}$ in dc by data variable d. The *reduced data constraint $dc|_{\mathcal{Q}'}$ of dc (with respect to $\mathcal{Q}'$)* is obtained by replacing every occurrence of port $q_i{\in}\mathcal{Q}_i$ by data variable d_i.

The reduced data constraint $dc|_{\mathcal{Q}}$ removes the occurrence of all ports in the merge set $\mathcal{Q}$, while preserving the information on admissible data values. By replacing *all* occurrences of ports $q{\in}\mathcal{Q}$ by the same data variable d, we ensure that the transmitted data value is the same on all ports contained in the merge set. Note that the constraints about whether dataflow is possible at all are already covered by valid colourings (see Definition 2.4.9).

We now have all the necessary concepts for defining the composition of TNA. The basic idea of TNA composition is along the same lines as the standard cross product in other automata models: the sets of read and write ports are joined, the ports in the merge sets are removed from the TNA, and the colourings of the involved transitions are composed. The composition of colourings (over disjoint port sets) is defined by standard function composition: for two colourings $\mathbb{c}_1{\in}\mathcal{P}_1$ and $\mathbb{c}_2{\in}\mathcal{P}_2$, with $\mathcal{P}_1{\cap}\mathcal{P}_2{=}\emptyset$, the *composition* $\mathbb{c}_1{\cup}\mathbb{c}_2$ is a new colouring $\mathbb{c}{=}\mathbb{c}_1{\cup}\mathbb{c}_2{\in}\mathbb{C}(\mathcal{P}_1{\cup}\mathcal{P}_2)$, with $\mathbb{c}(p){=}\mathbb{c}_1(p)$ iff $p{\in}\mathcal{P}_1$, and $\mathbb{c}(p){=}\mathbb{c}_2(p)$ iff $p{\in}\mathcal{P}_2$, for all ports $p{\in}\mathcal{P}_1{\cup}\mathcal{P}_2$.

Definition 2.4.14 (TNA Composition). Let $\mathcal{N}{=}\{\mathfrak{N}_1,\dots,\mathfrak{N}_k\}$, $k{\geq}1$, be a set of disjoint TNA, $\mathcal{Q}{=}\{\mathcal{Q}_1,\dots,\mathcal{Q}_n\}$, $n{\geq}1$, be a set of disjoint merge sets over $\bigcup\mathcal{P}_i$, $i{=}1,\dots,k$. The *composition of the* $\mathfrak{N}_i$ *over* $\mathcal{Q}$, denoted $\mathfrak{N}_1{\bowtie_\mathcal{Q}}\dots{\bowtie_\mathcal{Q}}\mathfrak{N}_k$ (or simply $\mathcal{N}{\bowtie_\mathcal{Q}}$), is a new TNA $\mathcal{N}{\bowtie_\mathcal{Q}}{=}(S,s_0,\mathcal{P},\mathcal{X},I,\mathcal{D},\#,E)$, with $S{=}\prod S_i$ (Cartesian product), $s_0{=}(s_{0,1},\dots,s_{0,k})$, $\mathcal{P}{=}\bigcup\mathcal{P}_i\backslash\bigcup\mathcal{Q}_i$, $\mathcal{X}{=}\bigcup\mathcal{X}_i$, $I((s_1,\dots,s_k)){=}\bigwedge I_i(s_i)$, $\mathcal{D}{=}\bigcup\mathcal{D}_i$, $\#{:}S{\to}2^\mathcal{D}$, with $\#((s_1,\dots,s_k)){=}\bigcup\#(s_i)$, and E is defined in (2.18).

$$\frac{\begin{array}{c}(s_1,\mathbb{c}_1,dc_1,cc_1,\lambda_1,s_1'){\in}E_1,\dots,(s_k,\mathbb{c}_k,dc_k,cc_k,\lambda_k,s_k'){\in}E_k,\\ \mathbb{c}{=}(\mathbb{c}_1{\cup}\dots{\cup}\mathbb{c}_k)|_\mathcal{P}\text{ valid over }\mathcal{Q}\\ dc{=}(dc_1{\wedge}\dots{\wedge}dc_k)|_\mathcal{Q}\\ cc{=}cc_1{\wedge}\dots{\wedge}cc_k,\lambda{=}\lambda_1{\circ}\dots{\circ}\lambda_k\end{array}}{((s_1,\dots,s_k),\mathbb{c},dc,cc,\lambda,(s_1',\dots,s_k')){\in}E} \tag{2.18}$$

We call the $\mathfrak{N}_i$ the *underlying TNA* of $\mathcal{N}{\bowtie_\mathcal{Q}}$.

A transition in the composed TNA results from composing k transitions (called *underlying transitions*) from the underlying TNA. If all underlying transitions are delays, the resulting transition in $\mathcal{N}{\bowtie_\mathcal{Q}}$ is a delay as well. If at least one of the underlying transitions is a communication, the transition in $\mathcal{N}{\bowtie_\mathcal{Q}}$ is a communication as well. In the latter case, the associated TNA of underlying delays (i.e., TNA which contribute a delay to the composed transition) perform zero-delay steps. As for TCA (cf. Definition 2.3.9 and explanations thereafter), we have that the semantics of TNA (in particular the fourth and fifth row of (2.15)) ensures that data values stored in data variables used by locations correctly carry over to the next location.

Example 2.4.15 (TNA Composition). Consider two instances of the TNA from Example 2.4.4. The second instance is identical to the TNA in Figure 2.8, except that we add a prime to all names (locations, ports, data variables, clocks). We compose the two TNA over the merge set $\mathcal{Q}{=}\{w,r'\}$, the result is shown in Figure 2.9. As has been done in Figure 2.7, we abbreviate location names for readability. The resulting TNA models an expiring FIFO buffer with capacity 2, where each buffer cell has its own expiration timer.

The explanation of the behaviour of the TNA is essentially equivalent to the explanation in Example 2.3.10. A significant difference is the fact that it is not possible

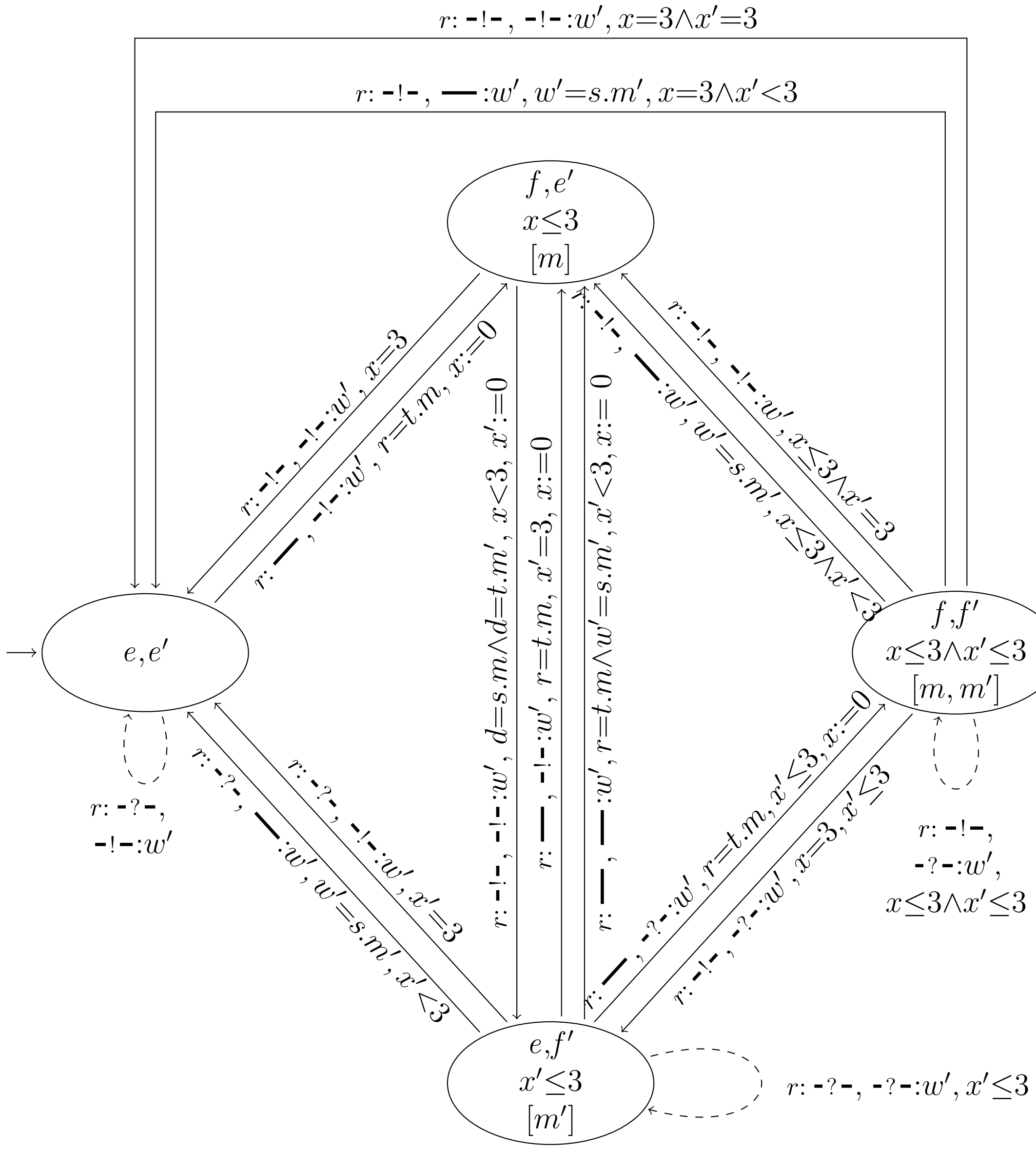

r: -!-, -!-:w′, x=3∧x′=3
r: -!-, —:w′, w′=s.m′, x=3∧x′<3
f,e′ x≤3 [m]
e,e′
f,f′ x≤3∧x′≤3 [m,m′]
e,f′ x′≤3 [m′]
r: -!-, -!-:w′, x=3
r: —, -!-:w′, r=t.m, x:=0
r: -!-, -!-:w′, d=s.m∧d=t.m′, x<3, x′:=0
r: —, -!-:w′, r=t.m, x′=3, x:=0
r: —, —:w′, r=t.m∧w′=s.m′, x′<3, x:= 0
r: -!-, —:w′, w′=s.m′, x≤3∧x′<3
r: -!-, -!-:w′, x≤3∧x′=3
r: -?-, -!-:w′
r: -?-, —:w′, w′=s.m′, x′<3
r: -?-, -!-:w′, x′=3
r: —, -?-:w′, r=t.m, x′≤3, x:=0
r: -!-, -?-:w′, x=3, x′≤3
r: -!-, -?-:w′, x≤3∧x′≤3
r: -?-, -?-:w′, x′≤3

Figure 2.9: TNA Composition: Example

to delay in location (f,e') (while this was possible in the TCA in Example 2.3.10). After entering this location, the only possible transitions are the communications to locations (e,e') and (e,f'). The first transition is taken if $x{=}3$, i.e., the timer of the first buffer cell has expired. The second transition corresponds to an "internal" transition, where the data item is transmitted from the first to the second buffer cell. This transition shows that the composed TNA is indeed a FIFO buffer: data items may not remain in the first buffer cell if the second buffer cell is empty, but are pushed towards the "end" as far as possible.

Also note that on the transition from (e,f') to (f,f'), the value of m' is unconstrained. Since the underlying transition in the second (primed) TNA is a delay, the semantics ensures that the value carries over to (f,f') unchanged.

Proposition 2.4.16 (TNA Composition). The composition of TNA over *disjoint* merge sets is commutative and—after applying the flip rule to remove redundant colourings—associative, up to isomorphy of location names.

Proof.

1. Commutativity follows from the commutativity of $\cup$ on port sets, clock sets and data variable sets, the commutativity of $\wedge$ on clock and data constraints, the commutativity of $\cup$ on colourings over disjoint ports sets, and the commutativity of $\circ$ on update maps over disjoint clock sets.

2. Associativity follows from the associativity of $\cup$ on port sets, clock sets and data variable sets, the associativity of $\wedge$ on clock and data constraints, the associativity of $\cup$ on colourings over disjoint port sets, the associativity of $\circ$ on update maps over disjoint clock sets, and the fact that for disjoint $\mathcal{P}_i$, i=1, 2, 3,, and disjoint $\mathcal{Q}_j$, j=1, 2, 3, with $\mathcal{Q}_j \subseteq \bigcup \mathcal{P}_i$, we have

$$(((\mathcal{P}_1 \cup \mathcal{P}_2) \setminus (\mathcal{Q}_1 \cup \mathcal{Q}_2)) \cup \mathcal{P}_3) \setminus \mathcal{Q}_3 = (((\mathcal{P}_2 \cup \mathcal{P}_3) \setminus (\mathcal{Q}_2 \cup \mathcal{Q}_3)) \cup \mathcal{P}_1) \setminus \mathcal{Q}_1$$

$\square$

The semantics of a composed TNA $\mathcal{N}\bowtie_{\mathcal{Q}}$ (Definition 2.4.14) is defined in the same way as for simple TNA (Definition 2.4.7), i.e., by executions of the associated LTS (Definition 2.4.6).

In (2.19), we show a run of the composed TNA from Figure 2.9 of length 7. Again, we omit data assignments on transitions which evaluate to $\bot$, and we abuse the single symbol $\bot$ to denote the empty data assignment. Further, we omit port names, the colourings correspond to port r on top and port w' below.

$$\begin{aligned}
&\langle (e,e'),\bot,\substack{x=0\\x'=0}\rangle \xrightarrow{\substack{\mathtt{-?-}\\\mathtt{-!-}},4} \langle (e,e'),\bot,\substack{x=4\\x'=4}\rangle \xrightarrow{\substack{\text{——}\\\mathtt{-!-}},\substack{r=3\\t.m=3}} \langle (f,e'),m=3,\substack{x=0\\x'=4}\rangle\\
&\xrightarrow{\substack{\mathtt{-!-}\\\mathtt{-!-}},\substack{t.m=3\\s.m'=3\\d=3}} \langle (e,f'),m'=3,\substack{x=0\\x'=0}\rangle \xrightarrow{\substack{\mathtt{-?-}\\\mathtt{-?-}},0.5} \langle (e,f'),m'=3,\substack{x=0.5\\x'=0.5}\rangle\\
&\xrightarrow{\substack{\text{——}\\\mathtt{-?-}},\substack{r=5\\t.m=5\\s.m'=3\\t.m'=3}} \langle (f,f'),\substack{m=5\\m'=3},\substack{x=0\\x'=0.5}\rangle \xrightarrow{\substack{\mathtt{-!-}\\\mathtt{-?-}},2.5} \langle (f,f'),\substack{m=5\\m'=3},\substack{x=2.5\\x'=3}\rangle\\
&\xrightarrow{\substack{\mathtt{-!-}\\\mathtt{-!-}},\substack{s.m=5\\t.m=5}} \langle (f,e'),m=5,\substack{x=2.5\\x'=3}\rangle \xrightarrow{\substack{\mathtt{-!-}\\\mathtt{-!-}},\substack{t.m=5\\s.m'=5\\d=5}} \langle (e,f),m'=5,\substack{x=2.5\\x'=0}\rangle\\
&\xrightarrow{\substack{\mathtt{-?-}\\\mathtt{-?-}},3} \langle (e,f),m'=5,\substack{x=5.5\\x'=3}\rangle \xrightarrow{\substack{\mathtt{-?-}\\\mathtt{-!-}},\bot} \langle (e,e'),\bot,\substack{x=5.5\\x'=3}\rangle \qquad (2.19)
\end{aligned}$$

The run describes almost the same behaviour as the TCA run presented in (2.14). In particular, the time values on transitions, that means the lengths of delays, are identical wherever possible. Yet, as mentioned in Example 2.4.15, the significant difference is that it is not possible to delay in location (f,e'): whenever the TNA enters that location, it (in this example) immediately transfers the data item from the first to the second buffer cell, thereby moving to location (e,f').

Remark 2.4.17 (Size of the Composition). The size of a TNA, i.e., the number of locations and transitions, can be exponential (in the number of locations and transitions of the underlying TNA) in the worst case. For model checking/verification, we present a linear technique to avoid the explicit construction of the composed TNA in Section 3.1.4.

2.4.4 Discussion

In this Section, we have described a powerful framework for *compositional construction of* and *coordination in* real-time dataflow networks, which takes into account environmental constraints from outside the network. The approach is suitable to model both the "algorithmic" behaviour of components and connectors (for example the internal implementation of the coordination pattern), and the inter-component coordination behaviour. In this way, whole networks can easily be described with our formalism.

The development of TNA in [Kem10] was—amongst others—motivated by the fact that in TCA, it is not possible to enforce a transition to be taken as soon as dataflow through some port is possible. For example, consider again the FIFO buffer in Figure 2.5: the TCA is not required to take the transition from location *empty* to *full* as soon as data is available through p, instead, it can delay for an arbitrary amount of time. In contrast, in the corresponding TNA in Figure 2.8, when data is available through r, the TNA can no longer delay in location *empty*. Yet, we could still permit to delay for an arbitrary amount of time in Figure 2.8, by changing the colouring $r{:}-?-$ to $r{:}-!-$ on the delay in location *empty*.

In this thesis, we have further extended the basic definition of TNA from [Kem10] in two ways, by introducing data guards and data variables (together with a domain $\mathcal{D}ata$ of admissible data values). Data guards on transitions allow to model coordination patterns which depend on concrete data values, rather than only on presence and absence of dataflow (which was the case in [Kem10]). The benefits of introducing data variables are twofold: as for TCA, using data variables in locations ("location memory") allows the coordination pattern to reason about data values which were exchanged *prior* to the current step. Secondly, in the composition of TNA, using data variables and data guards allows us to preserve the information about the data values which are exchanged through internal ports (cf. Definitions 2.4.13 and 2.4.14, and explanations before that).

With TNA, we have defined a powerful yet simple framework for defining and describing the behaviour of connectors and whole networks. Our liberal notion allows to encode many common (coordination) models, like for example TA and TCA, in our framework.[11] On the other hand, the state-based approach makes our framework easy to understand (and thus, use), and facilitates the introduction of new, user-defined coordination patterns.

[11]The basic idea for TCA is to restrict the use of data variables to locations, and to use the no-flow colour $-!-$ only. For TA, the basic idea is to not allow data variables or data guards at all, and permit only one port per transition. We skip the technical details.

2.5 Conclusion

In this Chapter, we have presented general concepts for handling time and data in real-time systems, and we have presented three formal models for specification of real-time systems and real-time coordination patterns.

The formal model of Timed Automata, presented in Section 2.2, has first been introduced in [AD94] (and later in [Alu99]), and—as has been outlined in Section 2.2.4—has been studied, modified and extended intensively since that time. The Definitions and results in Section 2.2 have not been developed by us, but are entirely based on previous work on this field, which we have made clear by providing pointers to corresponding fundamental literature all throughout the Section. The nonetheless in-depth character of Section 2.2 is owed to the facts that there exist so many variants of TA (with sometimes only minor differences) that it is impossible to use the notion of *the* Timed Automaton without further explanations, and that the time-related notions (guards, invariants, update maps) directly carry over to TCA and TNA. Section 2.2 thus serves the purpose to make clear which notions and concepts of TA we use in this thesis, and to establish the concepts for handling of real-time. Moreover, it serves as the formal basis for the formula representation and SAT-based verification of TA introduced in the next Chapter.

We have presented the second formal model, Timed Constraint Automata, in Section 2.3. The first formal definition of syntax and semantics of TCA can be found in [ABdBR04] (and its extended version [ABdBR07]), though the authors do not handle memory cells. In [PSHA09], memory cells are added to constraint automata, resulting in the formal model of CASM (Constraint Automata with Memory Cells), but as the name suggests, CASM do not include time. While the underlying idea of handling data and memory cells in CASM is similar to the approach presented here, the semantics of CASM is not defined formally, which leaves a number of open questions. For example, it is unclear whether a memory cell used by the target location of a CASM transition has to be initialised in the data guard of ingoing transitions, or what happens in case it is not initialised (is the value undefined, empty, random). From [PSHA09], we have adopted the notion of prefixing memory cells with “s.” and “t.” on transitions (cf. Remark 2.3.2), but to the best of our knowledge, this is the first work on combining TCA with memory cells, and defining a formal semantics.

Finally, in Section 2.4, we have presented our third and most powerful (with respect to expressiveness of the three models) formal model: Timed Network Automata. As explained in Section 2.4.4, we have originally developed TNA as an enhancement of TCA, to be able to specify constraints under which it is admissible to delay (further). In this work, we have improved the TNA model from [Kem10], by adding data guards and data values, with the advantages discussed in Section 2.4.4.

Chapter 3

SAT-based Verification

Model checking [CGP99, BK08] is the problem of automatically verifying (proving or disproving) whether a system conforms to its specification, given as a set of properties/constraints. Model checking usually consists of enumerating all reachable configurations of the system, and then checking whether the properties hold in these. Yet, the infinite state-space of real-time systems leads to severe limitations in scalability, even in very well-established model checkers like Uppaal [upp]. Especially for the verification of safety properties of real-time systems [CBRZ01, BCC^{+}03], Bounded Model Checking (BMC, [BCCZ99]) has turned out to be amongst the most promising approaches. Safety properties declare what should not happen—or equivalently, what should always happen—and are typically expressed as reachability properties. Safety properties can be disproved with a finite counterexample, i.e., a finite run, where the last configuration contains a contradiction to the property. The principle of BMC for safety properties is to examine prefix fragments of the transition system, and successively increase the exploration bound until it reaches (a computable indicator of) the diameter of the system—in which case the system has been proven safe—or an unsafe run has been discovered [ACKS02].

In this chapter, for each of the system models defined in the previous chapter (i.e., TA, TCA, TNA), we present an encoding in propositional logic, plus linear arithmetic on the rational numbers, which is tailored towards BMC. A *satisfiability check* on the resulting formula (SAT solving) using SMT solvers[1] like FOCI [FOC] or MathSAT [mat], is then used to find possible runs of the system.

The main idea is that for any system model $\mathfrak{S}$, with $\mathfrak{S}\in\{\mathfrak{A},\mathfrak{T},\mathfrak{N}\}$ (cf. Definitions 2.2.1, 2.3.1 and 2.4.2), we define a formula $\varphi(\mathfrak{S})$. This formula encodes the transition characteristics of $\mathfrak{S}$, that means, the possibilities to evolve to the next step $\mathtt{t+1}$, based on the configuration in the current step $\mathtt{t}$. For BMC, we unfold

[1]Satisfiability Modulo Theory (SMT) problems combine propositional satisfiability with an underlying theory, for example the theory of linear arithmetic over the real numbers. Atoms in an SMT problem can consist of propositional variables or theory atoms, and are combined with the Boolean connectives.

the formula $\varphi(\mathfrak{S})$ k times, i.e., we instantiate the "abstract" indices $\mathtt{t}$ and $\mathtt{t{+}1}$ for all steps 1 up to bound k, which yields a variant $\varphi(\mathfrak{S})_k$. Intuitively, a satisfying interpretation of the formula $\varphi(\mathfrak{S})_k$ corresponds to a run of the associated LTS $\mathfrak{S}_{\mathfrak{S}}$, i.e., to one possible behaviour for the first k steps. Consequently, the set of all possible valuations of $\varphi(\mathfrak{S})_k$ corresponds to the complete possible behaviour of $\mathfrak{S}$ for the first k steps.

The rest of this Chapter is organised as follows: in Section 3.1, we define the formula representation. As in the previous Chapter, we start with a general part (Section 3.1.1), and then discuss in detail the formula representation of the different system models and their products (Sections 3.1.2 to 3.1.4). In Section 3.2, we present the unfolding for BMC, and discuss some issues related to BMC. We discuss other possibilities for encoding, and motivate our design decisions in Section 3.3, and conclude the Chapter in Section 3.4.

3.1 Formula Representation

The possible behaviour (i.e., which transition can be taken) of a real-time system $\mathfrak{S}$ depends on the current system configuration (location, clock valuation, data variables, events, ports, memory cells) and changes over time. This time-dependent behaviour needs to be reflected by the formula $\varphi(\mathfrak{S})_k$. Therefore, we "parametrise" the variables representing the constituents of $\mathfrak{S}$ by the step $\mathtt{t}$ they are evaluated in. This is called *localisation*: the localisation ψ_t of a formula ψ is obtained by adding index t to all variable symbols occurring in ψ. Thus, if ψ is of vocabulary x, s, p, then ψ_t is of vocabulary $\mathtt{x_t}, \mathtt{s_t}, \mathtt{p_t}$.

In the next section, we present the representation for constituents common to more than one real-time system: clocks, locations, events, ports, and data variables/memory cells. In Sections 3.1.2, 3.1.3 and 3.1.4, we introduce the specific transition characteristics of TA, TCA and TNA, respectively.

3.1.1 Preliminaries

We now show how to represent constituents common to more than one real-time system.

3.1.1.1 Clocks, Clock Constraints

Let $\mathcal{X}$ be the set of clocks in $\mathfrak{S}$. For the representation of clocks, we first introduce a fresh clock z, called *absolute time reference*, which is not used in any clock constraint and which is never updated, thus, the value of z increases constantly. This clock is used to measure the absolute amount of time that has passed since the beginning of computation: for any step $\mathtt{t}$, the rational variable $\mathtt{z_t}$ (called *representation of* z) represents the value of z in step $\mathtt{t}$, i.e., the absolute amount of time which has passed from the beginning of computation up to step $\mathtt{t}$. For every clock $x{\in}\mathcal{X}$, the rational variable $\mathtt{x_t}$ (*clock reference (of clock* x*)*) is used to compute the value of clock x in step t, which is given by the difference $\mathtt{z_t}{-}\mathtt{x_t}$. Thus, for clock constraints $cc{=}x{\sim}n$ and

$cc'=x-y\sim n$ (cf. Definition 2.1.2), the formulas $\mathtt{cc_t}=\mathtt{z_t}-\mathtt{x_t}\sim n$ and $\mathtt{cc'_t}=\mathtt{y_t}-\mathtt{x_t}\sim n$,[2] (called *representation of cc and cc'*, respectively) evaluate to `true` iff cc and cc' hold in step `t`. The representation of other clock constraints is straightforward, by using conjunctions of the above constraints.

The underlying idea of clock references is that the variable $\mathtt{x_t}$ will keep its value as long as clock x is not updated in $\mathfrak{S}$. When clock x is updated, there are two possibilities (cf. Definition 2.1.5): either x is updated to a natural number $n\in\mathbb{N}$, or x is updated to the value of another clock x'. In the former case, the value of $\mathtt{x_t}$ is set to $\mathtt{z_t}-n$, in the latter case, it is set to the value of $\mathtt{x'_t}$. In both cases, the difference $\mathtt{z_t}-\mathtt{x_t}$ yields the correct value of x. This temporal difference representation significantly improves the SAT solving performance, due to the decreased number of arithmetic operations, see Section 3.3 for a more detailed discussion.

We illustrate the idea describe above in Figure 3.1. Above the line representing the value of z, we denote the updates, as found on transitions of $\mathfrak{S}$. Below the time line (x-axis), we denote the formulas which are used to set $\mathtt{x_t}$ to the correct value.

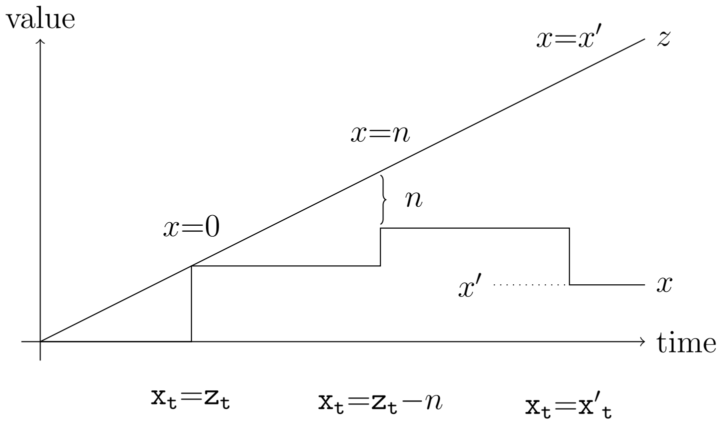

value
$x=x'$
z
$x=n$
n
$x=0$
x'
x
time
$\mathtt{x_t}=\mathtt{z_t}$
$\mathtt{x_t}=\mathtt{z_t}-n$
$\mathtt{x_t}=\mathtt{x'_t}$

Figure 3.1: Representation of Clock Values, Concept

By definition of the allowed updates for a clock x, the value of $\mathtt{x_t}$ is always smaller than the value of $\mathtt{z_t}$. The value of $\mathtt{x_t}$ can become negative in case of an update $\lambda(x)=n$, with $n>\mathtt{z_t}$ (or, obviously, in case x is updated to the value of another clock x' where $\mathtt{x'_t}$ is already negative).

3.1.1.2 Locations

Let S be the set of locations of $\mathfrak{S}$. We use a linear Boolean encoding for locations: for every location $s\in S$, the Boolean variable $\mathtt{s_t}$ (called *representation of s*) represents whether $\mathfrak{S}$ is in location s in step `t`. Please refer to Section 3.3 for a discussion of other possible encodings.

3.1.1.3 Data Values

To represent the (possibly infinite, countable) set of data values $\mathcal{D}ata$ for TCA and TNA, we use an injective mapping $\Delta:\mathcal{D}ata\rightarrow\mathbb{Z}$, which maps each element $\mathbb{d}_i\in\mathcal{D}ata$,

[2]Actually, the formula is $((\mathtt{z_t}-\mathtt{x_t})-(\mathtt{z_t}-\mathtt{y_t}))\sim n$, but this simplifies to $\mathtt{y_t}-\mathtt{x_t}\sim n$.

$\mathbb{d}_i \neq \perp$, to an integer number $\mathtt{n^i}$ (called *representation of* $\mathbb{d}_i$). If a total order $\leqslant$ exists on $\mathcal{D}ata$ (cf. Definition 2.1.7), we require Δ to preserve this total order, i.e., for all $\mathbb{d}_i, \mathbb{d}_j \in \mathcal{D}ata$ such that $\mathbb{d}_i \neq \mathbb{d}_j$, if $\mathbb{d}_i \leqslant \mathbb{d}_j$, then $\Delta(\mathbb{d}_i) = \mathtt{n^i} < \mathtt{n^j} = \Delta(\mathbb{d}_j)$. For the *representation of* $\perp$, we introduce an integer constant, denoted by $\mathtt{n}^\perp$, without assigning a specific value to it; see Remark 3.1.5 for further explanations. For explanatory purposes, we treat $\mathtt{n}^\perp$ similarly to $\mathtt{n^i}$, i.e., as if it was an element of $\mathbb{Z}$.

3.1.1.4 Events, Ports, Data Variables

To represent the set of events Σ of a TA $\mathfrak{A}$, we use a linear Boolean encoding: for every event $\mathfrak{a} \in \Sigma$, the Boolean variable $\alpha_\mathtt{t}$ (called *representation of* $\mathfrak{a}$) represents whether $\mathfrak{A}$ executes a transition in step $\mathtt{t}$ that is labelled with $\mathfrak{a}$. Please refer to Section 3.3 for a discussion on other possible encodings of events.

The basic idea for ports (in TCA or TNA) is the same as for events: for every port $p \in \mathcal{P}$, the Boolean variable $\mathtt{p_t}$ (called *port activity variable of* p) represents whether port p is active in step $\mathtt{t}$. In addition, to encode which data value is transmitted over an active port, for every port $p \in \mathcal{P}$, we introduce an integer variable $\mathtt{Dp_t}$ (called *port data variable*), which represents the data value pending on p in step $\mathtt{t}$. If p is inactive in step $\mathtt{t}$, $\mathtt{Dp_t}$ evaluates $\mathtt{n}^\perp$; see Remark 3.1.5 for further explanations.

For data variables, we use a similar set of variables, though with a slightly different meaning: for every data variable $d \in \mathcal{D}$, we introduce a Boolean variable $\mathtt{d_t}$, and an integer variable $\mathtt{Dd_t}$. The variable $\mathtt{d_t}$ (called *data fullness variable*) is used to indicate whether d is full in step $\mathtt{t}$. As for ports, the variable $\mathtt{Dd_t}$ (called *data content variable*) represents which data value (according to mapping Δ) is contained in d in step $\mathtt{t}$, in case d is not empty, $\mathtt{Dd_t}$ evaluates to $\mathtt{n}^\perp$ if d is empty in step $\mathtt{t}$.

Though the encoding using two variables per port/data variable might seem unnecessary, we need this for efficient abstraction. Please refer to Section 4.1 for more details.

3.1.1.5 Data Constraints

For a data constraint $dc = (\mathbf{D} \simeq \mathbf{D}')$, $\simeq \in \{=, \leqslant\}$ (cf. Definition 2.1.7), the formula $\mathtt{dc_t}$ (the *representation of* dc) evaluates to $\mathtt{true}$ iff dc holds in step $\mathtt{t}$. The constraint $\mathtt{dc_t}$ is defined by replacing ports and data values occurring in dc with their corresponding representations; i.e., replace $p \in \mathcal{P}|_{dc}$ by $\mathtt{Dp_t}$, and $\mathbb{d}_i \in \mathcal{D}ata|_{dc}$ by $\Delta(\mathbb{d}_i) = \mathtt{n}^i$. For data variables $d \in \mathcal{D}|_{dc}$, we need to take into account whether they are used by the source or the target location of the transition to which dc belongs (i.e., whether they occur as $s.d$ or as $t.d$ in dc, cf. Remarks 2.3.2 and 2.4.3), or only in dc itself. In the latter two cases, we replace $d \in \mathcal{D}$ by $\mathtt{Dd_t}$, in the first case, we replace d by $\mathtt{Dd_{t\text{-}1}}$. This corresponds to the fact that d can be used by the source location only *before* the execution of the transition, i.e., at step $\mathtt{t-1}$. The representation of other data constraints is straightforward, using conjunctions and negations of the aforementioned.

3.1.2 Timed Automata

The representation of the transition relation of TA has to model both action and delay transitions, cf. Section 2.2. It constrains the possible valuations of variables representing the automaton configuration at subsequent step `t+1` depending on those at `t`. Recalling the representation of clocks, locations and events from the previous section, the representation of TA is defined as follows.

Definition 3.1.1 (TA Representation). Let $\mathfrak{A}$ be a TA, with initial location $\bar{s}$,[3] let $e{=}(s, \mathfrak{a}, cc, \lambda, s')$ be a transition in $\mathfrak{A}$. The *formula representation of the transition relation of* $\mathfrak{A}$, denoted $\varphi(\mathfrak{A})$, is defined in (3.7) of Table 3.2.

$$\varphi^{init}(\mathfrak{A}) = \bar{\mathtt{s}}_0 \wedge \bigwedge_{s\in S, s\neq\bar{s}} \neg \mathtt{s}_0 \wedge \mathtt{I}(\bar{\mathtt{s}})_0 \wedge \bigwedge_{\mathfrak{a}\in\Sigma} \neg \alpha_0 \wedge (\mathtt{z}_0{=}0) \wedge \bigwedge_{x\in\mathcal{X}} (\mathtt{x}_0{=}0) \tag{3.1}$$

$$\varphi^{action}(e) = \mathtt{s}_\mathtt{t} \wedge \alpha_{\mathtt{t+1}} \wedge \mathtt{cc}_\mathtt{t} \wedge (\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}}) \wedge \bigwedge_{\lambda(x)=id} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x}_\mathtt{t}) \wedge \tag{3.2}$$

$$\bigwedge_{\lambda(x)=x'} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x'}_{\mathtt{t+1}}) \wedge \bigwedge_{\lambda(x)=n} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{z}_{\mathtt{t+1}}{-}n) \wedge \mathtt{s'}_{\mathtt{t+1}} \wedge \mathtt{I}(\mathtt{s'})_{\mathtt{t+1}}$$

$$\varphi^{delay}(s) = \mathtt{s}_\mathtt{t} \wedge \bigwedge_{\mathfrak{a}\in\Sigma} \neg \alpha_{\mathtt{t+1}} \wedge (\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_{\mathtt{t+1}}) \wedge \bigwedge_{x\in\mathcal{X}} (\mathtt{x}_\mathtt{t}{=}\mathtt{x}_{\mathtt{t+1}}) \wedge \mathtt{s}_{\mathtt{t+1}} \wedge \mathtt{I}(\mathtt{s})_{\mathtt{t+1}} \tag{3.3}$$

$$\varphi^{trans}(\mathfrak{A}) = \bigvee_{e\in E} \varphi^{action}(e) \vee \bigvee_{s\in S} \varphi^{delay}(s) \tag{3.4}$$

$$\varphi^{location}(\mathfrak{A}) = \bigvee_{s\in S} (\mathtt{s}_{\mathtt{t+1}} \wedge \bigwedge_{s'\in S, s'\neq s} \neg \mathtt{s}'_{\mathtt{t+1}}) \tag{3.5}$$

$$\varphi^{mutex}(\mathfrak{A}) = \bigvee_{\mathfrak{a}\in\Sigma} (\alpha_{\mathtt{t+1}} \wedge \bigwedge_{\mathfrak{a}'\in\Sigma, \mathfrak{a}'\neq\mathfrak{a}} \neg \alpha'_{\mathtt{t+1}}) \vee \bigwedge_{\mathfrak{a}\in\Sigma} (\neg \alpha_{\mathtt{t+1}}) \tag{3.6}$$

$$\varphi(\mathfrak{A}) = \varphi^{init}(\mathfrak{A}) \wedge \varphi^{trans}(\mathfrak{A}) \wedge \varphi^{location}(\mathfrak{A}) \wedge \varphi^{mutex}(\mathfrak{A}) \tag{3.7}$$

Table 3.2: Transition Relation Representation of TA

The idea of these formulas is as follows: the TA starts in its initial location (3.1), the invariant of which has to hold (by $\mathtt{I}(\mathtt{s})_\mathtt{t}$, we denote the localisation of the invariant $I(s)$ of some location s, cf. Section 3.1), no action is enabled, and all clocks start with value 0. Before executing an action transition $e{=}(s, \mathfrak{a}, cc, \lambda, s'){\in}E$ of step `t+1` in (3.2), the automaton is in location s (at step `t`), and the transition guard $\mathtt{cc}_\mathtt{t}$ is satisfied. On occurrence of event $\alpha_{\mathtt{t+1}}$, the transition fires. The value of the absolute time reference does not change (action transitions are instantaneous), other clocks are updated according to their value under update map λ (they either keep their value, are set to $\mathtt{z}_\mathtt{t}{-}n$ if $\lambda(x){=}n$, cf. Section 3.1.1.1, or get the value of another clock reference $\mathtt{x'}_\mathtt{t}$). After the execution (at step `t+1`), the automaton is in location s', the invariant of which has to hold. For a delay transition (3.3), the automaton remains in location s, the value of the absolute time reference increases, all clock references keep their value (there is no update, cf. (2.2)), no event $\mathfrak{a}{\in}\Sigma$ must occur, and the invariant has to hold after the time delay. Due to convexity, the invariant of the target location in both action and delay transitions only needs to be checked at the end of the transition/delay (that means at step `t+1`), as it inductively holds at

[3]To avoid confusion with localisation indices, we denote the initial location as $\bar{s}$ rather than s_0, so its representation is $\bar{\mathtt{s}}_0$ rather than the odd-locking $(\mathtt{s}_0)_0$).

the beginning (3.1). The disjunction of these formulas expresses (nondeterministic) transition choice (3.4). In any step, the current location and event are unique (mutual exclusion of location (3.5) and event variables (3.6)), to prevent $\varphi(\mathfrak{A})$ from following multiple transitions simultaneously.

Example 3.1.2 (TA Representation). Consider again the intelligent light switch presented in Figure 2.1 (Example 2.2.2). Let $\mathfrak{A}$ be the name of the automaton, let e_1, e_2, e_3, e_4 refer to the transitions from *off* to *light*, *light* to *off*, *light* to *bright*, and *bright* to *off*, respectively. The representation of $\mathfrak{A}$ is shown in Table 3.3. We omit constraints equal to `true`, like for example clock guards or empty conjunctions.

$$
\begin{aligned}
\varphi^{init}(\mathfrak{A}) &= \mathtt{off}_0\wedge\neg\mathtt{light}_0\wedge\neg\mathtt{bright}_0\wedge\neg\mathtt{press}_0\wedge\neg\tau_0\wedge(\mathtt{z}_0{=}0)\wedge(\mathtt{x}_0{=}0)\\
\varphi^{action}(e_1) &= \mathtt{off}_\mathtt{t}\wedge\mathtt{press}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t+1}{=}\mathtt{z}_\mathtt{t+1})\wedge\mathtt{light}_\mathtt{t+1}\\
\varphi^{action}(e_2) &= \mathtt{light}_\mathtt{t}\wedge\mathtt{press}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{z}_\mathtt{t}{-}\mathtt{x}_\mathtt{t}{>}3)\wedge(\mathtt{x}_\mathtt{t+1}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{off}_\mathtt{t+1}\\
\varphi^{action}(e_3) &= \mathtt{light}_\mathtt{t}\wedge\mathtt{press}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{z}_\mathtt{t}{-}\mathtt{x}_\mathtt{t}{\le}3)\wedge(\mathtt{x}_\mathtt{t+1}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{bright}_\mathtt{t+1}\\
\varphi^{action}(e_4) &= \mathtt{bright}_\mathtt{t}\wedge\mathtt{press}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t+1}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{off}_\mathtt{t+1}\\
\varphi^{delay}(\mathit{off}) &= \mathtt{off}_\mathtt{t}\wedge\neg\mathtt{press}_\mathtt{t+1}\wedge\neg\tau_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{\le}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_\mathtt{t+1})\wedge\mathtt{off}_\mathtt{t+1}\\
\varphi^{delay}(\mathit{light}) &= \mathtt{light}_\mathtt{t}\wedge\neg\mathtt{press}_\mathtt{t+1}\wedge\neg\tau_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{\le}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_\mathtt{t+1})\wedge\mathtt{light}_\mathtt{t+1}\\
\varphi^{delay}(\mathit{bright}) &= \mathtt{bright}_\mathtt{t}\wedge\neg\mathtt{press}_\mathtt{t+1}\wedge\neg\tau_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{\le}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_\mathtt{t+1})\wedge\mathtt{bright}_\mathtt{t+1}\\
\varphi^{trans}(\mathfrak{A}) &= \varphi^{action}(e_1)\vee\varphi^{action}(e_2)\vee\varphi^{action}(e_3)\vee\varphi^{action}(e_4)\vee\\
&\quad\ \varphi^{delay}(\mathit{off})\vee\varphi^{delay}(\mathit{bright})\vee\varphi^{delay}(\mathit{light})\\
\varphi^{location}(\mathfrak{A}) &= (\mathtt{off}_\mathtt{t+1}\wedge\neg\mathtt{light}_\mathtt{t+1}\wedge\neg\mathtt{bright}_\mathtt{t+1})\vee\\
&\quad\ (\mathtt{light}_\mathtt{t+1}\wedge\neg\mathtt{off}_\mathtt{t+1}\wedge\neg\mathtt{bright}_\mathtt{t+1})\vee\\
&\quad\ (\mathtt{bright}_\mathtt{t+1}\wedge\neg\mathtt{off}_\mathtt{t+1}\wedge\neg\mathtt{light}_\mathtt{t+1})\\
\varphi^{mutex}(\mathfrak{A}) &= (\mathtt{press}_\mathtt{t+1}\wedge\neg\tau_\mathtt{t+1})\vee(\tau_\mathtt{t+1}\wedge\neg\mathtt{press}_\mathtt{t+1})\vee(\neg\mathtt{press}_\mathtt{t+1}\wedge\neg\tau_\mathtt{t+1})\\
\varphi(\mathfrak{A}) &= \varphi^{init}(\mathfrak{A})\wedge\varphi^{trans}\wedge\varphi^{location}(\mathfrak{A})\wedge\varphi^{mutex}(\mathfrak{A})
\end{aligned}
$$

Table 3.3: Transition Relation Representation of TA: Example

The product of TA, as defined in Definition 2.2.8, in the worst case is exponential in the size of the underlying TA. We now present a representation of the product which is *linear* in the size of the underlying TA. The basic idea is to retain the representations of the individual automata, and define the product as their juxtaposition.

Definition 3.1.3 (TA Product Representation). Let $\mathfrak{A}_1$ and $\mathfrak{A}_2$ be TA, with $\mathcal{X}_1{\cap}\mathcal{X}_2{=}\emptyset$ and $S_1{\cap}S_2{=}\emptyset$, let $\varphi(\mathfrak{A}_1)$ and $\varphi(\mathfrak{A}_2)$ be the respective representations, as defined in Definition 3.1.1, with (3.3) replaced by (3.3'). The *formula representation* $\varphi(\mathfrak{A}_1{\bowtie}\mathfrak{A}_2)$ *of the product* $\mathfrak{A}_1{\bowtie}\mathfrak{A}_2$ is defined in (3.9).

$$\varphi^{delay}(s) = \mathtt{s_t} \land \bigwedge_{\mathfrak{a}\in\Sigma_v} \neg\alpha_{\mathtt{t+1}} \land (\mathtt{z_t} \le \mathtt{z_{t+1}}) \land \bigwedge_{x\in\mathcal{X}} (\mathtt{x_t} = \mathtt{x_{t+1}}) \land \mathtt{s_{t+1}} \land \mathtt{I(s)}_{\mathtt{t+1}} \tag{3.3'}$$

$$\varphi^{mutex}(\mathfrak{A}_1 \bowtie \mathfrak{A}_2) = \bigvee_{\mathfrak{a}\in\Sigma_1\setminus\Sigma_2} (\alpha_{\mathtt{t+1}} \land \bigwedge_{\mathfrak{a}'\in\Sigma_2\setminus\Sigma_1} \neg\alpha'_{\mathtt{t+1}}) \lor \bigvee_{\mathfrak{a}\in\Sigma_2\setminus\Sigma_1} (\alpha_{\mathtt{t+1}} \land \bigwedge_{\mathfrak{a}'\in\Sigma_1\setminus\Sigma_2} \neg\alpha'_{\mathtt{t+1}}) \tag{3.8}$$

$$\varphi(\mathfrak{A}_1 \bowtie \mathfrak{A}_2) = \varphi(\mathfrak{A}_1) \land \varphi(\mathfrak{A}_2) \land \varphi^{mutex}(\mathfrak{A}_1 \bowtie \mathfrak{A}_2) \tag{3.9}$$

The product representation (3.9) faithfully models the intended behaviour of the product of TA, as defined in Definition 2.2.8: to ensure that the event occurring in step t is unique within the system, we add the constraint on mutual exclusion between events that are local to one of the TA (3.8), shared events are already dealt with in (3.6).

3.1.3 Timed Constraint Automata

The main ideas of the representation of the transition relation of TCA are similar to the representation of TA, as defined in the previous section. In particular, the modelling of locations and clocks is identical. Yet, the representation needs to take care of the special behaviour of TCA, namely, that every visible transition is preceded by a positive time delay, whereas invisible transitions may be instantaneous. Conceptually, on execution of a transition, the delay is represented by evolving from step t to step t+1, while the (instantaneous) location change takes place at t+1.

To correctly represent these delayed transitions (cf. (2.7)) in the associated LTS $\mathfrak{S}_{\mathfrak{T}}$ (cf. Definition 2.3.5), we need a second type of clock constraints. Clock constraints in (2.7) are evaluated under two different valuations: the invariant $I(s')$ of the target location s' is evaluated under $(\nu{+}t)[\lambda]$, that means *after* the time delay $(+t)$ and *after* the execution (λ) of the transition. In contrast, the invariant $I(s)$ of the source location s and the clock guard cc of the transition are evaluated under $(\nu{+}t)$, that means *after* the passage of time, but *before* the execution of the transition. To access the clock values at this particular point in time "in the middle" of the execution step, we define the *inter-step representation* $\mathtt{cc_{t\Delta}}$ *of a clock constraint* cc . For $cc = (x{\sim}n)$, the inter-step representation is given by $\mathtt{cc_{t\Delta}} = \mathtt{z_{t+1}} - \mathtt{x_t} {\sim} n$. Note that for a clock constraint $cc' = (x{-}y{\sim}n)$, the inter-step representation is equivalent to the representation of cc' as defined in Section 3.1.1.1, since this representation does not contain the absolute time reference z anymore, and delaying does not change the difference of x and y.

We are now ready to define the formula representation of TCA.

Definition 3.1.4 (TCA Representation). Let $\mathfrak{T}$ be a TCA, with initial location $\bar{s}$ (as before, we denote the initial location as $\bar{s}$ rather than s_0), let $e{=}(s, P, dc, cc, \lambda, s')$ and $e'{=}(s, \emptyset, dc, cc, \lambda, s')$ be a visible and invisible transition in $\mathfrak{T}$, respectively. The *formula representation of the transition relation of* $\mathfrak{T}$, denoted $\varphi(\mathfrak{T})$,[4] is defined in (3.16) in Table 3.4.

[4]Without confusion, we use the same formula identifiers for all real-time systems. For example, we use φ^{init} to denote the initial constraints for TA (3.1), TCA (3.10), and (in the next section) TNA (3.20).

$$\begin{aligned}\varphi^{init}(\mathfrak{T}) = {} & \bar{\mathtt{s}}_0 \wedge \bigwedge_{s\in S, s\neq\bar{s}} \neg\mathtt{s}_0 \wedge \mathtt{I}(\bar{\mathtt{s}})_0 \wedge \bigwedge_{p\in\mathcal{P}} (\neg\mathtt{p}_0 \wedge (\mathtt{Dp}_0 = \mathtt{n}^{\perp})) \wedge \\ & \bigwedge_{d\in\mathcal{D}} (\neg\mathtt{d}_0 \wedge (\mathtt{Dd}_0 = \mathtt{n}^{\perp})) \wedge (\mathtt{z}_0 = 0) \wedge \bigwedge_{x\in\mathcal{X}} (\mathtt{x}_0 = 0)\end{aligned} \tag{3.10}$$

$$\begin{aligned}\varphi^{visible}(e) = {} & \mathtt{s}_\mathtt{t} \wedge \mathtt{I}(\mathtt{s})_{\mathtt{t}\Delta} \wedge \bigwedge_{p\in P} \mathtt{p}_{\mathtt{t+1}} \wedge \bigwedge_{p\notin P} \neg\mathtt{p}_{\mathtt{t+1}} \wedge \bigwedge_{d\notin\#(s')} \neg\mathtt{d}_{\mathtt{t+1}} \wedge \mathtt{dc}_{\mathtt{t+1}} \wedge \\ & \mathtt{cc}_{\mathtt{t}\Delta} \wedge (\mathtt{z}_\mathtt{t} < \mathtt{z}_{\mathtt{t+1}}) \wedge \bigwedge_{\lambda(x)=id} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{x}_\mathtt{t}) \wedge \bigwedge_{\lambda(x)=x'} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{x'}_{\mathtt{t+1}}) \wedge \\ & \bigwedge_{\lambda(x)=n} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{z}_{\mathtt{t+1}} - n) \wedge \mathtt{s'}_{\mathtt{t+1}} \wedge \mathtt{I}(\mathtt{s'})_{\mathtt{t+1}}\end{aligned} \tag{3.11}$$

$$\begin{aligned}\varphi^{invisible}(e') = {} & \mathtt{s}_\mathtt{t} \wedge \mathtt{I}(\mathtt{s})_{\mathtt{t}\Delta} \wedge \bigwedge_{p\in\mathcal{P}} \neg\mathtt{p}_{\mathtt{t+1}} \wedge \bigwedge_{d\notin\#(s')} \neg\mathtt{d}_{\mathtt{t+1}} \wedge \mathtt{dc}_{\mathtt{t+1}} \wedge \mathtt{cc}_{\mathtt{t}\Delta} \wedge \\ & (\mathtt{z}_\mathtt{t} \leq \mathtt{z}_{\mathtt{t+1}}) \wedge \bigwedge_{\lambda(x)=id} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{x}_\mathtt{t}) \wedge \bigwedge_{\lambda(x)=x'} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{x'}_{\mathtt{t+1}}) \wedge \\ & \bigwedge_{\lambda(x)=n} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{z}_{\mathtt{t+1}} - n) \wedge \mathtt{s'}_{\mathtt{t+1}} \wedge \mathtt{I}(\mathtt{s'})_{\mathtt{t+1}}\end{aligned} \tag{3.12}$$

$$\varphi^{trans}(\mathfrak{T}) = \bigvee_{e\in E, P\neq\emptyset} \varphi^{visible}(e) \vee \bigvee_{e'\in E, P=\emptyset} \varphi^{invisible}(e') \tag{3.13}$$

$$\varphi^{location}(\mathfrak{T}) = \bigvee_{s\in S} (\mathtt{s}_{\mathtt{t+1}} \wedge \bigwedge_{s'\in S, s'\neq s} \neg\mathtt{s}'_{\mathtt{t+1}}) \tag{3.14}$$

$$\begin{aligned}\varphi^{mutex}(\mathfrak{T}) = {} & \bigwedge_{p\in\mathcal{P}} (\neg\mathtt{p}_{\mathtt{t+1}} \vee \neg(\mathtt{Dp}_{\mathtt{t+1}} = \mathtt{n}^{\perp})) \wedge (\mathtt{p}_{\mathtt{t+1}} \vee (\mathtt{Dp}_{\mathtt{t+1}} = \mathtt{n}^{\perp})) \wedge \\ & \bigwedge_{d\in\mathcal{D}} (\neg\mathtt{d}_{\mathtt{t+1}} \vee \neg(\mathtt{Dd}_{\mathtt{t+1}} = \mathtt{n}^{\perp})) \wedge (\mathtt{d}_{\mathtt{t+1}} \vee (\mathtt{Dd}_{\mathtt{t+1}} = \mathtt{n}^{\perp}))\end{aligned} \tag{3.15}$$

$$\varphi(\mathfrak{T}) = \varphi^{init}(\mathfrak{T}) \wedge \varphi^{trans}(\mathfrak{T}) \wedge \varphi^{location}(\mathfrak{T}) \wedge \varphi^{mutex}(\mathfrak{T}) \tag{3.16}$$

Table 3.4: Transition Relation Representation of TCA

The automaton starts in its initial location $\bar{s}$ (3.10) in step $\mathtt{0}$, the invariant of which has to be satisfied, data must not flow through any port, all memory cells are empty, and all clocks are set to zero. Before executing a visible transition (3.11) in step $\mathtt{t}$, $\mathfrak{T}$ is in location s. After the elapse of a positive amount of time ($\mathtt{z}_{\mathtt{t-1}} < \mathtt{z}_\mathtt{t}$), after which the invariant $\mathtt{I}(\mathtt{s})_{\mathtt{t}\Delta}$ of s and the clock guard $\mathtt{cc}_{\mathtt{t}\Delta}$ of the transition hold, $\mathfrak{T}$ switches to location s', the invariant of which has to hold. All clocks are updated according to their value under λ, data flows through all ports p contained in the port set P, while the other ports are inactive, and the data constraint $\mathtt{dc}_\mathtt{t}$ is satisfied. Memory cells which are not used by the target location s' are empty after execution of the transition, i.e. they get the value $\perp$. As for TA (cf. explanations after Definition 3.1.1), convexity allows to check the invariant at the end of the time delay only, as it inductively holds at the beginning (3.10). The execution of an invisible transition (3.12) is similar, except that the amount of time elapsed may be zero, and data must not flow through any port. The disjunction of all visible and invisible transitions expresses nondeterministic transition choice (3.13). In any step, the current location is unique (3.14), the special value "no data" may only be pending at inactive ports, and may only be "contained" in a memory cell if that memory cells is indicated to be empty (3.15).

Remark 3.1.5 (Representation of $\perp$). As explained in Section 3.1.1, we leave the value of $\mathtt{n}^{\perp}$ initially unspecified. During Bounded Model Checking (see Section 3.2 for details), if a satisfying assignment for $\varphi(\mathfrak{T})$ exists, the solver will find and assign to $\perp$ a integer value such that the constraints in Table 3.4 are satisfied.

The actual value which is assigned to $\perp$ is not important. By construction, the constraints in Table 3.4 ensure that the value is different from all $\mathtt{n}^{\mathtt{i}}$ used in the constraints, i.e., from (the representation of) all data values pending at any active port or contained in any memory cell. Since we allow $\perp$ to be used in data constraints only in combination with equality (but not with $\leqslant$, cf. Definition 2.1.7), this uniqueness ensures that a data constraint $(\mathtt{Dp_t}{=}\perp)$ is satisfied iff port p is inactive in step $\mathtt{t}$, and a data constraint $(\mathtt{Dd_t}{=}\perp)$ is satisfied iff memory cell d is empty in step $\mathtt{t}$.

For finite data domains, we can make the following improvement to $\varphi(\mathfrak{T})$.

Remark 3.1.6 (Finite Data Domain). For finite domains, i.e., with $|\mathcal{D}ata\backslash\perp|{=}k$ for some $k{\in}\mathbb{N}$, we require that Δ maps the elements of $\mathcal{D}ata$ to subsequent integer numbers, with smallest element $\Delta(\perp){=}{-1}$, such that $\Delta(\mathcal{D}ata){=}\{-1,0,\ldots,k{-}1\}{\subset}\mathbb{Z}$. Further, we add the constraint

$$\bigwedge_{p\in\mathcal{P}}(\mathtt{Dp_{t+1}}{\le}k{-}1){\wedge}(\mathtt{Dp_{t+1}}{\ge}{-1}){\wedge}\bigwedge_{d\in\mathcal{D}}(\mathtt{Dd_{t+1}}{\le}k{-}1){\wedge}(\mathtt{Dd_{t+1}}{\ge}{-1})$$

to φ^{mutex} (3.15). This speeds up verification, since the number of possible valuations for ports and memory cells is decreased.

Example 3.1.7 (TCA Representation). Consider again the 1-bounded FIFO buffer presented in Figure 2.5. Let $\mathfrak{T}$ be the name of the automaton, let e_1, e_2, e_3 refer to the transitions from *empty* to *full*, *full* to *empty* (visible), and *full* to *empty* (invisible), respectively. The representation of $\mathfrak{T}$ is shown in Table 3.5. Again, we omit constraints equal to $\mathtt{true}$.

We now present a *linear* representation of products of TCA, which avoids the worst case exponential blow-up of the product definition in Definition 2.3.9. As for TA, the basic idea is to define the representation of the product as the conjunction of the individual representations. We require variables representing common ports to have the same name in both representations, such that constraints involving these ports are automatically satisfied simultaneously in both representations.

To correctly model transitions described by (2.11) in Definition 2.3.9, we first need to introduce explicit delay transitions: as explained after Definition 2.3.9, in case the transition described by (2.11) is preceded by a time delay, the other automaton actually performs a delay transition. The representation of a delay transition $\varphi^{delay}(s)$ in location s is defined in (3.17).

$$
\begin{aligned}
\varphi^{init}(\mathfrak{T}) =\;& \mathtt{empty_0}\wedge\neg\mathtt{full_0}\wedge\neg\mathtt{p_0}\wedge(\mathtt{Dp_0}{=}\mathtt{n}^{\perp})\wedge\neg\mathtt{q_0}\wedge(\mathtt{Dq_0}{=}\perp)\wedge\\
& \neg\mathtt{m_0}\wedge(\mathtt{Dm_0}{=}\mathtt{n}^{\perp})\wedge(\mathtt{z_0}{=}0)\wedge(\mathtt{x_0}{=}0)\\
\varphi^{visible}(e_1) =\;& \mathtt{empty_t}\wedge\mathtt{p_{t+1}}\wedge\neg\mathtt{q_{t+1}}\wedge(\mathtt{Dp_{t+1}}{=}\mathtt{m_{t+1}})\wedge(\mathtt{z_t}{<}\mathtt{z_{t+1}})\wedge\\
& (\mathtt{x_{t+1}}{=}\mathtt{z_{t+1}})\wedge\mathtt{full_{t+1}}\wedge(\mathtt{z_{t+1}}-\mathtt{x_{t+1}}{\leq}3)\\
\varphi^{visible}(e_2) =\;& \mathtt{full_t}\wedge\mathtt{q_{t+1}}\wedge\neg\mathtt{p_{t+1}}\wedge(\mathtt{Dm_{t+1}}{=}\mathtt{n}^{\perp})\wedge(\mathtt{Dq_{t+1}}{=}\mathtt{m_t})\wedge(\mathtt{z_{t+1}}-\mathtt{x_t}{<}3)\wedge\\
& (\mathtt{z_t}{<}\mathtt{z_{t+1}})\wedge(\mathtt{x_{t+1}}{=}\mathtt{x_t})\wedge(\mathtt{z_{t+1}}-\mathtt{x_t}{\leq}3)\wedge\mathtt{empty_{t+1}}\\
\varphi^{invisible}(e_3) =\;& \mathtt{full_t}\wedge(\mathtt{z_{t+1}}-\mathtt{x_t}{\leq}3)\wedge\neg\mathtt{p_{t+1}}\wedge\neg\mathtt{q_{t+1}}\wedge(\mathtt{Dm_{t+1}}{=}\mathtt{n}^{\perp})\wedge(\mathtt{z_{t+1}}-\mathtt{x_t}{=}3)\wedge\\
& (\mathtt{z_t}{\leq}\mathtt{z_{t+1}})\wedge(\mathtt{x_{t+1}}{=}\mathtt{x_t})\wedge\mathtt{empty_{t+1}}\\
\varphi^{trans}(\mathfrak{T}) =\;& \varphi^{visible}(e_1)\vee\varphi^{visible}(e_2)\vee\varphi^{invisible}(e_3)\\
\varphi^{location}(\mathfrak{T}) =\;& (\mathtt{empty_{t+1}}\wedge\neg\mathtt{full_{t+1}})\vee(\mathtt{full_{t+1}}\wedge\neg\mathtt{empty_{t+1}})\\
\varphi^{mutex}(\mathfrak{T}) =\;& (\neg\mathtt{p_{t+1}}\vee\neg(\mathtt{Dp_{t+1}}{=}\mathtt{n}^{\perp}))\wedge(\mathtt{p_{t+1}}\vee(\mathtt{Dp_{t+1}}{=}\mathtt{n}^{\perp}))\wedge\\
& (\neg\mathtt{q_{t+1}}\vee\neg(\mathtt{Dq_{t+1}}{=}\mathtt{n}^{\perp}))\wedge(\mathtt{q_{t+1}}\vee(\mathtt{Dq_{t+1}}{=}\mathtt{n}^{\perp}))\wedge\\
& (\neg\mathtt{m_{t+1}}\vee\neg(\mathtt{Dm_{t+1}}{=}\mathtt{n}^{\perp}))\wedge(\mathtt{m_{t+1}}\vee(\mathtt{Dm_{t+1}}{=}\mathtt{n}^{\perp}))\\
\varphi(\mathfrak{T}) =\;& \varphi^{init}(\mathfrak{T})\wedge\varphi^{trans}(\mathfrak{T})\wedge\varphi^{location}(\mathfrak{T})\wedge\varphi^{mutex}(\mathfrak{T})
\end{aligned}
$$

Table 3.5: Transition Relation Representation of TCA: Example

$$
\begin{aligned}
\varphi^{delay}(s) =\;& \mathtt{s_t}\wedge\bigwedge_{p\in\mathcal{P}}\neg\mathtt{p_{t+1}}\wedge\bigwedge_{d\in\mathcal{D}}(\mathtt{Dd_{t+1}}{=}\mathtt{Dd_t})\wedge\bigwedge_{x\in\mathcal{X}}(\mathtt{x_{t+1}}{=}\mathtt{x_t})\wedge(\mathtt{z_t}{\leq}\mathtt{z_{t+1}})\wedge\\
& \mathtt{I(s)_{t\Delta}}\wedge\mathtt{s_{t+1}}\wedge\mathtt{I(s)_{t+1}}
\end{aligned}
\tag{3.17}
$$

Note that these delay transitions are in accordance with Definition 2.3.1, since they correspond to the representation of invisible transitions (cf. (3.12)) of the form $(s, \emptyset, \bigwedge_{d\in\mathcal{D}}(s.d{=}t.d), \mathtt{true}, id, s)$. Therefore, in particular, (3.17) permits zero-delays.

Definition 3.1.8 (TCA Product Representation). Let $\mathfrak{T}_1$, $\mathfrak{T}_2$ be TCA, with $\mathcal{X}_1\cap\mathcal{X}_2{=}\emptyset$ and $S_1\cap S_2{=}\emptyset$, let $\varphi(\mathfrak{T}_1)$ and $\varphi(\mathfrak{T}_2)$ be the respective representations, as defined in Definition 3.1.4, with (3.13) replaced by (3.13') for $i{=}1,2$. The *formula representation* $\varphi(\mathfrak{T}_1\bowtie\mathfrak{T}_2)$ *of the product* $\mathfrak{T}_1\bowtie\mathfrak{T}_2$ is defined in (3.18).

$$
\varphi^{trans}(\mathfrak{T}_i) = \bigvee_{e\in E_i, P\neq\emptyset}\varphi^{visible}(e)\vee\bigvee_{e'\in E_i, P=\emptyset}\varphi^{invisible}(e')\vee\bigvee_{s\in S_i}\varphi^{delay}(s) \tag{3.13'}
$$

$$
\varphi(\mathfrak{T}_1\bowtie\mathfrak{T}_2) = \varphi(\mathfrak{T}_1)\wedge\varphi(\mathfrak{T}_2)\wedge\bigwedge_{s\in S_1}\varphi^{delay}(s)\wedge\bigwedge_{s\in S_2}\varphi^{delay}(s) \tag{3.18}
$$

The product representation (3.18) faithfully models the intended behaviour, as defined in Definition 2.3.9, but is still *linear* in the size of the underlying TCA. Note that the existence of such a linear product is not immediately clear, but in fact is a result of our design decision of explicitly mentioning all ports—active and inactive—on each transition (cf. (3.11), (3.12) and (3.17)). This decision—though seeming unnecessary at first glance—together with the assumption that common ports have the same name in both TCA, ensures that transitions in different TCA may only be executed in parallel (i.e., synchronise) if they fulfil the conditions described in Definition 2.3.9. In this way, we do not need to list all possible synchronisations (which are allowed by (2.10) and (2.11)) explicitly, and in this way avoid the exponential blow-up.

The hiding operation (cf. Definition 2.3.12) removes all information about a set of ports O from a TCA $\mathfrak{T}$. Hiding a set of ports O in the formula representation $\varphi(\mathfrak{T})$ amounts to existential quantification over the corresponding variables, i.e., port activity and data variables of the ports in O. For a TCA $\mathfrak{T}$, with formula representation $\varphi(\mathfrak{T})$, and a port set $O \subseteq \mathcal{P}$, the formula representation $\varphi(\mathfrak{T}\backslash_O)$ of automaton $\mathfrak{T}\backslash_O$ corresponds to

$$\exists\, \mathtt{O}\, \varphi(\mathfrak{T}), \tag{3.19}$$

with $\mathtt{O} = \bigcup_{p \in O} \{\mathtt{p_t}, \mathtt{Dp_t}\}$

In Definition 2.3.12, an additional clock is introduced to ensure correct timed behaviour of invisible transitions in $\mathfrak{T}\backslash_O$ which originate from visible transitions in $\mathfrak{T}$. Here, we do not need to introduce an additional clock: the formula representation of a visible transition explicitly requires a positive amount of time to elapse ($(\mathtt{z_{t\text{-}1}} < \mathtt{z_t})$, cf. (3.11)). Since $\mathtt{O}$ does not contain clock variables, this constraint remains unchanged even in case the transition becomes invisible, and therefore, correct timed behaviour, as required by Definition 2.3.12, is guaranteed.

3.1.4 Timed Network Automata

The representation of TNA follows the same ideas as the representations of TA and TCA. For clocks, clock constraints, locations, memory cells (data variables) and data constraints, we use the concepts introduced in Section 3.1.1. Yet, for ports of TNA, we need to extend the encoding with an additional variable, to be able to identify—in case of no dataflow—where the reason to delay comes from (cf. Section 2.4.1).

As defined in Section 3.1.1.4, for every port $p \in \mathcal{P}$, we have a Boolean variable $\mathtt{p_t}$ (port activity variable), with the intended meaning that $\mathtt{p_t}$ evaluates to `true` iff data flows through port p in step `t`, and an integer variable $\mathtt{Dp_t}$ (port data variable), which represents the data value pending at p in step `t`. In addition, the Boolean variable $\mathtt{cp_t}$ (called *port colour variable*) denotes where the reason for delay comes from in case $\mathtt{p_t}$ evaluates to `false`. For a given colouring $\mathbb{c}$, the *representation of* p *under* $\mathbb{c}$ *in step* `t`, denoted $\langle \mathtt{p}_{\mathbb{c}} \rangle_{\mathtt{t}}$, is given by $\neg\mathtt{p_t} \wedge \neg\mathtt{cp_t}$ iff $\mathbb{c}(p) = \text{-?-}$, by $\neg\mathtt{p_t} \wedge \mathtt{cp_t}$ iff $\mathbb{c}(p) = \text{-!-}$, and by $\mathtt{p_t} \wedge (\mathtt{cp_t} \vee \neg\mathtt{cp_t})$ iff $\mathbb{c}(p) = \text{——}$. In the latter case, the representation simplifies to $_{\mathtt{t}}$.

The representation of a TNA $\mathfrak{N}$ is now given as follows.

Definition 3.1.9 (TNA Representation). Let $\mathfrak{N}$ be a TNA, with initial location $\bar{s}$ (as before, we denote the initial location as $\bar{s}$ rather than s_0), $f = (s, \mathbb{c}, dc, cc, \lambda, s') \in E$ a communication, and $d = (s, \mathbb{c}, dc, cc, id, s) \in E$ a delay. The *formula representation of the transition relation of* $\mathfrak{N}$, denoted $\varphi(\mathfrak{N})$, is defined in (3.26) in Table 3.6.

$$\begin{aligned}
\varphi^{init}(\mathfrak{N}) = {} & \bar{\mathtt{s}}_0 \wedge \bigwedge_{s\in S, s\neq\bar{s}} \neg\mathtt{s}_0 \wedge \mathtt{I}(\bar{\mathtt{s}})_0 \wedge \bigwedge_{p\in\mathcal{P}} (\neg\mathtt{p}_0 \wedge (\mathtt{Dp}_0 = \mathtt{n}^{\perp}) \wedge \mathtt{cp}_0) \wedge && (3.20)\\
& \bigwedge_{d\in\mathcal{D}} (\neg\mathtt{d}_0 \wedge (\mathtt{Dd}_0 = \mathtt{n}^{\perp})) \wedge (\mathtt{z}_0 = 0) \wedge \bigwedge_{x\in\mathcal{X}} (\mathtt{x}_0 = 0) \\
\varphi^{commu}(f) = {} & \mathtt{s}_\mathtt{t} \wedge \bigwedge_{p\in\mathcal{P}} \langle \mathtt{p}_{\mathbb{c}} \rangle_{\mathtt{t+1}} \wedge \bigwedge_{d\notin\#(s')} \neg\mathtt{d}_{\mathtt{t+1}} \wedge \mathtt{dc}_{\mathtt{t+1}} \wedge \mathtt{cc}_\mathtt{t} \wedge (\mathtt{z}_{\mathtt{t+1}} = \mathtt{z}_\mathtt{t}) \wedge && (3.21)\\
& \bigwedge_{\lambda(x)=id} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{x}_\mathtt{t}) \wedge \bigwedge_{\lambda(x)=x'} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{x'}_{\mathtt{t+1}}) \wedge \\
& \bigwedge_{\lambda(x)=n} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{z}_{\mathtt{t+1}} - n) \wedge \mathtt{s'}_{\mathtt{t+1}} \wedge \mathtt{I}(\mathtt{s'})_{\mathtt{t+1}} \\
\varphi^{delay}(d) = {} & \mathtt{s}_\mathtt{t} \wedge \bigwedge_{p\in\mathcal{P}} \langle \mathtt{p}_{\mathbb{c}} \rangle_{\mathtt{t+1}} \wedge \bigwedge_{d\in\mathcal{D}} (\mathtt{Dd}_{\mathtt{t+1}} = \mathtt{Dd}_\mathtt{t}) \wedge \mathtt{dc}_{\mathtt{t+1}} \wedge \mathtt{cc}_\mathtt{t} \wedge (\mathtt{z}_{\mathtt{t+1}} \geq \mathtt{z}_\mathtt{t}) \wedge && (3.22)\\
& \bigwedge_{x\in\mathcal{X}} (\mathtt{x}_{\mathtt{t+1}} = \mathtt{x}_\mathtt{t}) \wedge \mathtt{cc}_{\mathtt{t+1}} \wedge \mathtt{s}_{\mathtt{t+1}} \wedge \mathtt{I}(\mathtt{s})_{\mathtt{t+1}} \\
\varphi^{trans}(\mathfrak{N}) = {} & \bigvee_{f \text{ comm.}} \varphi^{commu}(f) \vee \bigvee_{d \text{ delay}} \varphi^{delay}(d) && (3.23)\\
\varphi^{location}(\mathfrak{N}) = {} & \bigvee_{s\in S} (\mathtt{s}_{\mathtt{t+1}} \wedge \bigwedge_{s'\in S, s'\neq s} \neg\mathtt{s}'_{\mathtt{t+1}}) && (3.24)\\
\varphi^{mutex}(\mathfrak{N}) = {} & \bigwedge_{p\in\mathcal{P}} (\neg\mathtt{p}_{\mathtt{t+1}} \vee \neg(\mathtt{Dp}_{\mathtt{t+1}} = \mathtt{n}^{\perp})) \wedge (\mathtt{p}_{\mathtt{t+1}} \vee (\mathtt{Dp}_{\mathtt{t+1}} = \mathtt{n}^{\perp})) \wedge && (3.25)\\
& \bigwedge_{d\in\mathcal{D}} (\neg\mathtt{d}_{\mathtt{t+1}} \vee \neg(\mathtt{Dd}_{\mathtt{t+1}} = \mathtt{n}^{\perp})) \wedge (\mathtt{d}_{\mathtt{t+1}} \vee (\mathtt{Dd}_{\mathtt{t+1}} = \mathtt{n}^{\perp})) \\
\varphi(\mathfrak{N}) = {} & \varphi^{init}(\mathfrak{N}) \wedge \varphi^{trans}(\mathfrak{N}) \wedge \varphi^{location}(\mathfrak{N}) \wedge \varphi^{mutex}(\mathfrak{N}) && (3.26)
\end{aligned}$$

Table 3.6: Transition Relation Representation of TNA

The TNA starts in its initial location, the invariant of which holds, all ports are inactive, all memory cells are empty, and all clocks are set to zero (3.20).[5] The representation of a communication (3.21) ensures that the TNA is in location s before firing, and the clock guard cc holds. On execution of the transition, data flows according to colouring $\mathbb{c}$, the data values satisfy the data guard dc, all clocks are updated according to their value under λ, while the value of the absolute time reference z does not change, and memory cells not used by the target location loose their contents. After firing, the TNA is in location s', the invariant of which holds. The representation of a delay (3.22) is similar, except that the value of the absolute time reference increases, while all other clocks keep their value, and all memory cells keep their values as well. In addition, clock guard cc still needs to be satisfied after the time delay. Again, convexity of clock constraints allows us to check the invariant at the end of the time delay only, as it inductively holds at the beginning (3.20). The disjunction of these formulas expresses (nondeterministic) transition choice (3.23). In any step, the current location is unique (3.24), the special value "no data" may only be pending at inactive ports, and may only be "contained" in a memory cell if

[5]Note that it is not necessary to specify initial values for the port colour variables $\mathtt{cp}_0$, since the constraint $\neg\mathtt{p}_0$ is sufficient to express inactivity of port p. Yet, adding a valuation reduces the number of unspecified variables, and in this way speeds up verification.

that memory cells is indicated to be empty (3.25).

The results of Remark 3.1.5 (representation of $\bot$) and Remark 3.1.6 (representation of finite data domains) directly carry over from TCA to TNA.

Example 3.1.10 (TNA Representation). Consider again the 1-bounded FIFO buffer presented in Figure 2.8. Let $\mathfrak{N}$ be the name of the TNA, let f_1, f_2, f_3 refer to the communications from *empty* to *full*, *full* to *empty* (with clock guard $x<3$), and from *full* to *empty* (with clock guard $x=3$), respectively, and let d_1, d_2 refer to the delays in *empty* and *full*, respectively. The representation of $\mathfrak{N}$ is shown in Table 3.7. We omit constraints equal to `true`.

$$
\begin{aligned}
\varphi^{init}(\mathfrak{N}) &= \mathtt{empty_0}\wedge\neg\mathtt{full_0}\wedge\neg\mathtt{r_0}\wedge(\mathtt{Dr_0}{=}\mathtt{n}^\bot)\wedge\mathtt{cr_0}\wedge\neg\mathtt{w_0}\wedge(\mathtt{Dw_0}{=}\mathtt{n}^\bot)\wedge\mathtt{cw_0}\wedge\\
&\quad(\neg\mathtt{m_0}\wedge(\mathtt{Dm_0}{=}\mathtt{n}^\bot))\wedge(\mathtt{z_0}{=}0)\wedge(\mathtt{x_0}{=}0)\\
\varphi^{commu}(f_1) &= \mathtt{empty_t}\wedge\mathtt{r_{t+1}}\wedge(\neg\mathtt{w_{t+1}}\wedge\mathtt{cw_{t+1}})\wedge(\mathtt{Dr_{t+1}}{=}\mathtt{Dm_{t+1}})\wedge(\mathtt{z_{t+1}}{=}\mathtt{z_t})\wedge\\
&\quad(\mathtt{x_{t+1}}{=}\mathtt{z_{t+1}})\wedge\mathtt{full_{t+1}}\wedge(\mathtt{z_{t+1}}{-}\mathtt{x_{t+1}}{\le}3)\\
\varphi^{commu}(f_2) &= \mathtt{full_t}\wedge(\neg\mathtt{r_{t+1}}\wedge\mathtt{cr_{t+1}})\wedge\mathtt{w_{t+1}}\wedge\neg\mathtt{m_{t+1}}\wedge(\mathtt{Dw_{t+1}}{=}\mathtt{Dm_t})\wedge\\
&\quad(\mathtt{z_t}{-}\mathtt{x_t}{<}3)\wedge(\mathtt{z_{t+1}}{=}\mathtt{z_t})\wedge(\mathtt{x_{t+1}}{=}\mathtt{x_t})\wedge\mathtt{empty_{t+1}}\\
\varphi^{commu}(f_3) &= \mathtt{full_t}\wedge(\neg\mathtt{r_{t+1}}\wedge\mathtt{cr_{t+1}})\wedge(\neg\mathtt{w_{t+1}}\wedge\mathtt{cw_{t+1}})\wedge\neg\mathtt{m_{t+1}}\wedge(\mathtt{z_t}{-}\mathtt{x_t}{=}3)\wedge\\
&\quad(\mathtt{z_{t+1}}{=}\mathtt{z_t})\wedge(\mathtt{x_{t+1}}{=}\mathtt{x_t})\wedge\mathtt{empty_{t+1}}\\
\varphi^{delay}(d_1) &= \mathtt{empty_t}\wedge(\neg\mathtt{r_{t+1}}\wedge\neg\mathtt{cr_{t+1}})\wedge(\neg\mathtt{w_{t+1}}\wedge\mathtt{cw_{t+1}})\wedge(\mathtt{Dm_{t+1}}{=}\mathtt{Dm_t})\wedge\\
&\quad(\mathtt{z_{t+1}}{\ge}\mathtt{z_t})\wedge(\mathtt{x_{t+1}}{=}\mathtt{x_t})\wedge\mathtt{empty_{t+1}}\\
\varphi^{delay}(d_2) &= \mathtt{full_t}\wedge(\neg\mathtt{r_{t+1}}\wedge\mathtt{cr_{t+1}})\wedge(\neg\mathtt{w_{t+1}}\wedge\neg\mathtt{cw_{t+1}})\wedge(\mathtt{Dm_{t+1}}{=}\mathtt{Dm_t})\wedge(\mathtt{z_t}{-}\mathtt{x_t}{\le}3)\wedge\\
&\quad(\mathtt{z_{t+1}}{\ge}\mathtt{z_t})\wedge(\mathtt{x_{t+1}}{=}\mathtt{x_t})\wedge(\mathtt{z_{t+1}}{-}\mathtt{x_{t+1}}{\le}3)\wedge\mathtt{full_{t+1}}\wedge(\mathtt{z_{t+1}}{-}\mathtt{x_{t+1}}{\le}3)\\
\varphi^{trans}(\mathfrak{N}) &= \varphi^{commu}(f_1)\vee\varphi^{commu}(f_2)\vee\varphi^{commu}(f_3)\vee\varphi^{delay}(d_1)\vee\varphi^{delay}(d_2)\\
\varphi^{location}(\mathfrak{N}) &= (\mathtt{empty_{t+1}}\wedge\neg\mathtt{full_{t+1}})\vee(\mathtt{full_{t+1}}\wedge\neg\mathtt{empty_{t+1}})\\
\varphi^{mutex}(\mathfrak{N}) &= (\neg\mathtt{r_{t+1}}\vee\neg(\mathtt{Dr_{t+1}}{=}\mathtt{n}^\bot))\wedge(\mathtt{r_{t+1}}\vee(\mathtt{Dr_{t+1}}{=}\mathtt{n}^\bot))\wedge\\
&\quad(\neg\mathtt{w_{t+1}}\vee\neg(\mathtt{Dw_{t+1}}{=}\mathtt{n}^\bot))\wedge(\mathtt{w_{t+1}}\vee(\mathtt{Dw_{t+1}}{=}\mathtt{n}^\bot))\wedge\\
&\quad(\neg\mathtt{m_{t+1}}\vee\neg(\mathtt{Dm_{t+1}}{=}\mathtt{n}^\bot))\wedge(\mathtt{m_{t+1}}\vee(\mathtt{Dm_{t+1}}{=}\mathtt{n}^\bot))\wedge\\
\varphi(\mathfrak{N}) &= \varphi^{init}(\mathfrak{N})\wedge\varphi^{trans}(\mathfrak{N})\wedge\varphi^{location}(\mathfrak{N})\wedge\varphi^{mutex}(\mathfrak{N})
\end{aligned}
$$

Table 3.7: Transition Relation Representation of TNA: Example

Though the flip rule (Remark 2.4.11) reduces the size of TNA, the size of a composed TNA is still exponential in the worst case. We now present a *linear size* representation of the composition of TNA. The basic idea is similar to the product definition of TA and TCA (cf. Definitions 3.1.3 and 3.1.8): we do not explicitly compute the composition, but instead retain the representations of the single TNA, and define the representation of the composition via conjunction. Unfortunately,[6]

[6]Actually, we do not consider this a disadvantage, since we gain a composition that is linear.

this does not allow us to explicitly remove the ports contained in a merge set from the representation, and replace them by the same data variable (cf. Definition 2.4.13). Instead, we need to add additional constraints to ensure that (1) the representation of the composition correctly models the dataflow behaviour of the resulting internal port (cf. Definition 2.4.9), and that (2) all ports in the merge set agree on the same data value (cf. Definition 2.4.13 and preceding explanations).

Definition 3.1.11 (Internal Port Representation). Let $\mathcal{Q}$ be a merge set, $\mathcal{Q}^r \subseteq \mathcal{Q}$ and $\mathcal{Q}^w \subseteq \mathcal{Q}$ the subsets of read respectively write ports in $\mathcal{Q}$. For $p \in Q$, let $\mathtt{p_t}$, $\mathtt{Dp_t}$ and $\mathtt{cp_t}$ be the port activity, port data and port colour variable, respectively, let d be a fresh data variable (i.e., not yet used elsewhere), with data fullness variable $\mathtt{d_t}$ and data content variable $\mathtt{Dd_t}$. The *representation* $\varphi^{int\_port}(\mathcal{Q})$ *of internal port* $p_{\prec\mathcal{Q}}$ is given in (3.27).

$$\begin{aligned}
\varphi^{valid\_col1}(\mathcal{Q}) &= \bigvee_{w\in\mathcal{Q}^w} \mathtt{w_{t+1}} \rightarrow \Big(\bigwedge_{r\in\mathcal{Q}^r} \mathtt{r_{t+1}} \wedge \bigwedge_{w,w'\in\mathcal{Q}^w, w\neq w'} \neg(\mathtt{w_{t+1}} \wedge \mathtt{w'_{t+1}})\Big) \\
\varphi^{valid\_col2}(\mathcal{Q}) &= \bigvee_{r\in\mathcal{Q}^r} \mathtt{r_{t+1}} \rightarrow \bigvee_{w\in\mathcal{Q}^w} \mathtt{w_{t+1}} \\
\varphi^{valid\_col3}(\mathcal{Q}) &= \bigwedge_{p\in\mathcal{Q}} \neg\mathtt{p_{t+1}} \rightarrow \Big(\bigwedge_{w\in\mathcal{Q}^w} \mathtt{cw_{t+1}} \vee \bigvee_{r\in\mathcal{Q}^r} \mathtt{cr_{t+1}} \Big) \\
\varphi^{data\_flow}(\mathcal{Q}) &= \bigwedge_{p\in\mathcal{Q}} \neg\mathtt{p_{t+1}} \vee (\mathtt{Dp_{t+1}} = \mathtt{Dd_{t+1}}) \\
\varphi^{int\_port}(\mathcal{Q}) &= \varphi^{valid\_col1}(\mathcal{Q}) \wedge \varphi^{valid\_col2}(\mathcal{Q}) \wedge \varphi^{valid\_col3}(\mathcal{Q}) \wedge \varphi^{data\_flow}(\mathcal{Q})
\end{aligned} \tag{3.27}$$

The first three constraints directly correspond to the three conditions in Definition 2.4.9. For example, $\varphi^{valid\_col3}(\mathcal{Q})$ describes the constraints in condition 3 in Definition 2.4.9: if there is no flow at all ($\bigwedge_{p\in\mathcal{Q}} \neg\mathtt{p_t}$), then either all write ports provide a reason for delay ($\bigwedge_{w\in\mathcal{Q}^w} \mathtt{cw_t}$), or at least one read port provides a reason for delay ($\bigvee_{r\in\mathcal{Q}^r} \mathtt{cr_t}$). These three constraints thus capture *whether* data flows through $p_{\prec\mathcal{Q}}$. The fourth constraint $\varphi^{data\_flow}(\mathcal{Q})$—the conjuncts should be read as $\mathtt{p_t} \rightarrow (\mathtt{Dp_t} = \mathtt{Dd_t})$—expresses the fact that all active ports in the merge set (if $\mathtt{p_t}$ holds, p is active, cf. the beginning of Section 3.1.4) agree on the same data value, we use the data variable d as a placeholder for any possible data value. This constraint thus captures *which* data flows through $p_{\prec\mathcal{Q}}$.

Using this, the representation of TNA composition is defined as follows.

Definition 3.1.12 (TNA Composition Representation). Let $\mathcal{N}$ be a set of disjoint TNA, $\mathcal{Q}$ a set of disjoint merge sets over ports of TNA in $\mathcal{N}$. The *formula representation* $\varphi(\mathcal{N}\bowtie_{\mathcal{Q}})$ *of the composed TNA* $\mathcal{N}\bowtie_{\mathcal{Q}}$ is defined as

$$\varphi(\mathcal{N}\bowtie_{\mathcal{Q}}) = \bigwedge_{\mathfrak{N}\in\mathcal{N}} \varphi(\mathfrak{N}) \wedge \bigwedge_{\mathcal{Q}'\in\mathcal{Q}} \varphi^{int\_port}\mathcal{Q}' \tag{3.28}$$

To accommodate the fact that a port cannot be merged more than once (cf. beginning of Section 2.4.3), which now can be translated to “cannot be contained in more than one merge set”, we hide the ports in a merge set, using existential quantification: the *reduction of* $\varphi(\mathcal{N}\bowtie_{\mathcal{Q}})$ *to the external interface* (i.e., the set of ports, cf. Definition 2.4.2) is defined as

$$\exists \bigcup_{\mathcal{Q}'\in\mathcal{Q}} \mathcal{Q}'(\varphi(\mathcal{N}\bowtie_{\mathcal{Q}}))$$

In this way, ports that are contained in any merge set in $\mathcal{Q}$ cannot be merged again when composing $\mathcal{N}\bowtie_{\mathcal{Q}}$ with another TNA.

3.2 Bounded Model Checking

In this section, we briefly recall the concepts of Bounded Model Checking (BMC), and show how they can be applied to the representations of real-time systems defined in Section 3.1.

Bounded Model Checking (BMC) [BCC+03, BCCZ99, CBRZ01] has evolved from Symbolic Model Checking (SMC) [McM93], and can be seen as a subcategory of it. SMC techniques represent the system symbolically, and typically rely on binary decision diagrams (BDDs) [Bry86]. These BDD representations can handle hundreds of variables, but often blow up in space. In addition, the efficiency highly depends on the variable ordering in the BDD, yet, the problem of finding an efficient order is NP-hard, that means, there exists no efficient way of determining an efficient ordering a priori.

BMC was introduced “in an attempt to replace BDDs with SAT in SMC” [Bie09]. The key idea is to represent the system and the property to be checked symbolically (using propositional formulas), examine prefix fragments of the transition system for whether the property holds, and successively increase the exploration bound until it reaches (a computable indicator of) the diameter of the system—in which case the results are guaranteed to be complete, and the property holds—or an unsafe run violating the property has been discovered.

Although BMC is complete in theory once the diameter of the system is reached, it is often impractical to increase the exploration bound that far (see Section 3.2.4 for a more detailed discussion). Therefore, BMC techniques focus on falsification of (temporal) properties. Such properties can be disproved with a finite counterexample, i.e., a finite run, where at least one of the configurations contains a contradiction to the property. Reachability properties are well-suited to express safety properties of the form “a certain behaviour should not happen”, where the erroneous behaviour is defined by the possibility to reach a certain error location.

We first introduce some notations in Section 3.2.1, and then formalise these notions in Sections 3.2.2 and 3.2.3.

3.2.1 Notations

In the remainder of this section, we use $\mathfrak{S}$ to refer to any of the system models defined in Chapter 2: $\mathfrak{S}\in\{\mathfrak{A}, \mathfrak{T}, \mathfrak{N}\}$, cf. Definitions 2.2.1, 2.3.1 and 2.4.2. We use $\varphi(\mathfrak{S})$ to refer to the corresponding formula representation of $\mathfrak{S}$: $\varphi(\mathfrak{S})\in\{\varphi(\mathfrak{A}), \varphi(\mathfrak{T}), \varphi(\mathfrak{N})\}$, for both simple (Definitions 3.1.1, 3.1.4 and 3.1.9) and composed (Definitions 3.1.3, 3.1.8 and 3.1.12) systems.

For a formula $\varphi(\mathfrak{S})$, we use $\mathit{Vars}(\varphi(\mathfrak{S}))$ to denote the *set of variables* of $\varphi(\mathfrak{S})$, we write $\mathit{Vars}(\varphi(\mathfrak{S}))|_{\mathbb{B}}$, $\mathit{Vars}(\varphi(\mathfrak{S}))|_{\mathbb{N}}$ and $\mathit{Vars}(\varphi(\mathfrak{S}))|_{\mathbb{Q}}$ to denote the subsets of $\mathit{Vars}(\varphi(\mathfrak{S}))$ of propositional, integer and rational variables, respectively. We use $\mathit{Atoms}(\varphi(\mathfrak{S}))$ to denote the *set of propositional atoms* of $\varphi(\mathfrak{S})$, and write $\mathit{Conts}(\varphi(\mathfrak{S}))$ ("contents") as an abbreviation for $\mathit{Atoms}(\varphi(\mathfrak{S}))\cup\mathit{Vars}(\varphi(\mathfrak{S}))$.

An *interpretation σ of (the variables in)* $\varphi(\mathfrak{S})$ is a mapping $\sigma\colon\mathit{Vars}(\varphi(\mathfrak{S})) \to (\mathbb{B}\cup\mathbb{N}\cup\mathbb{Q})$, assigning to each variable $v\in\mathit{Vars}(\varphi(\mathfrak{S}))$ a value from the appropriate range (i.e., $\sigma(v)\in\mathbb{B}$ iff $v\in\mathit{Vars}(\varphi(\mathfrak{S}))|_{\mathbb{B}}$, $\sigma(v)\in\mathbb{N}$ iff $v\in\mathit{Vars}(\varphi(\mathfrak{S}))|_{\mathbb{N}}$, and $\sigma(v)\in\mathbb{Q}$ iff $v\in\mathit{Vars}(\varphi(\mathfrak{S}))|_{\mathbb{Q}}$).

Interpretation σ is called *model of* $\varphi(\mathfrak{S})$, denoted $\sigma\models\varphi(\mathfrak{S})$, if it *satisfies* $\varphi(\mathfrak{S})$, i.e., $\varphi(\mathfrak{S})$ evaluates to `true` under valuation σ. $\varphi(\mathfrak{S})$ is called *satisfiable* if at least one model of $\varphi(\mathfrak{S})$ exists. We denote the set of all models of $\varphi(\mathfrak{S})$ by $\mathcal{V}(\varphi(\mathfrak{S}))$. If $\varphi(\mathfrak{S})$ evaluates to `true` under *all* valuations, then $\varphi(\mathfrak{S})$ is called *tautology*, denoted $\models\varphi(\mathfrak{S})$.

We lift the above notations in the straightforward way from $\varphi(\mathfrak{S})$ to all types of formulas.

3.2.2 Unfolding for BMC

The formula representation $\varphi(\mathfrak{S})$ of a real-time system $\mathfrak{S}$, as defined in the Section 3.1, describes the transition characteristics of $\mathfrak{S}$ in terms of "abstract" steps `t` and `t+1`. In order to describe the reachability problem of BMC for k steps, the formula representation $\varphi(\mathfrak{S})$ is *unfolded*, i.e., instantiated for all steps 1 up to bound k. The resulting formula $\varphi(\mathfrak{S})_k$ is called *k-unfolding*.

Definition 3.2.1 (k-unfolding). Let $\mathfrak{S}$ be a real-time system, with formula representation $\varphi(\mathfrak{S})$, let $k\in\mathbb{N}$, $k\geq 1$ (called *unfolding depth*). The *k-unfolding $\varphi(\mathfrak{S})_k$ of* $\varphi(\mathfrak{S})$ is defined as

$$\varphi(\mathfrak{S})_k = \varphi^{init}(\mathfrak{S})\wedge \bigwedge_{0\leq j\leq k-1} (\varphi^{trans}(\mathfrak{S})_{\mathtt{j/t}}\wedge\varphi^{location}(\mathfrak{S})_{\mathtt{j/t}}\wedge\varphi^{mutex}(\mathfrak{S})_{\mathtt{j/t}}), \qquad (3.29)$$

where $\psi_{\mathtt{j/t}}$ denotes a variant of formula ψ with index `t` replaced by `j`.

Intuitively, a model σ of $\varphi(\mathfrak{S})_k$ corresponds to a run of $\mathfrak{S}_{\mathfrak{S}}$ of length k, i.e., to one possible behaviour of $\mathfrak{S}$ for the first k steps. Consequently, the set $\mathcal{V}(\varphi(\mathfrak{S})_k)$ of all models of $\varphi(\mathfrak{S})_k$ describes all possible behaviours of $\mathfrak{S}$ for the first k steps.

Notation 3.2.2 (Unfolding). For $j\in\mathbb{N}$, $j\geq 0$, we write $\varphi^{trans}(\mathfrak{S})_{(\mathtt{j})}$ to denote the transition constraints from step `j` to step `j+1`, that is, $\varphi^{trans}(\mathfrak{S})_{(\mathtt{j})}=\varphi^{trans}(\mathfrak{S})_{\mathtt{j/t}}$. Equivalently, we write $\varphi^{location}(\mathfrak{S})_{(\mathtt{j})}$ respectively $\varphi^{mutex}(\mathfrak{S})_{(\mathtt{j})}$ to denote the mutual exclusion constraints on locations respectively events or ports in step `j`, that is, $\varphi^{location}(\mathfrak{S})_{(\mathtt{j})}=\varphi^{location}(\mathfrak{S})_{(\mathtt{j-1})/\mathtt{t}}$, and $\varphi^{mutex}(\mathfrak{S})_{(\mathtt{j})}=\varphi^{mutex}(\mathfrak{S})_{(\mathtt{j-1})/\mathtt{t}}$. Intuitively, for a formula $\psi_{(\mathtt{j})}$, `j` denotes the (smallest) index which appears on variables in ψ.

3.2.3 BMC of Properties

BMC is best suited for checking safety properties of the form "a certain behaviour should not happen", expressed through reachability of error locations. If an error location s is (not) reachable within k steps, s is called *(not) k-step reachable*. The safety property ϕ saying *s is not k-step reachable* is expressed by the formula

$$\phi = \bigwedge_{0 \le i \le k} \neg \mathtt{s_i}, \tag{3.30}$$

where $\mathtt{s}$ is the representation of s. To express that s is not reachable after *exactly* k-steps (for example because we already know that s is not $(k-1)$-step reachable) we simply choose $\phi = \neg \mathtt{s_k}$.

Checking whether a system $\mathfrak{S}$ is safe with respect to s amounts to conjoining the k-unfolding (3.29) with the negated property $\neg\phi$ (as explained above, BMC focusses on falsification of properties):

$$\varphi(\mathfrak{S})_k \wedge \bigvee_{0 \le i \le k} \mathtt{s_i}. \tag{3.31}$$

In this way, a model of (3.31) corresponds to a run of $\mathfrak{S}$ that contains error location s, that means to a system behaviour violating the property ϕ. In this case, $\mathfrak{S}$ is unsafe with respect to s, that means, the property ϕ does not hold in the system $\mathfrak{S}$. If no such model exists, i.e., the formula (3.31) is unsatisfiable, $\mathfrak{S}$ is safe with respect to s within bound k, but nothing can be said about safety within bounds $>k$.

Lifting (3.30) to reason about configurations or even execution sequences is straightforward. For example, the property "if the location in the current step is s, then the location in the next step will be s'" can be represented as

$$(\mathtt{s_0} \wedge \mathtt{s'_1}) \vee (\mathtt{s_1} \wedge \mathtt{s'_2}) \vee \ldots (\mathtt{s_{k\text{-}1}} \wedge \mathtt{s'_k}). \tag{3.32}$$

Other properties can be represented using the encoding in [ACKS02], where the authors have defined a translation from LTL to propositional formulas for BMC. For example, (3.32) corresponds to the LTL property $s \rightarrow \bigcirc\, s'$. We skip the details of the encoding, as they are beyond the scope of this thesis. In general, the only restriction we impose on formulas ϕ used as properties is that they may only reason about variables contained in $\varphi(\mathfrak{S})_k$, i.e., we require $\mathit{Vars}(\phi) \subseteq \mathit{Vars}(\varphi(\mathfrak{S})_k)$.

3.2.4 Completeness of BMC

BMC is used to inspect runs of a certain length k. If an error location is reachable within k steps, a counterexample to the safety property has been found. Otherwise, the exploration bound is increased, and the reachability check is repeated. The question remains whether there exists a *completeness threshold*, i.e., some bound k', for which we can safely conclude that the error location is not reachable, even when further increasing the exploration bound.

A first candidate is the diameter of the associated LTS $\mathfrak{S}_{\mathfrak{S}}$, reduced to runs which start in the initial configuration q_0.[7] This is indeed a completeness threshold: as soon as the exploration bound k is equal to the diameter, the results of BMC are complete, since BMC considers *all* runs of length k, and thus every reachable configuration will be reached by at least one of the runs. Unfortunately, no efficient algorithms exist to compute the diameter of a timed system.

Another candidate is the *recurrence diameter* of the associated LTS $\mathfrak{S}_{\mathfrak{S}}$, cf. also [BCCZ99]: the recurrence diameter of $\mathfrak{S}_{\mathfrak{S}}$ is the length of the longest loop-free run (cf. Definitions 2.2.4, 2.3.5 and 2.4.6). This is a completeness threshold as well: by definition, any run with length greater than the recurrence diameter must contain a loop, and thus cannot contain new configurations which have not been visited before. Unfortunately, the recurrence diameter can be considerably larger than the real diameter. Consider for example a fully connected graph with n nodes: while the diameter is 1 (every node is reachable from every other node), the recurrence diameter is $n-1$ (longest loop-free path). Yet, the recurrence diameter is more practical than the (real) diameter, since the fact that a run is loop-free can easily be expressed in our framework.

Definition 3.2.3 (Representation of Loop-Free Runs). Let $\mathfrak{S}$ be a real-time system, with set of locations S, set of clocks $\mathcal{X}$, and set of data variables $\mathcal{D}$ (only for TCA and TNA). Let $\varphi(\mathfrak{S})_k$ be the k-unfolding, and $\sigma \in \mathcal{V}(\varphi(\mathfrak{S})_k)$ a model of $\varphi(\mathfrak{S})_k$. The model σ corresponds to a *loop-free run* if it satisfies the *loop-free condition* $\varphi_k^{\textit{lp-free}}(\mathfrak{S})$, i.e. (in addition to $\sigma \models \varphi(\mathfrak{S})_k$) $\sigma \models \varphi_k^{\textit{lp-free}}(\mathfrak{S})$, where $\varphi_k^{\textit{lp-free}}(\mathfrak{S})$ is defined as

$$\varphi_k^{\textit{lp-free}}(\mathfrak{S}) = \bigwedge_{0 \le i < j \le k} \Big(\bigvee_{s \in S} \neg(\mathtt{s_i} = \mathtt{s_j}) \vee \bigvee_{x \in \mathcal{X}} \neg(\mathtt{x_i} = \mathtt{x_j}) \vee \bigvee_{d \in \mathcal{D}} \neg(\mathtt{Dd_i} = \mathtt{Dd_j}) \Big) \tag{3.33}$$

If $\mathfrak{S}$ is a TA, we omit the last disjunct.

For a run to be loop-free, all configurations need to be different. Configurations consist of (cf. Definitions 2.2.4, 2.3.5 and 2.4.6) the current location, the valuation of the clocks, and (only for TCA and TNA) the valuation of the data variables. Condition (3.33) expresses that for any two configurations (in steps i and j), at least one of these constituents is different.

We can use the loop-free condition in two ways: testing whether a model corresponds to a loop-free run, and calculating the recurrence diameter. The former is done by checking whether $\sigma \models \varphi_k^{\textit{lp-free}}(\mathfrak{S})$ for a model $\sigma \in \mathcal{V}(\varphi(\mathfrak{S})_k)$. To actually calculate the recurrence diameter, we inductively check (i.e., for increasing values of k) whether the formula $\varphi(\mathfrak{S})_k \wedge \varphi_k^{\textit{lp-free}}(\mathfrak{S})$ is satisfiable.[8] The recurrence diameter is the smallest k such that $\varphi(\mathfrak{S})_{k+1} \wedge \varphi_{k+1}^{\textit{lp-free}}(\mathfrak{S})$ is not satisfiable anymore.

[7]The diameter of a system—a well-known concept from graph theory—is a natural number, which corresponds to the longest shortest path between any two states. Here, the diameter is the longest shortest run from the initial configuration q_0 to any other configuration of $\mathfrak{S}_{\mathfrak{S}}$.

[8]Assuming that the subformula $\varphi(\mathfrak{S})_k$, when checked in isolation, is satisfiable for all values of k, i.e., we can always find a run of length k.

3.2.5 Correctness

Theorem 3.2.4 (Correctness). The formula representation $\varphi(\mathfrak{S})$ of a real-time system $\mathfrak{S}$ is correct, that means $\varphi(\mathfrak{S})$ exhibits the same behaviour as $\mathfrak{S}$.

The proof of Theorem 3.2.4 can be found in the Appendix, in Section A.1.

3.3 Discussion

In this section, for some of the constituents of real-time systems, we discuss other possible encodings, and motivate our design decisions.

3.3.1 Occurrence of Actions on Transitions of TA

The transition relation representation of TA models transitions from step `t` to step `t+1` (cf. (3.2) and (3.3)), with the action α occurring at step `t+1`. The choice of whether to model the occurrence of α at step `t` or `t+1` is to a certain extend arbitrary, since conceptually, the event occurs *while* taking the transition (that means, *in between* steps `t` and `t+1`). We believe both ways are intuitive by some means or other.

We decided to model the occurrence at step `t+1` since this is in accordance with the activity of ports in TCA and TNA, which also occurs at steps `t+1`. This results in a uniform handling of all “transition labels”, whether they are actions in TA or ports in TCA respectively TNA. We benefit from this point in the concretisation of abstract counterexamples, cf. Section 4.3.3.

3.3.2 Choice of Variable Types

When designing the formula representation for a real-time system, different aspects and requirements influence the design decisions, the two major ones being *universality* and *efficiency*. Universality in this context means that the resulting formulas should be platform independent and general enough such that they can be integrated seamlessly (or with only minor adaptations) into most (standard) frameworks. Efficiency is understood with respect to speed of verification.

To meet the first requirement, we have reduced the variable types used in the representation to rational, integer and Boolean variables. Rational variables are needed to represent real-time. Though clocks are real-valued (cf. Section 2.1.1), a rational encoding is sufficient here, since linear arithmetic using only integer constants (as is used in clock constraints, cf. Definition 2.1.2) is equisatisfiable for rational and real numbers. Integer variables are used for data variables and data values. We could have actually used rational variables for these as well, but this would have required additional constraints to restrict the admissible valuations of such variables, for example, to preserve the total order, cf. Definition 2.1.7. Boolean variables are used to represent the remaining constituents. Moreover, we reduce the number of arithmetic operations to addition, subtraction, equality and comparison.

To meet the second requirement, we use Boolean variables whenever possible. Propositional satisfiability (SAT solving) has been extensively studied in recent years (see for example [PBG05] for an overview); as a result, existing techniques are by now well-optimised and efficient, cf. for example [GN07, MMZ$^+$01]. SAT solvers internally work with formulas in conjunctive normal form (CNF).[9] We design our formulas in CNF whenever possible, to avoid unnecessary transformations in the SAT solver. Moreover, most of the CNF clauses are binary (two literals) or even unit (one literal) clauses. With respect to speed of verification, this is very efficient: the 2-SAT problem (i.e., with binary clauses only) is polynomial, and formulas with n unit clauses can be solved in $O(n)$. Formulas (3.5), (3.6), (3.14) and (3.24) could be expressed in CNF with binary clauses as well. Yet, they are not, but are rather tailored for abstraction already: after abstraction, the formulas in CNF would loose the information of mutual exclusion, and result in a too coarse (or even wrong) abstraction. Due to the disjunctive nature of transition choices, (3.4), (3.13) and (3.23) are not in CNF, but they could easily be transformed to short CNF (see e.g. [Häh93]) when introducing new symbols.

3.3.3 Temporal Difference Encoding of Clocks

For the representation of clock values, we use an approach similar to the one presented in [ACKS02]. We introduce a fresh clock z (the absolute time reference), to measure the absolute amount of time that has passed since the beginning of computation, cf. Section 3.1.1.1. The representation of the value of clock x in step $\mathtt{t}$ is given by the difference $\mathtt{z_t}-\mathtt{x_t}$ of the representation variables of z and x, this difference is also used in the representation of clock constraints. An update of clock x according to some update map λ is represented by setting $\mathtt{x_t}=\mathtt{z_t}-n$ iff $\lambda(x)=n$, and by setting $\mathtt{x_t}=\mathtt{x'_t}$ iff $\lambda(x)=x'$ for some clock x'. If $\lambda(x)=id$, the value of $\mathtt{x_t}$ carries over to the next step: $\mathtt{x_t}=\mathtt{x_{t+1}}$.

This design decision might seem unintuitive at first glance. Yet, the major advantage of the approach becomes clear when considering the representation of delays. In our approach, on execution of a delay from step $\mathtt{t}$ to step $\mathtt{t+1}$, all clock variables keep their values, only the value of the absolute time reference changes:

$$\bigwedge_{x\in\mathcal{X}} (\mathtt{x_t}=\mathtt{x_{t+1}})\wedge(\mathtt{z_t}\sim\mathtt{z_{t+1}}),\ \text{with}\ \sim\in\{\leq,<\} \tag{*}$$

(cf. (3.3), (3.11), (3.12), (3.17) and (3.22)). In an encoding which does not use the absolute time reference, on the other hand, *all* clock variables of step $\mathtt{t+1}$ are different from those in step $\mathtt{t}$. Furthermore, the representation needs take care of the fact that all clock variables change by the same amount. The encoding of clock variables in a delay step might look like

$$\bigwedge_{x\in\mathcal{X}} (\mathtt{x_{t+1}}-\mathtt{x_t}=\mathtt{d_t}),\ \text{with}\ \mathtt{d_t}\ \text{a rational variable.} \tag{**}$$

Here, $\mathtt{d_t}$ is a variable representing the amount of time for which the system delays.

[9] A formula is in conjunctive normal form, if it is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is a propositional atom or its negation.

With respect to speed of verification, (*) is more efficient than (**): in the former case, the values of all but one (namely $\mathtt{z}_{\mathtt{t+1}}$) clock variable in step $\mathtt{t+1}$ are predetermined by the values in step $\mathtt{t}$. Thus, the satisfiability of the (clock constraint) formulas of step $\mathtt{t+1}$ depends on one variable only. All conflict clauses which the solver learns while trying to find a model of the formula reason about $\mathtt{z}_{\mathtt{t+1}}$, which quickly restricts the number of possible valuations of $\mathtt{z}_{\mathtt{t+1}}$, and in this way leads to fewer backtracking steps. In the latter case, the satisfiability is subject to a number of free variables. Since all variables of step $\mathtt{t+1}$ depend on the values of a different variable of step $\mathtt{t}_{\mathtt{t}}$, each learnt conflict clause can only restrict the possible valuations of a single variable, which potentially leads to more backtracking steps.

A second advantage of the encoding using the absolute time reference is the fact that properties of the form "something happens after x time units" can be specified more easily, by using the difference $\mathtt{z}_{\mathtt{j}}-\mathtt{z}_{\mathtt{i}}$, for some $0\leq i<j\leq k$, $i,j\in\mathbb{N}$.

3.3.4 Linear Boolean Encoding of Finite Sets

After having chosen to represent the set of locations of a real-time system $\mathfrak{S}$ using Boolean variables (cf. Section 3.1.1), there are still two options: using a *linear* Boolean encoding or a *logarithmic* Boolean encoding.

Let $S=\{s_0,\ldots,s_{n-1}\}$ be the set of locations. In the linear encoding which we use in Section 3.1.1, we introduce one Boolean variable $\mathtt{s}$ for each location s, with the intended meaning that the localisation $\mathtt{s}_{\mathtt{t}}$ is true iff $\mathfrak{S}$ is in location s in step $\mathtt{t}$. Since $\mathfrak{S}$ can only be in one location at the same time, this encoding requires an additional mutual exclusion constraint, to ensure exactly one of the variables is true in every step (cf. (3.5), (3.14) and (3.24)).[10] This mutual exclusion constraint is quadratic in the number of locations.

For a logarithmic encoding, we would introduce a vector $[s]$ of $j=\lceil\log_2(n)\rceil$ Boolean variables, encoding a j-digit Boolean value. The intended meaning is that the localisation $[\mathtt{s}]_{\mathtt{t}}$ of $[s]$ encodes the value i, denoted by $[\mathtt{s}^{\mathtt{i}}]_{\mathtt{t}}$, iff the system $\mathfrak{S}$ is in location s_i in step $\mathtt{t}$. No additional mutual exclusion constraint is needed. Depending on the representation of transitions (see below), we might however need an additional constraint to disallow superfluous valuations of $[s]$, since in case $2^{\lceil\log_2(n)\rceil}>n$, the number of possible valuations of $[s]$ exceeds the actual number of locations.[11]

While the logarithmic encoding is slightly more efficient, due to the decreased number of variables, we have nevertheless chosen for Boolean encoding. One reason is that the differences in speed of verification are small, the real bottleneck is generated by rational clock variables. Moreover, abstraction of locations would be more involved when using a logarithmic encoding, and there is no straightforward way anymore of defining an abstraction function purely based on the syntactic categories of variables (cf. Definition 4.1.5).

[10]Observe that these mutual exclusion constraints could easily be expressed in CNF. Yet, they are not, but are already tailored for abstraction, cf. Chapter 4. After abstraction, an equivalent formula in CNF would loose the information of mutual exclusion, and result in a too coarse abstraction.

[11]For example, for six locations, i.e., $|S|=6$, $[s]$ consists of $\lceil\log_2(6)\rceil=3$ Boolean variables, with which we could represent $2^3=8>6$ locations.

This argumentation holds in the same way for the set of events of TA, since every transition in a TA is labelled with a distinct event. In TCA and TNA, however, a *set of* ports can be active on each transition, so no mutual exclusion constraint is needed.

3.3.5 Encoding of Transitions

For the encoding of transitions, there are a number possibilities of how to define the logical structure of the representation. For explanatory purposes, let $e \in E$ denote a transition from location s to location s'. Let $\mathtt{s_t}$ and $\mathtt{s'_{t+1}}$ be the representations of source location s (in step $\mathtt{t}$) and target location s' (in step $\mathtt{t+1}$), let $\varphi_{\mathtt{t}}$ denote the remaining constraints of step $\mathtt{t}$ ("preconditions of e", e.g., the representation $\mathtt{I(s)_t}$ of the invariant of s), and let $\varphi_{\mathtt{t+1}}$ denote the remaining constraints of step $\mathtt{t+1}$ ("postconditions of e"). Constraints involving variables of both $\mathtt{t}$ and $\mathtt{t+1}$ are also contained in $\varphi_{\mathtt{t+1}}$. The exact structure of $\varphi_{\mathtt{t}}$ and $\varphi_{\mathtt{t+1}}$ is irrelevant here, as we only use them to explain the conceptual idea of transition representation.

The two main conceptual alternatives one can think of for encoding the transition relation are

$$\bigwedge_{e \in E} (\mathtt{s_t} \wedge \varphi_{\mathtt{t}} \rightarrow \mathtt{s'_{t+1}} \wedge \varphi_{\mathtt{t+1}}) \text{ (or, equivalently, using } \leftarrow \text{) and} \tag{3.34}$$

$$\bigvee_{e \in E} (\mathtt{s_t} \wedge \varphi_{\mathtt{t}} \wedge \mathtt{s'_{t+1}} \wedge \varphi_{\mathtt{t+1}}) \tag{3.35}$$

Though (3.34) might seem more intuitive (it can be read as "if the preconditions are fulfilled, move to the next location"), and is in a way closer to CNF (after rewriting $\rightarrow$, note that (3.35) is actually in DNF), our encoding is based (3.35), cf. Section 3.1. The reason is that (3.34) cannot handle nondeterminism. To illustrate this, consider the real-time system part in Figure 3.8. If $(x \leq 2)$, both clock guards are satisfied, i.e., both transitions are enabled, and the next location is chosen nondeterministically. The representation (omitting irrelevant constraints) of

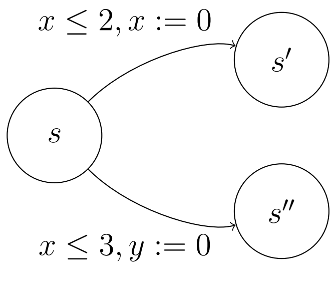


Figure 3.8: Representation of Transitions, Motivation

the two transitions in Figure 3.8, based on the two alternatives presented above, is given in (3.36) respectively (3.37).[12]

[12]Note that (3.37) does not entirely correspond to the representations defined in Section 3.1, but is used for illustration purposes only.

$$\begin{aligned}&(\mathtt{s_t}\wedge(\mathtt{z_t}-\mathtt{x_t}\leq 2) \rightarrow \mathtt{s'_t}\wedge(\mathtt{x_{t+1}}=\mathtt{z_{t+1}})\wedge(\mathtt{y_{t+1}}=\mathtt{y_t})\wedge(\mathtt{z_{t+1}}=\mathtt{z_t})) \wedge \\ &(\mathtt{s_t}\wedge(\mathtt{z_t}-\mathtt{x_t}\leq 3) \rightarrow \mathtt{s''_t}\wedge(\mathtt{y_{t+1}}=\mathtt{z_{t+1}})\wedge(\mathtt{x_{t+1}}=\mathtt{x_t})\wedge(\mathtt{z_{t+1}}=\mathtt{z_t}))\end{aligned} \tag{3.36}$$

$$\begin{aligned}&(\mathtt{s_t}\wedge(\mathtt{z_t}-\mathtt{x_t}\leq 2) \wedge \mathtt{s'_t}\wedge(\mathtt{x_{t+1}}=\mathtt{z_{t+1}})\wedge(\mathtt{y_{t+1}}=\mathtt{y_t})\wedge(\mathtt{z_{t+1}}=\mathtt{z_t})) \vee \\ &(\mathtt{s_t}\wedge(\mathtt{z_t}-\mathtt{x_t}\leq 3) \wedge \mathtt{s''_t}\wedge(\mathtt{y_{t+1}}=\mathtt{z_{t+1}})\wedge(\mathtt{x_{t+1}}=\mathtt{x_t})\wedge(\mathtt{z_{t+1}}=\mathtt{z_t}))\end{aligned} \tag{3.37}$$

The obvious problem in (3.36) is the fact that there exists no satisfying valuation in case $x\leq 2$: the left hand side of both implications is satisfied, but due to the mutual exclusion constraint in locations,[13] only one of the right hand sides can be satisfied; thus, no satisfying valuation exists.

A second problem with a representation according to (3.34) is the fact that logical implication allows the right hand side to be true, while the left hand side is false. For the transition relation, this means that it would be possible to find a satisfying valuation for the next step, even if there is no satisfying valuation for the current step.

3.4 Conclusion

In this Chapter, we have established the formal basis for extensive model checking and verification of the real-time systems presented in Chapter 2.

In Section 3.1, we have presented an encoding in propositional logic with linear rational arithmetic for each of the system models defined in Chapter 2. The representation of TA, as defined in Section 3.1.2, has essentially been presented in [KP07]. The representation of TCA, as defined in Section 3.1.3, is an extension of the work presented in [Kem11] (which in turn in based on [Kem09]). Since in Chapter 2, we have extended the formal model of TCA from [ABdBR07] (which was also used in [Kem11]) with memory cells, consequently this extension had to be included in the representation as well. Equivalently, we have extended the TNA model from [Kem10] with data values and memory cells in Section 2.4, and have extended the representation of TNA in Section 3.1.4 accordingly. For both TCA and TNA, we have sketched the difficulties that arise from memory cells being used in source and target location of the same transition in Section 3.1.1.5.

We have shown how to apply Bounded Model Checking to the representation of real-time systems in Section 3.2. The notion of unfolding (Section 3.2.2) and the concepts for representation of properties (Section 3.2.3) have been presented in previous work, and have been restated here for clarity and consistency. We have provided a discussion on completeness of BMC in Section 3.2.4, and a proof of correctness in Section 3.2.5. The latter in particular takes into account the new representation features regarding memory cells, data variables and data values.

Finally, in Section 3.3, we have provided an extensive discussion on other possibilities to represent real-time systems (using different encodings or variable types, for example), and we have motivated our design decisions.

[13]This problem would occur in the very same way for logarithmic encoding, since the current location is unique at any time.

Chapter 4

Abstraction Refinement

Abstraction refinement [CGJ+03, HJMM04] is a promising direction of research to overcome the challenges of the state explosion problem and infinite state model checking, while preserving correctness of verification results. The general abstraction refinement paradigm [CGJ+03] consists of three steps: (1) generate the initial abstraction, (2) model check the abstract system, and, if required, (3) refine the abstraction, and repeat.

The general idea of *abstraction* is to reduce the system complexity, by removing information which is considered irrelevant for the verification of a particular property, and therefore can be safely removed from the system. The obvious problem is how to determine which information is relevant: if the abstraction is too fine—i.e., removes too little information—verification still suffers from the state explosion problem. If, on the other hand, the abstraction is too coarse—i.e., removes too much information—verification suffers from too much information loss,[1] and the results are likely to be wrong. Consequently, abstraction techniques are distinguished based on how they deal with the information loss [CGJ+03].

Over-approximation techniques (also called *conservative techniques*), for example predicate abstraction [GS97], enrich the system behaviour by releasing constraints, such that correctness of the abstract system implies correctness of the concrete system. Over-approximation techniques admit *false negatives* (also called *spurious counterexamples*), i.e., a system behaviour which violates the property in the abstract system, but is not reproducible on the original (concrete) system. *Under-approximation techniques*, on the other hand, constrain the system behaviour by removing irrelevant parts, such that a property violation in the abstract system implies a property violation in the concrete system. Under-approximation techniques admit *false positives*, i.e., a system behaviour which satisfies the property in the abstract system, but violates it in the concrete system. In this work, we deal with

[1]By definition, abstraction *always* means information loss. Yet, with respect to the property to be verified, some information is irrelevant, and can be safely removed.

over-approximation techniques. For a more detailed description of abstraction techniques, see for example [CGP99, CGJ+03].

In the context of abstraction, the general idea of *refinement* can be described as "undo part of the abstraction". If a counterexample to the property under test has been detected in the abstract system (using over-approximation techniques, for under-approximation, the reasoning is different), it needs to be checked whether the counterexample comprises a violation of the property, or whether it is spurious. In the former case, the counterexample to the property is reproducible in the original system, which means the property does not hold in the original system. In the latter case, the counterexample is not reproducible in the original system, which means the spurious counterexample is due to wrong (too coarse) abstraction, and the abstract system needs to be refined. The refinement ideally takes into account the reason why the counterexample has been feasible in the abstract system, in order to prevent this erroneous behaviour in further verification steps.

The remainder of this Chapter is organised as follows: in Section 4.1, we describe our uniform abstraction methodology *abstraction by merging omission*. The methodology is flexible to operate on different system types, since it is defined based on syntactical categories of variables, and takes the constituents of the system under consideration that are to be abstracted as parameters. In Section 4.2, we explain how to translate a counterexample (to the property under test) obtained from the abstract system back into the concrete system. Section 4.3 gives a brief overview of Craig interpolation [Cra57], discusses the expressiveness of the generated interpolants, and how they can be used to derive information about the cause of a spurious counterexample. In the end, we show how to syntactically reorder the input problem (without changing the semantics) to increase the expressiveness of the interpolants. In Section 4.4, we show how to refine the abstraction in case it has turned out to be too coarse, and discuss heuristics on the application of different refinement options. We conclude the Chapter in Section 4.5.

4.1 Abstraction by Merging Omission

In this section, we present a simple and fast, but nevertheless powerful, uniform abstraction technique specifically tailored to work on logical formulas: *abstraction by merging omission (MO)* [KP07, Kem11]. By removing constraints which are considered irrelevant to a particular (safety) property, MO yields an over-approximation.

The basic idea of MO is to reduce the system complexity of a real-time system $\mathfrak{S}$, $\mathfrak{S}\in\{\mathfrak{A},\mathfrak{T}\}$ (i.e., $\mathfrak{S}$ is a TA or a TCA, for discussion of abstraction of TNA, please refer to Section 6.1), by decreasing the number of symbols in the formula representation $\varphi(\mathfrak{S})$, while retaining as much information as possible about the transition characteristics (the abstract formula is weaker than $\varphi(\mathfrak{S})$, though). MO is defined for formulas in negation normal form (NNF),[2] to which $\varphi(\mathfrak{S})$ can be easily

[2]A formula is defined to be in NNF if negation only appears in front of literals, and $\{\neg, \vee, \wedge\}$ are the only allowed Boolean connectives. In propositional logic, every formula can be transformed into an equivalent formula in NNF, by (1) replacing implications and equivalences by their definitions (i.e., using only $\{\neg, \vee, \wedge\}$), (2) using De Morgan's laws to push negation inside, and (3) eliminating double negations.

transformed. MO uniformly works on the different syntactical categories contained in $\varphi(\mathfrak{S})$: it merges Boolean variables, by mapping them to the same image according to a *map of merging* γ, and it removes rational variables and arithmetic constraints according to a *set of omission* $\mathcal{O}$.

The intended idea of the set of omission $\mathcal{O}$ is to contain constraints and parameters whose *removal* from $\varphi(\mathfrak{S})$ enriches the behaviour of the represented system $\mathfrak{S}$, i.e., whose removal enlarges the set of valuations satisfying the formula representation $\varphi(\mathfrak{S})$ of $\mathfrak{S}$. For example, removing a clock constraint from a transition enriches the behaviour, in that the transition is enabled more often. Note that this is only true for convex clock constraints, because with logical conjunction $\wedge$ as the only logical connective (cf. Definition 2.1.2), removing any of the conjuncts removes a subconstraint, and thus enlarges the set of satisfying valuations.

The intended idea of the map of merging γ is to contain mappings for variables whose *mergence* in $\varphi(\mathfrak{S})$ enriches the behaviour of $\mathfrak{S}$. For example, merging two locations enriches the behaviour, in that the combined location allows for additional runs containing in- and outgoing transitions of different underlying locations.

We allow merging for propositional variables only, and omission for rational variables and arithmetic constraints. We first introduce some notation.

Notation 4.1.1 (Variable Sets (cf. Section 3.1.1)). For any real-time system $\mathfrak{S}$, let $\mathtt{S}$ and $\mathtt{X}$ be the sets of variables representing locations and clocks, respectively. For a TA $\mathfrak{A}$, let Σ be the set of variables representing events. For a TCA $\mathfrak{T}$, let $\mathtt{P_A}$, $\mathtt{P_{DA}}$, $\mathtt{D_F}$ and $\mathtt{D_{CO}}$ be the sets of variables representing port activity variables, port data variables, data fullness variables and data content variables, respectively. All variable sets are understood to be without indices.

We lift the notations from Section 2.1 in the straightforward way to reason about representation variables rather than constituents of real-time systems. In particular:

- by $\mathcal{CC}(\mathtt{X})$ and $\mathtt{X}|_{\mathtt{cc}}$, we denote the set of clock constraints over clock variables in $\mathtt{X}$, and the set of clock variables that occur in a clock constraint $\mathtt{cc}\in\mathcal{CC}(\mathtt{X})$, respectively (cf. Definition 2.1.2)

- by $\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$, $\mathtt{P_{DA}}|_{\mathtt{dc}}$ and $\mathtt{D_{CO}}|_{\mathtt{dc}}$, we denote the set of data constraints over port data variables in $\mathtt{P_{DA}}$ and data content variables in $\mathtt{D_{CO}}$, the set of port data variables that occur in a data constraint $\mathtt{dc}\in\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$, and the set of data content variables that occur in a data constraint $\mathtt{dc}\in\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$, respectively (cf. Definition 2.1.7)

- by $\mathcal{CC}(\mathtt{X})|_{\mathfrak{S}}$, we denote the set of clock constraints over clock variables in $\mathtt{X}$ that occur in the formula representation of a real-time system $\mathfrak{S}$; by $\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})|_{\mathfrak{S}}$, we denote the set of data constraints over port data variables in $\mathtt{P_{DA}}$ and data content variables in $\mathtt{D_{CO}}$ that occur in the formula representation of $\mathfrak{S}$ (cf. Notations 2.2.3 and 2.3.4).

The exact nature of the map of merging γ and the set of omission $\mathcal{O}$ (i.e., to which system parts are γ and $\mathcal{O}$ applicable) depends on the the underlying system $\mathfrak{S}$. We now define these for TA and TCA separately.

Definition 4.1.2 (Map of Merging, Set of Omission for TA). Let $\mathfrak{A}$ be a TA, with formula representation $\varphi(\mathfrak{A})$, and variable sets as introduced in Notation 4.1.1, let $\mathtt{P}_{\mathfrak{A}}{=}(\mathtt{S}{\cup}\Sigma)$.

The *map of merging* $\gamma_{\mathfrak{A}}$ *of* $\mathfrak{A}$ is a total map $\gamma_{\mathfrak{A}}{:}\mathtt{P}_{\mathfrak{A}}{\to}(\mathtt{P}_{\mathfrak{A}}{\cup}\mathtt{P'}_{\mathfrak{A}})$, with $\mathtt{P'}_{\mathfrak{A}}$ some fresh set of propositional variables, and $\gamma_{\mathfrak{A}}(p){=}\gamma_{\mathfrak{A}}(p')$ only if $(p,p'{\in}\mathtt{S})$ or $(p,p'{\in}\Sigma)$. The *set of omission* $\mathcal{O}_{\mathfrak{A}}$ *of* $\mathfrak{A}$ is $\mathcal{O}_{\mathfrak{A}}{\subseteq}\mathtt{X}{\cup}\mathcal{CC}(\mathtt{X})$, where $\mathcal{CC}(\mathtt{X})$ only contains atomic formulas (i.e., which do not contain the logical operator $\wedge$, cf. Definition 2.1.2).

The map of merging $\gamma_{\mathfrak{A}}$ merges locations or actions, it is the identity for elements not intended to be merged. The additional constraint "only if $(p,p'{\in}\mathtt{S})$ or $(p,p'{\in}\Sigma)$" ensures that γ only maps variables to the same image if they are of the same conceptual type (a location cannot be not merged with an action). The set of omission $\mathcal{O}_{\mathfrak{A}}$ allows to remove clocks or single clock constraints. In the former case, the intended idea is to completely remove $\mathtt{x}$ from $\varphi(\mathfrak{A})$ (i.e., every occurrence of $\mathtt{x}$). For this reason, all clock constraints $\mathtt{cc}$ reasoning about $\mathtt{x}$, that means with $\mathtt{x}{\in}\mathtt{X}|_{\mathtt{cc}}$, need to be removed as well (even if $\mathtt{cc}$ itself is not contained in $\mathcal{O}_{\mathfrak{A}}$). The abstraction function (Definition 4.1.5) automatically takes care of this.

Definition 4.1.3 (Map of Merging, Set of Omission for TCA). Let $\mathfrak{T}$ be a TCA, with formula representation $\varphi(\mathfrak{T})$, and variable sets as introduced in Notation 4.1.1, let $\mathtt{P}_{\mathfrak{T}}{=}(\mathtt{S}{\cup}\mathtt{P_A}{\cup}\mathtt{D_F})$.

The *map of merging* $\gamma_{\mathfrak{T}}$ *of* $\mathfrak{T}$ is a total map $\gamma_{\mathfrak{T}}{:}\mathtt{P}_{\mathfrak{T}}{\to}(\mathtt{P}_{\mathfrak{T}}{\cup}\mathtt{P'}_{\mathfrak{T}})$, with $\mathtt{P'}_{\mathfrak{T}}$ some fresh set of propositional variables, and $\gamma_{\mathfrak{T}}(p){=}\gamma_{\mathfrak{T}}(p')$ only if $(p,p'{\in}\mathtt{S})$, or $(p,p'{\in}\mathtt{P_A})$, or $(p,p'{\in}\mathtt{D_F})$. The *set of omission* $\mathcal{O}_{\mathfrak{T}}$ *of* $\mathfrak{T}$ is $\mathcal{O}_{\mathfrak{T}}{\subseteq}\mathtt{X}{\cup}\mathcal{CC}(\mathtt{X}){\cup}\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$, where $\mathcal{CC}(\mathtt{X})$ and $\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$ do not contain compound formulas (i.e., do not contain logical operators $\wedge$ and $\neg$, cf. Definitions 2.1.2 and 2.1.7). In addition, if for some data constraint $\mathtt{dc}{\in}\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})|_{\mathfrak{T}}$, there exists $\mathtt{Dp}{\in}\mathtt{P_{DA}}|_{\mathtt{dc}}$ with $\gamma(\mathtt{p}){\neq}id$, or $\mathtt{Dd}{\in}\mathtt{D_{CO}}|_{\mathtt{dc}}$ with $\gamma(\mathtt{d}){\neq}id$ (where $\mathtt{p}$ and $\mathtt{Dp}$ are port activity and port data variable of the same port p, and $\mathtt{d}$ and $\mathtt{Dd}$ are data fullness and data content variable of the same data variable d, cf. Section 3.1.1.4), we require $\mathtt{dc}{\in}\mathcal{O}_{\mathfrak{T}}$.

The map of merging $\gamma_{\mathfrak{T}}$ allows to merge locations, ports (actually port activity variables) or memory cells (actually data fullness variables), again it is the identity for elements not intended to be merged, and can only merge variables of the same conceptual type. The set of omission $\mathcal{O}_{\mathfrak{T}}$ allows to remove clocks and single clock constraints as before, and in addition allows to remove single data constraints. As for TA, the abstraction function (Definition 4.1.5) automatically takes care of removing all clock constraints $\mathtt{cc}$ with $\mathtt{x}{\in}\mathtt{X}|_{\mathtt{cc}}$ for a clock $\mathtt{x}{\in}\mathcal{O}_{\mathfrak{T}}$, even if $\mathtt{cc}{\notin}\mathcal{O}_{\mathfrak{T}}$.

For ports intended to be merged, i.e., with $\gamma(\mathtt{p}){\neq}id$, the automatic removal of data constraints by the abstraction function does not work in the same way as it does for clocks. The reason is that for a clock $\mathtt{x}$ and a clock constraint $\mathtt{cc}$, a simple syntactic check $\mathtt{x}{\in}\mathtt{X}|_{\mathtt{cc}}$ is sufficient to determine whether $\mathtt{cc}$ needs to be removed or not (see above). For ports, on the other hand, the set $\mathtt{P}_{\mathfrak{T}}$ contains port activity variables, while data constraints contain port data variables. Therefore, we need to explicitly add all data constraints to $\mathcal{O}_{\mathfrak{T}}$ that reason about ports intended to be merged. This is done by the last condition in Definition 4.1.3. The same argumentation holds for data fullness and data content variables.

Note that it is indeed necessary to *completely remove* data constraints that reason about ports and memory cells intended to be merged. The naive approach of replacing port data variables respectively data content variables in a data constraint by their image under $\gamma_{\mathfrak{T}}$ does not work, and may result in unsatisfiable data constraints. As an example, suppose a transition with port set $\{p, q\}$ and data constraint $((p{=}1) \wedge (q{=}2))$, and suppose $\gamma_{\mathfrak{T}}(\mathtt{p}){=}\gamma_{\mathfrak{T}}(\mathtt{q}){=}\mathtt{r}$. Straightforward syntactic replacement of port data variables would yield a transition with port set $\{r\}$ and data constraint $((r{=}1) \wedge (r{=}2))$. While the original (unabstracted) data constraint is satisfiable, the abstract data constraint is not; such abstraction would not yield an over-approximation.

Notation 4.1.4 (Abstraction). Since MO works uniformly on the different syntactical categories, in the sequel, we omit indices $\mathfrak{A}$ respectively $\mathfrak{T}$, and write γ, $\mathcal{O}$, $\mathtt{P}$ and $\mathtt{P}'$ only.

We use the term *domain of the abstraction*, denoted by ${}^{\bullet}\alpha$,[3] to refer to the set of parameters intended to be abstracted. That is, the domain of the abstraction consists of all elements in $\mathcal{O}$, and of those elements in $\mathtt{P}$ where γ is not the identity, that means which are mapped to an element in $\mathtt{P}'$. Formally, ${}^{\bullet}\alpha = \mathcal{O} \cup \gamma^{-1}(\mathtt{P}')$. Note that ${}^{\bullet}\alpha$ contains both (single) variables and (arithmetic) constraints.

We can now define the abstraction function.

Definition 4.1.5 (Abstraction by Merging Omission). Let $\mathfrak{S}$ be a real-time system, with $\varphi(\mathfrak{S})$ in NNF, and γ, $\mathcal{O}$, $\mathtt{P}$ and $\mathtt{P}'$ defined in Definitions 4.1.2 respectively 4.1.3.

The *abstraction of* $\varphi(\mathfrak{S})$ *(by merging omission) with respect to* $\mathcal{O}$ *and* γ, denoted as $\alpha_{\mathcal{O},\gamma}(\varphi(\mathfrak{S}))$, is defined in (4.4). We may write $\alpha(\varphi(\mathfrak{S}))$ if $\mathcal{O}$ and γ are clear form the context.

MO uniformly captures abstraction on all syntactic categories contained in $\varphi(\mathfrak{S})$: variables and constraints (cf. Section 3.2.1 for the definition of *Conts*$(\cdot)$) not meant to be abstracted (i.e., which are not contained in the domain of the abstraction ${}^{\bullet}\alpha$) are kept unchanged (4.1a). The map γ is applied to all positive propositional variables (4.1b). For negative propositional variables $\mathtt{p}$ (i.e., which occur as $\neg\mathtt{p}$ in $\varphi(\mathfrak{S})$) meant to be abstracted, we distinguish two cases: if there exists a positive propositional variable $\mathtt{p}'$ in the same conjunction as $\mathtt{p}$,[4] and with the same image under γ (i.e., $\mathtt{p}$ and $\mathtt{p}'$ are to be merged), we replace $\mathtt{p}$ with its positive image under γ (4.1c). The idea is that positive propositional variables are used to describe the "behaviour"—source and target of a transition, for example—while negative propositional variables are used to ensure consistency—mutual location exclusion,

[3]This notation anticipates Definition 4.1.5, where we use the symbol α to denote the abstraction function.

[4]*In the same conjunction* means enclosed by the same pair of parenthesis (note that some parenthesis can be omitted though, cf. Notation 2.1.1). For example, in $(\mathtt{p} \wedge \mathtt{p}' \wedge \mathtt{p}'') \vee \mathtt{p}'''$, $\mathtt{p}$, $\mathtt{p}'$ and $\mathtt{p}''$ are in the same conjunction, but $\mathtt{p}'''$ is not.

$$\alpha'(L) = \begin{cases} L & \mathit{Conts}(L) \cap {}^\bullet\alpha{=}\emptyset \quad (4.1a) \\ \gamma(L) & \mathit{Conts}(L) \cap {}^\bullet\alpha{\neq}\emptyset, L{=}\mathtt{p}{\in}\mathtt{P} \quad (4.1b) \\ \gamma(L) & \mathit{Conts}(L) \cap {}^\bullet\alpha{\neq}\emptyset, L{=}\neg\mathtt{p}, \mathtt{p}{\in}\mathtt{P}, \quad (4.1c) \\ & \exists\mathtt{p}'{\in}\mathtt{P} \text{ in the same conjunction as } \mathtt{p} : \gamma(\mathtt{p}){=}\gamma(\mathtt{p}') \\ \neg\gamma(L) & \mathit{Conts}(L) \cap {}^\bullet\alpha{\neq}\emptyset, L{=}\neg\mathtt{p}, \mathtt{p}{\in}\mathtt{P}, \quad (4.1d) \\ & \neg\exists\mathtt{p}'{\in}\mathtt{P} \text{ in the same conjunction as } \mathtt{p} : \gamma(\mathtt{p}){=}\gamma(\mathtt{p}'), \\ & \exists\neg\mathtt{p}'', \mathtt{p}''{\in}\mathtt{P}, \text{ in the same conjunction as } \mathtt{p} : \gamma(\mathtt{p}){=}\gamma(\mathtt{p}'') \\ \mathtt{true} & \text{otherwise} \quad (4.1e) \end{cases}$$

$$\alpha'(F \wedge G) = \alpha'(F) \wedge \alpha'(G) \tag{4.2a}$$
$$\alpha'(F \vee G) = \alpha'(F) \vee \alpha'(G) \tag{4.2b}$$
$$\gamma_\alpha(\mathtt{P}) = \bigwedge_{p\in\gamma(\mathtt{P})\setminus\mathtt{P}} \Big(\big(\big(\bigwedge_{p'\in\gamma^{-1}(p)} \neg p'\big) \vee p\big) \wedge \big(\bigvee_{p'\in\gamma^{-1}(p)} p' \vee \neg p\big)\Big) \tag{4.3}$$
$$\alpha(\varphi(\mathfrak{S})) = \alpha'(\varphi(\mathfrak{S})) \wedge \gamma_\alpha \tag{4.4}$$

Here, F and G are formulas in NNF, and L is a literal.

Figure 4.1: Abstraction by merging omission

for example. Therefore, if such $\mathtt{p}'$ exists, we can dismiss the literal $\neg\mathtt{p}$, since $\mathtt{p}$ and $\mathtt{p}'$ are mapped to the same image under γ, and we do not need the consistency constraint (for consistency between $\mathtt{p}$ and $\mathtt{p}'$) anymore. Note that replacing $\neg\mathtt{p}$ by `true` is possible as well, but this would yield a much coarser abstraction. If no such $\mathtt{p}'$ exists, but instead there exists a negative propositional variable $\mathtt{p}''$ with the same image under γ (4.1d), then we replace $\mathtt{p}$ and $\mathtt{p}''$ by their negative image under γ. Again, replacing $\neg\mathtt{p}$ and $\neg\mathtt{p}'$ by `true` is possible but would yield a much coarser abstraction. In all other cases, α' maps the literal to `true` (4.1e). In particular, this last case handles arithmetic constraints (both positive and negative) meant to be abstracted. Remember that arithmetic constraints not meant to be abstracted are handled in (4.1a) already.

In this way, α' performs a quick variant of existential abstraction [CGJ+03], while exploiting the syntactic categories and structural relationships of elemens in our formula representation.

In order to guarantee that MO yields an over-approximation, we need to keep track of the relation between symbols in $\mathtt{P}$ and their abstract counterparts in $\mathtt{P}'$. For this reason, we add the constraint γ_α (4.3) to the abstract formula (note that (4.3) is in NNF, and is equivalent to $\bigwedge_{p\in\gamma(\mathtt{P})\setminus\mathtt{P}} \big((\bigvee_{p'\in\gamma^{-1}(p)} p') \leftrightarrow p\big)\big)$.

As mentioned above after Definitions 4.1.2 and 4.1.3, the abstraction function automatically removes clock constraints $\mathtt{cc}$ over clocks $\mathtt{x}{\in}{}^\bullet\alpha$ (i.e., with $\mathtt{x}{\in}\mathtt{X}|_{\mathtt{cc}}$), even if $\mathtt{cc}{\notin}{}^\bullet\alpha$. The reason is that the only applicable rule in (4.1) for literal $\mathtt{cc}$ is (4.1e). Rule (4.1a) is not applicable, since the intersection $\mathit{Conts}(\mathtt{cc}){\cap}{}^\bullet\alpha$ is not empty, and all other rules only handle propositional variables. Therefore, constraint $\mathtt{cc}$ is replaced by `true` by (4.1e).

In contrast, for ports intended to be abstracted (and equivalently for memory cells), the situation is different: a data constraint `dc` over port p contains the port data variable $\mathtt{Dp}{\in}\mathtt{P}_{\mathtt{DA}}$ (cf. Section 3.1.1), while the domain of the abstraction ${}^{\bullet}\alpha$ contains the port activity variable $\mathtt{p}{\in}\mathtt{P}_{\mathtt{A}}$. As a consequence, the intersection $\mathit{Conts}(\mathtt{dc}){\cap}{}^{\bullet}\alpha$ in (4.1a) would be empty. To correctly abstract from the data constraint, we explicitly add `dc` to ${}^{\bullet}\alpha$ (cf. Definition 4.1.3), such that the intersection is not empty anymore.

Notation 4.1.6 (Abstraction). Without confusion, in the sequel we use the symbol α only, and omit the symbol α'. For example, for a literal L, we write $\alpha(L)$ instead of $\alpha'(L)$.

We get the following results.

Lemma 4.1.7 (Abstraction by Weakening). Abstraction by merging omission, as defined in Definition 4.1.5, yields an over-approximation, that means $\alpha(F)$ is weaker than F in the sense that the implication $F{\to}\alpha(F)$ is *valid* (true in all models).

Proof. The proof can be found in Section A.2 in the Appendix, on Page 138. □

Theorem 4.1.8 (Correctness of Abstraction). Abstraction by merging omission, as defined in Definition 4.1.5, yields a correct over-approximation on sets of runs.

Proof. The proof can be found in Section A.2 in the Appendix, on Page 146. □

So far, we have assumed the variables to be without indices. Lifting α to the presence of localisations is straightforward: γ and $\mathcal{O}$ are understood oblivious to indices in the NNF of $\varphi(\mathfrak{S})$, such that indices directly carry over to $\varphi(\mathfrak{S})_k$ unchanged. Defining different abstractions for different steps is possible using the same definition of α, but we consider it to be less useful. Note that α is homomorphic with respect to $\{\wedge,\vee\}$, which proves the equality of $\alpha(\varphi(\mathfrak{S})_k)$ and $\alpha(\varphi(\mathfrak{S}))_k$ (except for speed of computing the abstraction, where $\alpha(\varphi(\mathfrak{S}))_k$ is superior).

Example 4.1.9 (Abstraction). Consider again the formula representation $\varphi(\mathfrak{A})$ of the intelligent light switch, as shown in Table 3.3 in Example 3.1.2. According to Definition 4.1.2, we have $\mathtt{P}{=}(\mathtt{S}{\cup}\Sigma){=}\{\mathtt{off},\mathtt{light},\mathtt{bright},\mathtt{press},\tau\}$. For merging locations `light` and `bright` into one location `on`, we define $\mathcal{O}{=}\emptyset$, $\mathtt{P}'{=}\{\mathit{on}\}$, $\gamma(\mathtt{light}){=}\gamma(\mathtt{bright}){=}\mathtt{on}$, and $\gamma(\mathtt{p}){=}\mathit{id}$ for $\mathtt{p}{\in}\mathtt{P}\backslash\{\mathtt{light},\mathtt{bright}\}$.

The abstraction by merging omission of $\varphi(\mathfrak{A})$ with respect to γ and $\mathcal{O}$, $\alpha(\varphi(\mathfrak{A}))$, is shown in Table 4.2.

Note that we have simplified the formulas, by removing redundant parts. For example, after applying the abstraction function α, the first entry in Table 4.2

$$
\begin{aligned}
\alpha(\varphi^{init}(\mathfrak{A})) &= \mathtt{off}_0\wedge\neg\mathtt{on}_0\wedge\neg\mathtt{press}_0\wedge\neg\tau_0\wedge(\mathtt{z}_0{=}0)\wedge(\mathtt{x}_0{=}0)\\
\alpha(\varphi^{action}(e_1)) &= \mathtt{off}_\mathtt{t}\wedge\mathtt{press}_{\mathtt{t+1}}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}})\wedge(\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{z}_{\mathtt{t+1}})\wedge\mathtt{on}_{\mathtt{t+1}}\\
\alpha(\varphi^{action}(e_2)) &= \mathtt{on}_\mathtt{t}\wedge\mathtt{press}_{\mathtt{t+1}}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}})\wedge(\mathtt{z}_\mathtt{t}-\mathtt{x}_\mathtt{t}{>}3)\wedge(\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{off}_{\mathtt{t+1}}\\
\alpha(\varphi^{action}(e_3)) &= \mathtt{on}_\mathtt{t}\wedge\mathtt{press}_{\mathtt{t+1}}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}})\wedge(\mathtt{z}_\mathtt{t}-\mathtt{x}_\mathtt{t}{\leq}3)\wedge(\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{on}_{\mathtt{t+1}}\\
\alpha(\varphi^{action}(e_4)) &= \mathtt{on}_\mathtt{t}\wedge\mathtt{press}_{\mathtt{t+1}}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}})\wedge(\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{off}_{\mathtt{t+1}}\\
\alpha(\varphi^{delay}(\mathit{off})) &= \mathtt{off}_\mathtt{t}\wedge\neg\mathtt{press}_{\mathtt{t+1}}\wedge\neg\tau_{\mathtt{t+1}}\wedge(\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_{\mathtt{t+1}})\wedge(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_{\mathtt{t+1}})\wedge\mathtt{off}_{\mathtt{t+1}}\\
\alpha(\varphi^{delay}(\mathit{light})) &= \mathtt{on}_\mathtt{t}\wedge\neg\mathtt{press}_{\mathtt{t+1}}\wedge\neg\tau_{\mathtt{t+1}}\wedge(\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_{\mathtt{t+1}})\wedge(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_{\mathtt{t+1}})\wedge\mathtt{on}_{\mathtt{t+1}}\\
&= \alpha(\varphi^{delay}(\mathit{bright}))\\
\alpha(\varphi^{trans}(\mathfrak{A})) &= \alpha(\varphi^{action}(e_1))\vee\alpha(\varphi^{action}(e_2))\vee\alpha(\varphi^{action}(e_3))\vee\alpha(\varphi^{action}(e_4))\vee\\
&\quad\ \alpha(\varphi^{delay}(\mathit{off}))\vee\alpha(\varphi^{delay}(\mathit{light}))\\
\alpha(\varphi^{location}(\mathfrak{A})) &= (\mathtt{off}_{\mathtt{t+1}}\wedge\neg\mathtt{on}_{\mathtt{t+1}})\vee(\mathtt{on}_{\mathtt{t+1}}\wedge\neg\mathtt{off}_{\mathtt{t+1}})\\
\alpha(\varphi^{mutex}(\mathfrak{A})) &= (\mathtt{press}_{\mathtt{t+1}}\wedge\neg\tau_{\mathtt{t+1}})\vee(\tau_{\mathtt{t+1}}\wedge\neg\mathtt{press}_{\mathtt{t+1}})\vee(\neg\mathtt{press}_{\mathtt{t+1}}\wedge\neg\tau_{\mathtt{t+1}})\\
\gamma_\alpha(\mathtt{P}) &= ((\neg\mathtt{light}_\mathtt{t}\wedge\neg\mathtt{bright}_\mathtt{t})\vee\mathtt{on}_\mathtt{t})\wedge(\mathtt{light}_\mathtt{t}\vee\mathtt{bright}_\mathtt{t}\vee\neg\mathtt{on}_\mathtt{t})\\
\alpha(\varphi(\mathfrak{A})) &= \alpha(\varphi^{init}(\mathfrak{A}))\wedge\alpha(\varphi^{trans})\wedge\alpha(\varphi^{location}(\mathfrak{A}))\wedge\alpha(\varphi^{mutex}(\mathfrak{A}))\wedge\gamma_\alpha(\mathtt{P})
\end{aligned}
$$

Table 4.2: Abstraction by Merging Omission: Example

contains conjunct $\neg\mathtt{on}_0$ twice, resulting from applying γ to $\neg\mathtt{bright}_0$ and $\neg\mathtt{light}_0$ in the original formula (cf. Table 3.3).

4.2 Concretisation

In the previous Section, we have defined abstraction by merging omission on the formula representation $\varphi(\mathfrak{S})$ of a real-time system $\mathfrak{S}$. We have explained how to obtain the abstract formula $\alpha(\varphi(\mathfrak{S}))$ from $\varphi(\mathfrak{S})$, by removing parameters that are considered irrelevant for the verification of a particular safety property.

The general problem with abstraction is that typically, it is not clear up front what is the set of relevant parameters that need to be kept in order to preserve correctness of verification results. It can therefore happen that one or more parameters from this set of relevant parameters are removed by the abstraction. If this happens, we get *false negatives*: a false negative is a "proof" that the property is violated in the abstract system, while in the original (concrete) system it is not (cf. the explanations at the beginning of this Chapter).

Thus, if a counterexample to the property under test has been found in the abstract system, we need to check whether this counterexample is real (i.e., corresponds to a true violation of the property, also in the original system) or spurious (i.e., is due to wrong abstraction, and the abstraction needs to be refined). To this end, the abstract counterexample is translated back into the original system, to determine whether it is reproducible there. This is called *concretisation.* In our context, concretisation works as follows.

First recall that to check whether a property expressed by formula ϕ holds in the abstract system $\alpha(\varphi(\mathfrak{S}))$, we check the conjunction $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi$ for satisfiability (cf. Section 3.2.3). If the formula is satisfiable, this indicates that the property does not hold in the abstract system, and every model σ of $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi$ corresponds to a run of the abstract system which violates the property.

Next, observe that every model σ of $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi$ is a total assignment to the variables in $\mathit{Vars}(\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi)$, but only a partial assignment to the variables in $\mathit{Vars}(\varphi(\mathfrak{S})_k \wedge \neg\phi)$, that means some variables in $\varphi(\mathfrak{S})_k$ are left unconstrained by σ. This reflects the fact that one run in the abstract system can correspond to a set of runs in the original system.

Concretisation now consists in trying to extend the model σ of $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi$ to a model of $\varphi(\mathfrak{S})_k \wedge \neg\phi$, that agrees with σ on variables in $\mathit{Vars}(\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi)$. This is done by interpreting σ as a conjunction of variable assignments:

$$\rho_\sigma = \bigwedge_{\substack{\mathtt{v} \in \mathit{Vars}(\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi) \\ \sigma(\mathtt{v}) = v}} (\mathtt{v} = v), \tag{4.5}$$

and trying to find a model for the conjunction $\varphi(\mathfrak{S})_k \wedge \neg\phi \wedge \rho_\sigma$. We call ρ_σ the *witness run (for ϕ)*. As the witness run is highly restrictive (it singles out only one abstract path), the abstract counterexample guides the search through the concrete system with a very narrow focus and is highly efficient. If such a model for $\varphi(\mathfrak{S})_k \wedge \neg\phi \wedge \rho_\sigma$ exists, that means, if the witness run is concretisable, we have found a run in the original system that violates the property. If no such model exists, i.e, $\varphi(\mathfrak{S})_k \wedge \neg\phi \wedge \rho_\sigma$ is unsatisfiable, the counterexample is spurious, and the abstraction needs to be refined.

Remark 4.2.1 (Concretisation). Since $\sigma \models (\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi)$ (i.e., σ is a model of $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi$), in particular $\sigma \models \neg\phi$. By construction of ρ_σ, we have $\models \rho_\sigma \to \neg\phi$. For every valuation σ', we thus have $\sigma' \models \varphi(\mathfrak{S})_k \wedge \neg\phi \wedge \rho_\sigma$ iff $\sigma' \models \varphi(\mathfrak{S})_k \wedge \rho_\sigma$. For this reason, we can safely remove the conjunct $\neg\phi$ from the formula $\varphi(\mathfrak{S})_k \wedge \neg\phi \wedge \rho_\sigma$ during concretisation, and check the satisfiability of $\varphi(\mathfrak{S})_k \wedge \rho_\sigma$ only, without affecting the results. We will use this fact for interpolation.

In the next Section, we describe how to use Craig interpolants to derive information about which parameters need to be refined.

4.3 Interpolation

In this section, we introduce Craig interpolants (Section 4.3.1), and discuss their expressive power in the context of SAT-based verification (Section 4.3.2). We do *not* show how to derive/compute interpolants, since powerful tools exist for this task. The interested reader is referred to [McM03, McM05b], for example.

4.3.1 Craig Interpolants

Craig interpolants have been defined in [Cra57]. They are defined for pairs of inconsistent formulas (i.e., formulas which are unsatisfiable together), and provide a way of capturing the reason of inconsistency.

Definition 4.3.1 (Craig Interpolant). Let (F_1, F_2) be a pair of inconsistent formulas, i.e., with $\models \neg(F_1 \wedge F_2)$. A *Craig interpolant for* F_1 *and* F_2 (or simply *interpolant*) is a formula G such that

$$\models F_1 \rightarrow G, \tag{4.6}$$
$$\models G \rightarrow \neg F_2,^{5} \text{ and} \tag{4.7}$$
$$\mathit{Vars}(G) \subseteq \mathit{Vars}(F_1) \cap \mathit{Vars}(F_2). \tag{4.8}$$

Here, $\mathit{Vars}(\varphi)$ denotes the set of variables in a formula φ. F_1 is called *prefix of* G, and F_2 is called *suffix of* G.

For a pair of inconsistent formulas, an interpolant always exists [Cra57], and can be derived from a resolution proof of inconsistency of F_1 and F_2 in time linear in the size of the proof [Pud97, McM03, McM05b]. Originally [Cra57], interpolants were defined for purely propositional formulas, but it has been shown that they can be derived in the same way for formulas with linear (in)equalities over rational numbers, see [McM04] for details. Interpolants are not unique: for a pair of inconsistent formulas, typically many formulas exist which fulfil the conditions in Definition 4.3.1. Furthermore, resolution proofs are not unique either, that means the same pair of formulas can have different resolution proofs of inconsistency, depending on for example to which subformulas the resolution rule is applied first.

An interpolant G for an inconsistent pair of formulas (F_1, F_2) captures the reason of inconsistency as follows: G is always an over-approximation of the prefix F_1 (4.6), that means every model of F_1 is also a model of G. In this way, G captures the facts that can be derived from the prefix, that means the maximal set of facts that holds for every valuation of the prefix. Moreover, G is also an under-approximation of the negated suffix $\neg F_2$ (4.7), that means every model of G is also a model of $\neg F_2$. In this way, G captures facts that are inconsistent with F_2, that means the minimal set of facts that can never be extended to a satisfying valuation of the suffix. Since the interpolant contains only common symbols of prefix and suffix (4.8), it captures the cause of inconsistency, in that the constraints expressed by the interpolant are sufficient to show the inconsistency of prefix and suffix.

Interpolants are defined for pairs of formulas. If we have a single unsatisfiable formula F that is a conjunction of two subformulas, i.e., $F{=}F_1 \wedge F_2$, we can split F around this top-level conjunction, and derive an interpolant for the resulting pair (F_1, F_2). If F is a conjunction of more than two subformulas, i.e., $F{=}F_1 \wedge \ldots \wedge F_n$, for $n{>}2$, we can interpret F as a sequence of formulas $F_1, \ldots, F_n$, and derive an interpolant for every possible nonempty bisection (i.e., both subsequences contain at least one formula each) of the sequence. This corresponds to deriving an interpolant for every possible split around a top-level conjunction of F. In this way, we get a sequence of $n{-}1$ interpolants $G_1, \ldots, G_{n-1}$, such that G_i is an interpolant for the pair $(F_1 \wedge \ldots \wedge F_i, F_{i+1} \wedge \ldots \wedge F_n)$, $i{\in}\{1, \ldots, n{-}1\}$. In [McM05a], the author showed

[5]The notion $\models \neg(G \wedge F_2)$ is also commonly found in the literature, but in this context, we consider our (equivalent) notion $\models G \rightarrow \neg F_2$ more comprehensible.

that if the sequence of interpolants $G_1, \ldots, G_{n-1}$ is derived from *the same* refutation proof for the inconsistency of $F_1, \ldots, F_n$, then

$$\models (G_i \wedge F_{i+1}) \rightarrow G_{i+1} \tag{4.9}$$

holds for every $i \in \{1, \ldots, n-1\}$.

There exist a number of tools that support interpolant generation for SMT formulas, like for example CSIsat [BZM08, csi], FOCI [FOC] or MathSAT [mat]. The former only supports interpolant generation for a pair of formulas. The latter two support the approach just described: when called on a sequence of n inconsistent formulas, they generate a sequence of $n-1$ interpolants with the property (4.9).

4.3.2 Expressiveness of Interpolants

In this section, we discuss what kind of information about the cause of unsatisfiability can be derived from interpolants, and we show how the sequential order of formulas can influence the generated interpolants.

For a pair of inconsistent formulas (F_1, F_2), we can derive a single interpolant G. Without further taking into account the concrete structure of prefix F_1 and suffix F_2, we can derive the following information about the inconsistency of F_1 and F_2 from G:

- G=`true`: we do not get any information about the prefix, since $F_1 \rightarrow$ `true` (4.6) holds for every F_1. We know that the suffix by itself is inconsistent, since `true` $\rightarrow \neg F_2$ (4.7) only evaluates to `true` if F_2 evaluates to `false`
- G=`false`: we know that the prefix by itself is inconsistent, since $F_1 \rightarrow$ `false` (4.6) only evaluates to `true` if F_1 evaluates to `false`. We do not get any information about the suffix, since `false` $\rightarrow F_2$ (4.7) holds for every F_2.
- $G=F$ for some formula F, $F\neq$`true`, $F\neq$`false`: we do not get any information about prefix or suffix alone, but we know that the conjunction is unsatisfiable. As explained above, the constraints expressed by F are sufficient to prove the inconsistency of F_1 and F_2.

Obviously, changing the sequential order of F_1 and F_2— that means deriving an interpolant G' for the pair (F_2, F_1)—does not yield additional information about the cause of unsatisfiability. In particular, observe that if G is an interpolant for (F_1, F_2), then it follows directly from Definition 4.3.1 that $\neg G$ is an interpolant for (F_2, F_1). However, when deriving a sequence of $n-1$ interpolants from a sequence of n formulas, the sequential order of the formulas has a considerable influence on the generated interpolants, and thus on their expressiveness, as the following example shows.

Example 4.3.2 (Sequential Formula Order and Expressiveness of Interpolants). Consider two sequential orders for a set of inconsistent formulas:

$$(a \leftrightarrow b), (a \leftrightarrow c), (b \leftrightarrow e), (e \leftrightarrow f), (c \leftrightarrow d), (a \leftrightarrow \neg b), \text{ and} \tag{4.10}$$

$$(c \leftrightarrow d), (a \leftrightarrow c), (a \leftrightarrow b), (a \leftrightarrow \neg b), (b \leftrightarrow e), (e \leftrightarrow f). \tag{4.11}$$

The inconsistency is solely caused by the two formulas $(a \leftrightarrow b)$ and $(a \leftrightarrow \neg b)$. If we apply the SMT solver MathSAT to each of the two sequences, we get the interpolant sequences shown in Table 4.3 (for (4.10) on the left, for (4.11) on the right).[6]

formulas	interpolants	formulas	interpolants
$(a \leftrightarrow b)$		$(c \leftrightarrow d)$	
	$(a \leftrightarrow b)$		`true`
$(a \leftrightarrow c)$		$(a \leftrightarrow c)$	
	$(a \leftrightarrow b) \wedge (a \leftrightarrow c)$		`true`
$(b \leftrightarrow e)$		$(a \leftrightarrow b)$	
	$(a \leftrightarrow b) \wedge (a \leftrightarrow c) \wedge (b \leftrightarrow e)$		$(a \leftrightarrow b)$
$(e \leftrightarrow f)$		$(a \leftrightarrow \neg b)$	
	$(a \leftrightarrow b)$		`false`
$(c \leftrightarrow d)$		$(b \leftrightarrow e)$	
	$(a \leftrightarrow b)$		`false`
$(a \leftrightarrow \neg b)$		$(e \leftrightarrow f)$	

Table 4.3: Interpolant sequences for different formula orders

Notice that the interpolants derived for sequence (4.11) (on the right side of Table 4.3) are much simpler than the interpolants derived for (4.10). In particular, there is only one interpolant $\notin \{\texttt{true}, \texttt{false}\}$ on the right side. This is due to the fact that the formulas in the sequence (4.11) are arranged in such a way that the number of common variables of prefix and suffix is minimised. As a result, the cause of unsatisfiability becomes clear immediately from the interpolants derived for (4.11): the third interpolant $(a \leftrightarrow b)$ (the only interpolant $\notin \{\texttt{true}, \texttt{false}\}$) precisely describes the constraints that cause the inconsistency.

The interpolants derived for sequence (4.10) (on the left side of Table 4.3), on the other hand, are more complex,[7] because the number of common variables of prefix and suffix is larger. As a result, the cause of unsatisfiability is not immediately clear (though the repeated occurrence of interpolant $(a \leftrightarrow b)$ gives an indication already, but this is in general not the case for larger formula sequences and larger numbers of common variables).

The example has shown that reducing the number of common variables of prefix and suffix helps to obtain more expressive interpolants: in the best case, a number of interpolants simplify to `true` or `false`, which allows to reduce the sequence of formulas to an inconsistent subsequence. Moreover, less common variables of prefix and suffix—and thus less variables that can be contained in the interpolant—yield less complex invariants, which in turn capture the reason of inconsistency more precisely. To illustrate this last statement, consider the third interpolant $(a \leftrightarrow b) \wedge (a \leftrightarrow c) \wedge (b \leftrightarrow e)$

[6]Read the Table as follows: for each interpolant, the prefix of the interpolant consists of all formulas above it, and the suffix consists of all formulas below it. For example, for the second interpolant on the left, $(a \leftrightarrow b) \wedge (a \leftrightarrow c)$, the prefix is $(a \leftrightarrow b) \wedge (a \leftrightarrow c)$, and the suffix is $(b \leftrightarrow e) \wedge (e \leftrightarrow f) \wedge (c \leftrightarrow d) \wedge (a \leftrightarrow \neg b)$.

[7]More complex in the sense that they involve more variables, and have more satisfying interpretations.

derived for (4.10) (left side of Table 4.3), and the third interpolant $(a{\leftrightarrow}b)$ derived for (4.11) (right side of Table 4.3). While the latter precisely describes the constraints causing the inconsistency of (4.11) (as explained above), the former contains the superfluous conjuncts $(a{\leftrightarrow}c)$ and $(b{\leftrightarrow}e)$.

Note that the argumentation for interpolants `true` and `false` about the unsatisfiability of prefix and suffix directly carries over from pairs of formulas. That is, the suffix of an interpolant `true` (which is now a set of formulas) is inconsistent by itself, and so is the prefix of an interpolant `false`. For example, for the second interpolant `true` on the right side of Table 4.3, the suffix $(a{\leftrightarrow}b)\wedge(a{\leftrightarrow}\neg b)\wedge(b{\leftrightarrow}e)\wedge(e{\leftrightarrow}f)$ is inconsistent by itself. We will come back to this fact in the next section.

4.3.3 Sequential Formula Order for $\varphi(\mathfrak{S})$

We now present a sequential formula order for the subformulas of $\varphi(\mathfrak{S})$ that takes into account the considerations from the previous Section.

Remember that we need to refine the abstraction if the witness run ρ_σ represents a spurious counterexample, that means if the conjunction $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi$ (of abstract system and property) is satisfiable, but the conjunction $\varphi(\mathfrak{S})_k\wedge\rho_\sigma$ (of original system and witness run) is not, cf. Section 4.2 and in particular Remark 4.2.1. To find the parameters that were wrongly abstracted, we derive a sequence of interpolants for $\varphi(\mathfrak{S})_k\wedge\rho_\sigma$, and from these determine the ill-abstracted parameters. To increase the expressiveness of the interpolants, we take into account the results from the previous section, by syntactically reordering the subformulas of $\varphi(\mathfrak{S})_k\wedge\rho_\sigma$ (without changing the semantics), such that the number of common variables is minimised. Intuitively, the resulting sequence results from "interleaving" elements from $\varphi(\mathfrak{S})_k$ and ρ_σ, based on their unfolding depth. In detail, we construct the sequence as follows.

First, we split $\varphi(\mathfrak{S})_k$ around top-level conjunctions, based on the partition in (3.29), and reconjunct elements where all variables have the same unfolding depth, resulting in the following set

$$\begin{aligned}\{\varphi^{init}(\mathfrak{S}),\varphi^{trans}(\mathfrak{S})_{(\mathtt{0})},\varphi^{location}(\mathfrak{S})_{(\mathtt{1})}\wedge\varphi^{mutex}(\mathfrak{S})_{(\mathtt{1})},\ldots\\ \ldots,\varphi^{trans}(\mathfrak{S})_{(\mathtt{k-1})},\varphi^{location}(\mathfrak{S})_{(\mathtt{k})}\wedge\varphi^{mutex}(\mathfrak{S})_{(\mathtt{k})}\}\end{aligned}\tag{4.12}$$

Next, we do the same for the witness run: we split ρ_σ around the conjunctions (cf. (4.5)), and reconjunct elements (variable assignments) with the same unfolding depth, which yields the set

$$\{\bigwedge_{\substack{\mathtt{v_0}\in Vars(\rho_\sigma)\\ \sigma(\mathtt{v_0})=v}}(\mathtt{v_0}{=}v),\ldots,\bigwedge_{\substack{\mathtt{v_k}\in Vars(\rho_\sigma)\\ \sigma(\mathtt{v_k})=v}}(\mathtt{v_k}{=}v)\}\tag{4.13}$$

Finally, we join the two sets, conjunct elements with the same unfolding depth, and stratify (i.e., sort by unfolding depth) the formulas. The result is the following sequence of formulas

$$
\begin{aligned}
&\varphi^{init}(\mathfrak{S})\wedge \bigwedge_{\substack{\mathtt{v_0}\in \mathit{Vars}(\rho_\sigma)\\ \sigma(\mathtt{v_0})=v}} (\mathtt{v_0}{=}v), \qquad (4.14)\\
&\varphi^{trans}(\mathfrak{S})_{(\mathtt{0})},\\
&\varphi^{location}(\mathfrak{S})_{(\mathtt{1})}\wedge\varphi^{mutex}(\mathfrak{S})_{(\mathtt{1})}\wedge \bigwedge_{\substack{\mathtt{v_1}\in \mathit{Vars}(\rho_\sigma)\\ \sigma(\mathtt{v_1})=v}} (\mathtt{v_1}{=}v),\\
&\varphi^{trans}(\mathfrak{S})_{(\mathtt{1})},\dots,\\
&\varphi^{trans}(\mathfrak{S})_{(\mathtt{k-1})},\\
&\varphi^{location}(\mathfrak{S})_{(\mathtt{k})}\wedge\varphi^{mutex}(\mathfrak{S})_{(\mathtt{k})}\wedge \bigwedge_{\substack{\mathtt{v_k}\in \mathit{Vars}(\rho_\sigma)\\ \sigma(\mathtt{v_k})=v}} (\mathtt{v_k}{=}v)
\end{aligned}
$$

In the sequel, without confusion we may use $\varphi(\mathfrak{S})_k\wedge\rho_\sigma$ to refer to the "reordered" variant (4.14). For example, by "the sequence of interpolants derived for $\varphi(\mathfrak{S})_k\wedge\rho_\sigma$" (in Section 4.4, for example), we actually mean to the sequence of interpolants derived for (4.14).

Reordering the subformulas of $\varphi(\mathfrak{S})_k\wedge\rho_\sigma$ in this way has two major advantages. The first advantage is that we have minimised the number of common as much as possible: elements of the sequence alternately contain variables of one respectively two unfolding depths,[8] and due to the stratification, every two subsequent elements of the sequence share variables of exactly one unfolding depth. In this way, every interpolant derived for any bisection of the sequence into prefix and suffix can contain variables of (at most, cf. interpolants `true` and `false`) one unfolding depth. Minimising the number of common variables has the advantages illustrated in the previous Section.

The second advantage is that prefixes and suffixes of any interpolant directly correspond to prefixes and suffixes of the witness run: for any interpolant G_i, with $1\leq i\leq 2k$,[9] a model of the prefix of G_i directly corresponds to a prefix of length $\lfloor\frac{i}{2}\rfloor$ of the run through $\mathfrak{S}$ represented by the witness run ρ_σ. Again, this is due to the fact that we stratified the formulas in (4.14). If $G_i{=}\mathtt{false}$, we can thus conclude that the prefix of length $\lfloor\frac{i}{2}\rfloor$ of the witness run, which is represented by those elements of (4.13) with unfolding depth smaller or equal to $\lfloor\frac{i-1}{2}\rfloor$, is not concretisable. Equivalently, if $G_i{=}\mathtt{true}$, we can conclude that the suffix of length $\lceil\frac{2k-i}{2}\rceil$ of the witness run, which is represented by those elements of (4.13) with unfolding depth greater or equal to $\lceil\frac{i}{2}\rceil$, is not concretisable.

Remark 4.3.3. To illustrate the bounds in the above explanations (for example, to illustrate that the prefix of interpolant G_i indeed corresponds to a prefix of the witness run of length $\lfloor\frac{i}{2}\rfloor$), consider the following example. For unfolding depth

[8] In (4.14), odd-numbered elements contain variables of one unfolding depth. For example, the first element $\varphi^{init}(\mathfrak{S})\wedge\bigwedge_{\mathtt{v_0}\in \mathit{Vars}(\rho_\sigma),\sigma(\mathtt{v_0})=v}(\mathtt{v_0}{=}v)$ contains variables of unfolding depth 0 only. Even-numbered elements contain variables of two unfolding depths. For example, the second element $\varphi^{trans}(\mathfrak{S})_{(\mathtt{0})}$ contains variables of unfolding depths 0 and 1.

[9] Observe that for unfolding depth $\mathtt{k}$, the sequence (4.14) has $2k{+}1$ elements, which allows to derive a sequence $G_1,\dots,G_{2k}$ of $2k$ interpolants.

3, the sequence (4.14) has 2∗3+1 = 7 elements, for which we can derive 2∗3 = 6 interpolants $G_1, \ldots, G_6$:

$$
\begin{array}{ll}
\varphi^{init}(\mathfrak{S}) \wedge \bigwedge_{\mathtt{v}_0 \in Vars(\rho_\sigma), \sigma(\mathtt{v}_0)=v} (\mathtt{v}_0 = v) & \\
 & G_1 \\
\varphi^{trans}(\mathfrak{S})_{(0)} & \\
 & G_2 \\
\varphi^{location}(\mathfrak{S})_{(1)} \wedge \varphi^{mutex}(\mathfrak{S})_{(1)} \wedge \bigwedge_{\mathtt{v}_1 \in Vars(\rho_\sigma), \sigma(\mathtt{v}_1)=v} (\mathtt{v}_1 = v) & \\
 & G_3 \\
\varphi^{trans}(\mathfrak{S})_{(1)} & \\
 & G_4 \\
\varphi^{location}(\mathfrak{S})_{(2)} \wedge \varphi^{mutex}(\mathfrak{S})_{(2)} \wedge \bigwedge_{\mathtt{v}_2 \in Vars(\rho_\sigma), \sigma(\mathtt{v}_2)=v} (\mathtt{v}_2 = v) & \\
 & G_5 \\
\varphi^{trans}(\mathfrak{S})_{(2)} & \\
 & G_6 \\
\varphi^{location}(\mathfrak{S})_{(3)} \wedge \varphi^{mutex}(\mathfrak{S})_{(3)} \wedge \bigwedge_{\mathtt{v}_3 \in Vars(\rho_\sigma), \sigma(\mathtt{v}_3)=v} (\mathtt{v}_3 = v) &
\end{array}
$$

Consider the case i=3: the prefix of interpolant G_3 consists of three formulas, which correspond to a run of length $\lfloor \frac{3}{2} \rfloor = 1$, and this run is represented by the variable valuations (i.e., the elements of (4.13)) with unfolding depths smaller or equal to $\lfloor \frac{3-1}{2} \rfloor = 1$. The suffix of G_3 consists of four formulas, which correspond to a run of length $\lceil \frac{6-3}{2} \rceil = 2$, and this run is represented by the variable valuations with unfolding depths greater or equal to $\lceil \frac{3}{2} \rceil = 2$.

4.4 Refinement

If a counterexample to the property under test has been detected in the abstract system, and this counterexample has turned out to be spurious in the concretisation step, the abstraction is too coarse and needs to be refined.

Most refinement approaches are based on information obtained from the spurious counterexample; the most well-known technique is called *counterexample-guided abstraction refinement (CEGAR)* [CGJ+03]. In CEGAR, one spurious abstract counterexample (corresponding to a set of runs in the concrete system, cf. Section 4.2) is ruled out within every refinement step, that means the abstraction is modified in such a way that the spurious counterexample is not feasible anymore.

We propose a variant of CEGAR, by defining two possibilities of refining the abstraction, both based on information obtained from the spurious counterexample.

4.4.1 Ruling Out a Counterexample Trace

The first refinement option is to rule out the spurious counterexample just found. The spurious counterexample is given by the set of valuations that constitute the witness run ρ_σ, cf. (4.5). To ensure that the behaviour expressed by the witness run is not feasible anymore in future verification steps, we could simply add $\neg\rho_\sigma$ as an additional conjunct to the representation of the abstract system $\alpha(\varphi(\mathfrak{S}))_k$, and, instead of checking $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\phi$ (cf. Section 4.2), model check $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\rho_\sigma \wedge \neg\phi$ in the next verification step. This is essentially equivalent to basic CEGAR.

Yet, we can do better, by taking into account information obtained from the interpolants. Let $G_1, \ldots, G_n$ be the sequence of interpolants derived for $\varphi(\mathfrak{S})_k \wedge \rho_\sigma$, let G_f, $1 \leq f \leq n$, be the first interpolant in the sequence which is equal to `false`, and let G_t, $1 \leq t \leq n$, be the last interpolant in the sequence which is equal to `true` (note that G_t and G_f do not necessarily exist).

If G_f exists, we know (cf. Definition 4.3.1 and Section 4.3.3) that the prefix of G_f is unconcretisable. Let i be the unfolding depth of variables contained in the interpolant G_{f-1}, which is the interpolant directly preceding G_f (remember that every interpolant contains variables of at most one unfolding depth). We reduce the witness run ρ_σ of length k to a subset $\rho_{\sigma_{\leq i}}$ of length i containing only variable valuations of variables with unfolding depth i or smaller, and model check $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\rho_{\sigma_{\leq i}} \wedge \neg\phi$ in the next verification step. In this way, in a single refinement step, we not only rule out *one* abstract counterexample, as is done in basic CEGAR, but we rule out *a set* of abstract counterexamples at once.

If G_t exists, we can use this interpolant in a similar way: let j be the unfolding depth of variables contained in the interpolant G_{t+1}, which is the interpolant directly following G_t. We reduce the witness run ρ_σ of length k to a subset $\rho_{\sigma_{\geq j+1}}$ of length $k-j+1$ containing only variable valuations of variables with unfolding depth $j+1$ or larger, and model check $\alpha(\varphi(\mathfrak{S}))_k \wedge \neg\rho_{\sigma_{\geq j+1}} \wedge \neg\phi$ in the next verification step.

If neither G_f nor G_t exists, or if $G_t = G_{f-1}$ (the latter typically only happens with small toy examples), this refinement step is not applicable. If both G_f and G_t exists, and $G_t \neq G_{f-1}$, we choose the interpolant that leads to a shorter counterexample fragment being ruled out, that is, we choose G_f if $i < k-j+1$ (with i, j as above), and G_t otherwise. The underlying idea is that a shorter witness run fragment corresponds to a larger set of witness runs of the full length k, and thus a larger set of counterexamples is ruled out at once.

4.4.2 Refining a Previously Abstracted Parameter

The second refinement option is to reduce the domain of the abstraction ${}^\bullet\alpha$, by removing a parameter from it (that means, either remove a parameter from $\mathcal{O}$, or remove a mapping from γ). In this way, the parameter is put back into the abstract system, such that it is considered again in future verification steps. More concretely, the current abstraction $\alpha_{\mathcal{O},\gamma}$ is transformed into a new abstraction $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}$ with ${}^\bullet\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}} \subset {}^\bullet\alpha_{\mathcal{O},\gamma}$, where either $\widetilde{\mathcal{O}} \subset \mathcal{O}$, or $\widetilde{\gamma}^{-1}(\mathtt{P}') \subset \gamma^{-1}(\mathtt{P}')$. The new abstraction $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}$ is computed as follows.

To refine a parameter $\mathtt{e} \in \mathcal{O}$ (a clock, a clock constraint or a data constraint), the parameter is simply removed from $\mathcal{O}$, that means $\widetilde{\mathcal{O}} = \mathcal{O} \setminus \mathtt{e}$. However, it is only possible to refine a clock constraint `cc` over a clock $\mathtt{x} \in \mathtt{X}|_{\mathtt{cc}}$ if `x` itself is not abstracted, that means $\mathtt{x} \notin {}^\bullet\alpha$, since otherwise, the new abstraction $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}$ would produce a system that contains a clock constraint over an undefined clock. Equivalently, it is only possible to refine a data constraint `dc` over over a port `p` with $\mathtt{Dp} \in \mathtt{P}_{\mathtt{DA}}|_{\mathtt{dc}}$, or over a memory cell `m` with $\mathtt{Dm} \in \mathtt{D}_{\mathtt{CO}}|_{\mathtt{dc}}$ if `p` and `m` themselves are not abstracted, that means $\mathtt{m} \notin {}^\bullet\alpha$, and $\mathtt{p} \notin {}^\bullet\alpha$.

If, instead, a parameter $\mathtt{e}{\in}\gamma^{-1}(\mathtt{P}')$ (a location, an event, a port or a memory cell) is to be refined, the mapping $\{\mathtt{e}{\mapsto}\mathtt{e}'\}$, with $\mathtt{e}'{\in}\mathtt{P}'$, is removed from γ and replaced by the identity mapping. If $\mathtt{e}$ was merged with one other parameter $\tilde{\mathtt{e}}$ only, that means there exists exactly one $\tilde{\mathtt{e}}{\in}\mathtt{P}$ with $\gamma(\tilde{\mathtt{e}}){=}\gamma(\mathtt{e}){=}\mathtt{e}'$, then the mapping for $\tilde{\mathtt{e}}$ is removed as well: $\widetilde{\gamma}{=}\gamma\oplus\{\mathtt{e}{\mapsto}\mathtt{e}, \tilde{\mathtt{e}}{\mapsto}\tilde{\mathtt{e}}\}$, where $\oplus$ denotes function overriding.[10] Otherwise, that means if $|\gamma^{-1}(\gamma(\mathtt{e}))| > 2$, no further changes need to be made to γ: $\widetilde{\gamma}{=}\gamma\oplus\{\mathtt{e}{\mapsto}\mathtt{e}\}$.

To solve the question which parameters from ${}^\bullet\alpha$ are to be considered for refinement, that means to identify the ill-abstracted parameters, we consider the set $\mathit{Conts}(G_f){\cap}{}^\bullet\alpha \overset{\text{def}}{=} {}^\bullet\alpha|_{G_{f-1}}$, with G_{f-1} as before, of elements which are potentially responsible for the spurious counterexample. Alternatively, we can consider the set ${}^\bullet\alpha|_{G_{t+1}}$, with G_{t+1} as before. After choosing one of the parameters from any of the sets, we refine it according to the procedure explained above. In the next verification step, the property is checked on the abstract system computed with the new abstraction function $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}$, that means model checking is applied to $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}(\varphi(\mathfrak{S}))_k{\wedge}\neg\phi$.

Note that we could actually take any interpolant for this refinement option, that means consider any set ${}^\bullet\alpha|_{G_i}$, with $1{\le}i{\le}n$, since by definition, every interpolant captures the reason of inconsistency of its prefix and suffix (cf. Definition 4.3.1 and Section 4.3.2). However, since $\mathit{Conts}(G_f) = \mathit{Conts}(G_t) = \emptyset$, choosing either G_f or G_t is not reasonable. Choosing any interpolant $G_{f'}$, with $f'{>}f$ (that means, which occurs in the sequence $G_1, \ldots, G_n$ at some point after G_f), is not reasonable either: the prefix of G_f is inconsistent by itself, that means the cause of unsatisfiability is entirely contained in this prefix. An interpolant $G_{f'}$, whose prefix contains the prefix of G_f, can therefore not provide more information about the cause of unsatisfiability. A similar argumentation holds for interpolants $G_{t'}$, with $t'{<}t$, which occur in the sequence $G_1, \ldots, G_n$ at some point before G_t. Note that in SMT solver tools like [csi, FOC, mat], such interpolants $G_{f'}$ and $G_{t'}$ are simplified to `false` respectively `true`, even if the solver derived an interpolant $\neq$`false` respectively $\neq$`true` for the particular bisection.

Let $F_1, \ldots, F_{n+1}$ be the sequence of formulas in $\alpha(\varphi(\mathfrak{S}))_k{\wedge}\neg\phi$ for which the interpolants $G_1, \ldots, G_n$ are derived. The reason why we choose G_{f-1} to determine candidates for refinement is the following: we know that the prefix of G_f (the set of formulas $F_1, \ldots, F_f$) is unsatisfiable, but the prefix of G_{f-1} (the set of formulas $F_1, \ldots, F_{f-1}$) is not. Thus, the addition of "suffix" F_f to the prefix $F_1, \ldots, F_{f-1}$ causes the whole sequence $F_1, \ldots, F_f$ to be unsatisfiable, and interpolant G_f describes the cause of this unsatisfiability (cf. also (4.9)). The argumentation for choosing G_{t+1} is similar.

4.4.3 Refinement Heuristics

For conciseness of explanation, we refer to the refinement option presented in Section 4.4.1 as the *first* (refinement) option, and to the refinement option presented in

[10]The mapping for $\tilde{\mathtt{e}}$ could actually remain in γ, since a single mapping $\{\tilde{\mathtt{e}}{\mapsto}\mathtt{e}'\}$, with $|\gamma^{-1}(\gamma(\tilde{\mathtt{e}}))| = 1$, corresponds to pure syntactic replacement of $\tilde{\mathtt{e}}$ by $\mathtt{e}'$, and therefore does not change the satisfiability of the result. Yet, $\tilde{\mathtt{e}}$ is a parameter in the original system $\varphi(\mathfrak{S})$, while $\mathtt{e}'$ is not, therefore, we prefer to remove $\mathtt{e}'$.

Section 4.4.2 as the *second* (refinement) option.

The problem with the second refinement option is *which* parameter to choose from $^\bullet\alpha|_{G_{f-1}}$ (respectively from $^\bullet\alpha|_{G_{t+1}}$) for refinement: since interpolants are not unique, typically not all parameters in the set are responsible for the present spurious counterexample, and some of the parameters might not be ill-abstracted at all. As an example, consider again the interpolants in the left column of Table 4.3: the second and the third interpolant contain parameters b and c, respectively b, c and e, which are completely irrelevant to the cause of inconsistency of the formulas in (4.10). Additionally, it is in general not clear under which conditions to apply either the first or the second refinement option. Thus, the remaining difficulty is to define heuristics describing the application of the two refinement options. Finding adequate heuristics is a problem common to almost all refinement approaches, cf. for example [CGJ+03, CCK+02].

While the first refinement option is quick and easy to apply, in that the application is straightforward once a counterexample has been found, it cannot yield results as long as essential parameters are inadequately abstracted. Application of the second option is not so straightforward and in addition involves more computational effort (recomputing the abstraction), but this option is indispensable to add ill-abstracted parameters back into the system. However, if the second option is applied too frequently, the abstract system quickly collapses to the original system. It is thus necessary to define heuristics that strike a suitable balance between options one and two.

The best solution to solving this problem is to leave the decision (which option and—for option two—which parameter to choose) to the human experts. The reasoning is that system developers who have designed and improved the system in a number of iterations in the development process have a deep insight into the functioning of the system, and, when presented with a counterexample and/or a set of potentially responsible parameters, these experts will be able to deduce information about the quality/applicability of the different refinement options.

To support the developers in reaching a decision, and as a first step towards automatic abstraction refinement, we propose the following heuristic, which in particular uses both refinement options one and two. We first rule a fixed number of counterexamples (for example $\frac{k}{2}$), using option one, and in every iteration record the set $^\bullet\alpha|_{G_{f-1}}$ (respectively $^\bullet\alpha|_{G_{t+1}}$) of parameters that are potentially responsible. Let $^\bullet\alpha|_{G^i_{f-1}}$ denote the set obtained in the i-th iteration. After this, we inspect the multiset $^\bullet\alpha|_{G^1_{f-1}} \cup {}^\bullet\alpha|_{G^2_{f-1}} \cup \ldots \cup {}^\bullet\alpha|_{G^n_{f-1}}$, and determine the parameter(s) with the highest multiplicity. If there exists a single such parameter, we refine it with refinement option two. Otherwise, if there are two or more parameters with the same (highest) multiplicity, we can either randomly choose a parameter among these, or continue applying option one until one parameter in the multiset has a higher multiplicity than the others.

The idea of this heuristic is as follows: as explained above, applying refinement option one is quick and easy, moreover, it will never result in an abstraction that is too fine (in contrast, when refining a parameter with option two that is irrelevant to the property, the abstraction becomes unnecessarily fine). We can therefore apply option one a number of times, without loosing efficiency. By taking into account the set of potentially responsible parameters ${}^{\bullet}\alpha|_{G_{f-1}}$ obtained from (many) different iterations, we increase the probability of choosing a parameter that is indeed ill-abstracted, since ill-abstracted parameters will occur more frequently.

4.5 Conclusion

In this Chapter, we have presented a general framework for abstraction and refinement of TA and TCA that are represented in propositional logic with linear arithmetic.

In Section 4.1, we have defined our uniform abstraction function. It is essentially equivalent to the abstraction function presented in [Kem11] (which in turn is an improved variant of the abstraction functions presented in [KP07, Kem09]). The major difference between the abstraction function presented here and the one in [KP07, Kem09] is the fact that we do *not* in general map negative propositional variables to `true`. Such an abstraction function would effectively remove the mutual exclusion constraint for locations (cf. (3.5), (3.14) and (3.24)) and TA events (3.6), as well as part of the consistency constraint on data values in TCA and TNA ((3.15) and (3.25)). Instead, we take into account the special characteristics of our formula representation, and the syntactic context of the propositional variable (4.1c), (4.1d), and in this way retain more information. See Section A.2 for further details.

In addition, while the abstraction functions presented in previous work were tailored to one system type each, we have shown in Section 4.1 that the abstraction function presented here uniformly handles both TA and TCA. We have provided correctness results, which in particular take into account the new representation features of memory cells and data values. In Section 4.2, we have restated in more detail the concept of concretisation, which has been briefly sketched in previous work [KP07, Kem11].

Section 4.3 deals with Craig interpolation. After a brief introduction to Craig interpolants, presented mainly for completeness, we have provided an extensive discussion on expressiveness of interpolants, and the type of information that can be derived from (a sequence of) interpolants. We use these results in Section 4.3.3 to reorder the formulas in the representation of TA and TCA such that the information derived from interpolants is maximised.

Finally, in Section 4.4, we have discussed two refinement options in detail. While these options were sketched in previous work already ([KP07, Kem11]), we here provide a detailed discussion of how and when to apply each of the options, and discuss advantages and disadvantages. Section 4.4.3 discusses refinement heuristics based on the two refinement options. Apart from automatic abstraction refinement, these heuristics can also be used to support user decisions on which refinement option to choose.

Chapter 5

Tool Development and Application to Case Studies

In the previous Chapters, we have presented the theoretical foundations of a framework for modelling and analysing distributed real-time systems. While it is of course very important to have a well-grounded and correct theoretical basis to start from, any formalism can only be used in practice if it comes along with adequate tool support. Such tool support can in turn be used to show the usability and applicability of a formalism. In particular, graphical editors offer intuitive and easy access to a new formalism, even for the unexperienced user. In this Chapter, we present our work on tool development, which has been done in the context of this thesis.

The rest of this chapter is organised as follows: in Section 5.1, we present the details of the tool implementation. We start by giving a brief introduction to the *Extensible Coordination Tools* (ECT) in Section 5.1.1, a tool suite that is being developed in the Foundations of Software Engineering group[1] (SEN3) at CWI. In the remainder of Section 5.1, We then present the support for modelling and analysis of TCA that we have integrated into ECT . In Section 5.2, we use a small example to explain the most important features of our tool, and show the typical workflow when using it. Finally, in Section 5.3, we introduce two case studies, present experimental results when using our tool to model check them, and discuss the advantages of using our formalism and tool over using other formalisms.

Except for Sections 5.1.1 (which is presented to provide a deep insight into the functioning of our implementation in the overall context of ECT) and 5.3.1 (which has been presented in [Kem11]), this Chapter describes original work, which has not been presented elsewhere.

[1]`http://www.cwi.nl/research-groups/Foundations-of-software-engineering`

5.1 Implementation

We have implemented our work on TCA and integrated it as part of the *Extensible Coordination Tools* (ECT, [ECT]), building on the *Extensible Automata (EA) framework*. In the next section (Section 5.1.1), we give a high-level overview of ECT, and briefly introduce the architecture of the EA framework. In Section 5.1.2, we present the implementation of the TCA editor plugin. Finally, Section 5.1.3 explains the implementation of the formula generation from TCA. If not explicitly mentioned, all implementation is done in *Java* [GJSB05].

5.1.1 The Extensible Automata framework in ECT

ECT is an integrated graphical development environment available for the Eclipse [Ecl] platform. It consists of a set of plugins that support modelling and analysis of component-based systems. The core of ECT has been developed and implemented in the context of the PhD thesis of Christian Krause [Kra11]. ECT provides extensive support for systems which are specified in the channel-based coordination language $\mathcal{R}$eo [Arb04], including a graphical modelling environment (editor), conversion to and from other models (for example to quantitative intentional automata (QIA) and (in turn) markov chains [AMM$^+$09], to CA [ABRS04], to mCRL2 [KKdV10, Kra11], from BPMN, UML sequence diagrams and BPEL [CKA10]), java code generation, and animation. In addition, ECT comprises plugins to directly edit most of the aforementioned models, amongst other for CA, QIA and mCRL2 (for BPMN, an Eclipse plugin outside ECT already exists [BPM]). Please refer to [Kra11, ECT] for a complete and detailed description of ECT.

Our implementation of the TCA plugin uses the *Extensible Automata* (EA) framework in ECT. The framework was developed and implemented with the intention to "provide a unified framework for deriving automata-based models for $\mathcal{R}$eo" [Kra11], but allows to extend existing and define new automata-based models in general (i.e., detached from $\mathcal{R}$eo). In the remainder of this Section, we give a brief overview of the EA framework; again, we refer to [Kra11] for a complete and detailed description.

The meta model of the framework comprises the packages *cwi.ea.automata* (cf. Figure 5.1) and *cwi.ea.extensions* (cf. Figure 5.2).[2] Conceptually, there are two different types of elements in the EA framework: extensible elements and extensions (extending elements). The interfaces *IExtensible* in *cwi.ea.automata* and *IExtension* in *cwi.ea.extensions* mirror this structure. Every *IExtensible* owns a number of *IExtension*s.

Package *cwi.ea.automata* contains classes for the basic extensible elements that make up every automata-based model: Automaton, State,[3] and Transition. These extend the abstract base class *ExtensibleElement* (in package *cwi.ea.extensions*), which implements interface *IExtensible*.

[2]Figures 5.1 and 5.2 are essentially taken from [Kra11]. Equivalent diagrams can also be found as part of the source code of the ECT tools, publicly available at `http://reo.project.cwi.nl/`.

[3]Our notion of *location* (cf. Chapter 2) equates to the notion of *state* in the EA framework.

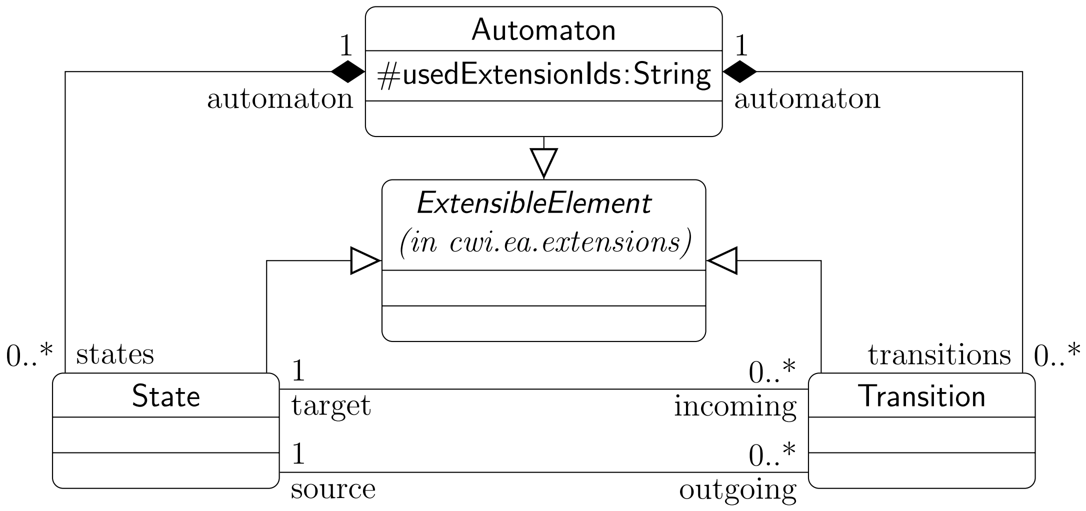


Figure 5.1: Package *cwi.ea.automata*

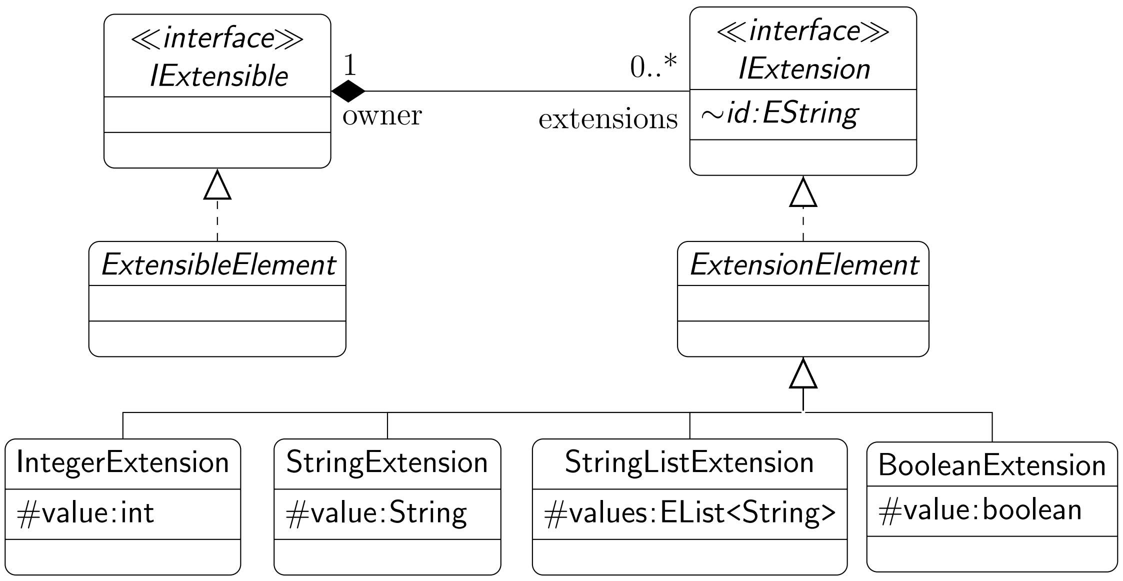


Figure 5.2: Package *cwi.ea.extensions*

Extensions are key/value pairs, where the key is a unique ID of type String, and the value contains the content information of the extension. There are four predefined extension value types in *cwi.ea.extensions*: StringExtension, StringListExtension, BooleanExtension and IntegerExtension. These extend the abstract base class *ExtensionElement*, which implements *IExtension*. New custom types can be easily defined by extending *ExtensionElement* or one of its subclasses.

To make clear the difference between *plugin* and *extension*: a plugin is an encapsulation of behaviour, providing support for a certain feature. For example, the TCA plugin provides support for modelling and analysing TCA in various ways. A plugin typically comprises several extensions. Every extension implements one aspect of the feature, for example, modelling TCA transitions by adding real-time aspects to transitions. Every extension is enabled for (i.e., applicable to) exactly one of the extensible elements Automaton, State or Transition.

Every extension has a provider class (*extension provider*), cf. Figure 5.3. An extension provider has to implement the interface *IExtensionProvider*, and one of the interfaces *ITextualExtensionProvider* (for textual extensions, i.e. labels) or *ICustomExtensionProvider* (for other types of extensions), all contained in package *cwi.ea*. The provider class defines how to handle the extension in the editor, it contains methods amongst others for parsing, editing and validating (syntax and semantic checks) the extension, or for providing a default extension. For conciseness of explanation, in the sequel, we refer to extensions by the name of their provider class, while omitting the suffix Provider.

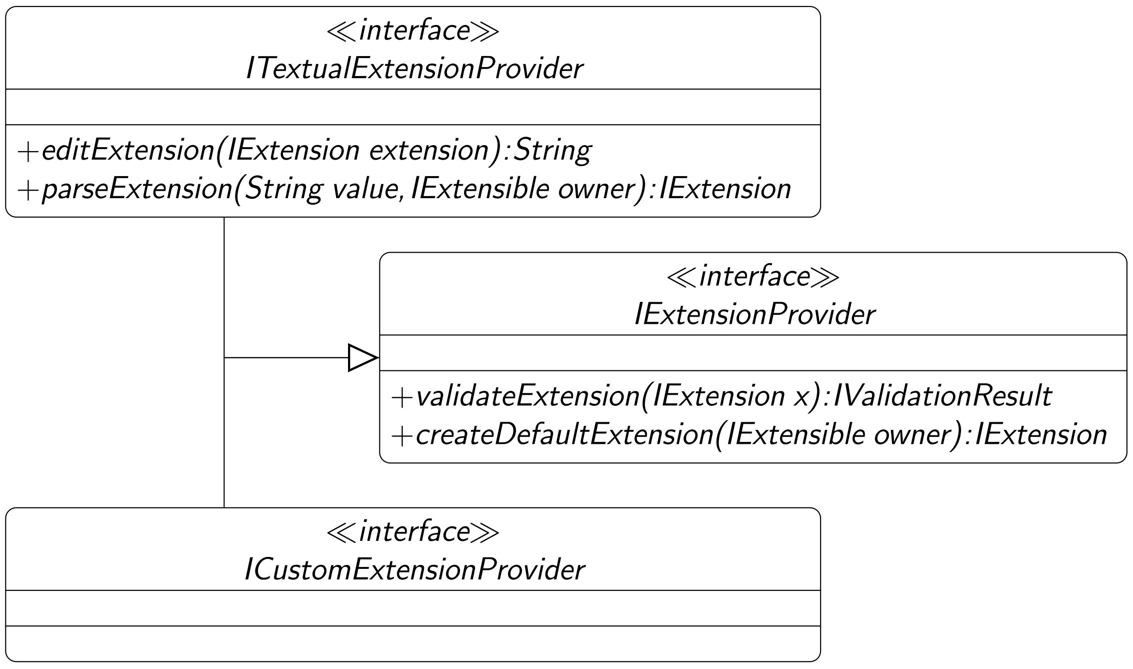

≪interface≫
ITextualExtensionProvider
+editExtension(IExtension extension):String
+parseExtension(String value, IExtensible owner):IExtension
≪interface≫
IExtensionProvider
+validateExtension(IExtension x):IValidationResult
+createDefaultExtension(IExtensible owner):IExtension
≪interface≫
ICustomExtensionProvider

Figure 5.3: Package *cwi.ea*

An EA automaton can in principle use any combination of the extensions defined in plugins within the EA framework (see [Kra11] for a complete overview of supported extensions). In the manifest file `plugin.xml` that accompanies every plugin definition in Eclipse, it is possible to define dependencies and mutual exclusion constraints among extensions (either among extensions of the particular plugin, or among extensions of the particular plugin and other plugins). Moreover, in the `plugin.xml` files of EA plugins, it is possible to define an automaton type, that has a predefined set of extensions enabled on creation.

5.1.2 The Timed Constraint Automaton Plugin

We have added support for TCA to the EA framework. Since TCA are an extension of CA, we were able to use the existing extensions for initial locations (StartStateExtension), port names on automaton level (AutomatonPortNames), active ports (TransitionPortNames), and location memory (StateMemoryExtension). To support the particular features of TCA, we implemented a number of new extensions. All of the new extensions implement interface *ITextualExtensionProvider* (cf. Figure 5.3), and are simple enough such that we were able to use the predefined extension value type StringExtension (cf. Figure 5.2). All extensions are contained in package *cwi.ea.extensions.clocks*. The following overview shows the new extensions, their format as seen in the editor, and their most important features. The default value is generated when calling *createDefaultExtension()* , the syntactic checks are performed when calling *validateExtension* (both from *IExtensionProvider*).

AutomatonClocks (all clocks defined for a TCA): comma separated list of clock names. Enabled for Automaton.[4] Default value "" (empty string, i.e., no clocks).

StateInvariant (location invariants): clock constraint formula according to Definition 2.1.2. Enabled for State. Default value `true`. Syntactic check that formula is well-formed, and that every clock used in the formula is defined in AutomatonClocks.

TransitionGuard (clock guards on transitions): clock constraint formula according to Definition 2.1.2. Enabled for Transition. Default value `true`. Syntactic check that formula is well-formed, and that every clock used in the formula is defined in AutomatonClocks.

TransitionUpdate (clock updates on transitions): comma separated list of clock assignments according to Definition 2.1.5.[5] Enabled for Transition. Default value "" (i.e., all clocks keep their value). Syntactic check that no clock is assigned more than once, and that every clock used in an assignment is defined in AutomatonClocks.

TCADataConstraints (data guards on transitions): data constraint formula according to Definition 2.1.7. Enabled for Transition. Default value `true`. Syntactic check that every port used in the data constraint is defined in Transi-

[4]See above, every extension is enabled for exactly one of Automaton, State, Transition.

[5]More concretely, write `x`=n for an update $\lambda(x)=n$, and `x`=`y` for an update $\lambda(x)=y$. Clocks not mentioned are assumed to keep their value.

tionPortNames, and every memory cell used in the data constraint is defined in StateMemoryExtension of either source or target location of the transition.

Despite the fact that a data constraint extension for CA already existed (ConstraintExtension), we had to provide the new extension TCADataConstraints for data guards in TCA. The reason is that both types of data constraints provide distinct features, which are not supported by the other data constraint type. For example, CA data constraints allow to reason about functions on the data values, which is not supported in TCA. On the other hand, CA data constraints require that every memory cell used by the target location of the transition is initialised in the data constraint, while for TCA, we do not impose this restriction (cf. Remark 2.3.2).

In the `plugin.xml` file, we defined a number of constraints on (in)admissible and required combinations of extensions, as shown in Table 5.4. We also defined a new automaton type *Timed Constraint Automaton*. When a new automaton of this type is created in the editor, it has the aforementioned extensions (except of course for ConstraintExtension) enabled.

Extension	Requires Extension	Mutually exclusive with
TransitionGuard	AutomatonClocks, TransitionUpdate	
TransitionUpdate	AutomatonClocks, TransitionGuard	
StateInvariant	AutomatonClocks	
TCADataConstraints		ConstraintExtension (CA data constraints)

Table 5.4: Dependencies between extensions in the TCA plugin

The syntactic checks described above are executed in methods *parseExtension* , *editExtension* and *validateExtension* (cf. Figure 5.3). For these checks, the new extensions use the helper classes TCAClocksParser and TCADataParser, contained in package *cwi.ea.extensions.clocks.parsers*. These parsers are generated from the grammar files `TCAClocks.g` and `TCAData.g` using the parser generator ANTLR [ANT].

5.1.3 From TCA to Formulas

We have implemented the translation of TCA to propositional formulas with linear arithmetic, as presented in Section 3.1.3, in ECT. In this section, we describe the implementation of our TCA formula generation.

ECT provides support for code generation (into an arbitrary target language) in package *cwi.codegen*, cf. Figure 5.5. A new code generator is defined by implementing the interface *ICodeGenerator* (or extending one of its abstract subclasses). Interface *IGenModel* encapsulates the data needed for code generation, that means, the properties (content information) of the underlying model that influence the generated code. It offers methods to manipulate these properties. Properties are

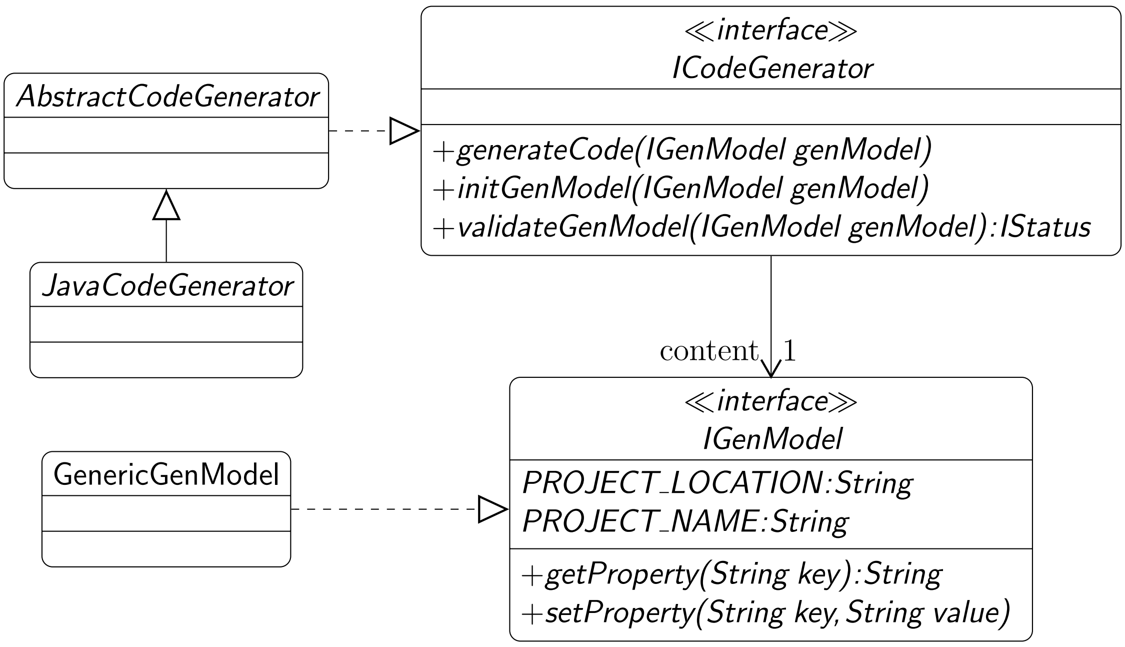

AbstractCodeGenerator
≪interface≫
ICodeGenerator
+generateCode(IGenModel genModel)
+initGenModel(IGenModel genModel)
+validateGenModel(IGenModel genModel):IStatus
JavaCodeGenerator
content
1
≪interface≫
IGenModel
GenericGenModel
PROJECT_LOCATION:String
PROJECT_NAME:String
+getProperty(String key):String
+setProperty(String key, String value)

Figure 5.5: Package *cwi.codegen*

key/value pairs of type String, new properties can be easily defined by calling *setProperty(key,value)* on an (until then) undefined key. *IGenModel* defines the two properties *PROJECT_LOCATION* and *PROJECT_Name* which every code generator should support.

The sources for the TCA formula generation extension are located in package *cwi.ea.extensions.clocks.codegen*, cf. Figure 5.6. Interface *SMTFormulaTemplate* can be seen as the main class of the extension. It defines a generic template for intermediate representation of TCA, and serves as a superclass for template classes that generate formulas for different back-ends. Currently, there exists only one class that implements *SMTFormulaTemplate* in package *cwi.ea.extensions.clocks.codegen*, MathSATFormulaTemplate. This class is used to generate input files for the MathSAT [mat] tool.[6]

For each of the constituents φ^X of the formula representation (3.16) (cf. Table 3.4 on Page 50), *SMTFormulaTemplate* defines a method signature *buildX* to generate the corresponding formula: *buildInit()* for φ^{init} (3.10), *buildTrans()* for φ^{trans} (3.13), *buildLocation()* for $\varphi^{location}$ (3.14), and *buildMutex()* for φ^{mutex} (3.15). The boolean parameter *unfold* in the latter three is used to determine whether to generate the step-abstract variant of the formula, that means with indices `t` and `t+1`, or the unfolded variant (cf. Section 3.2.2). In the former case, the return value should be a singleton list, in the latter case, the list should contain one entry for every unfolding depth, sorted in ascending order. The boolean parameter *product* in *buildTrans()* determines whether to generate delay transitions for products of TCA, cf. (3.17). Subclasses of *SMTFormulaTemplate* will have to implement these methods such that they generate formulas in the appropriate input format for the intended back-end solver, while following the constraints explained above.

[6]For MathSAT version 4.2.17.

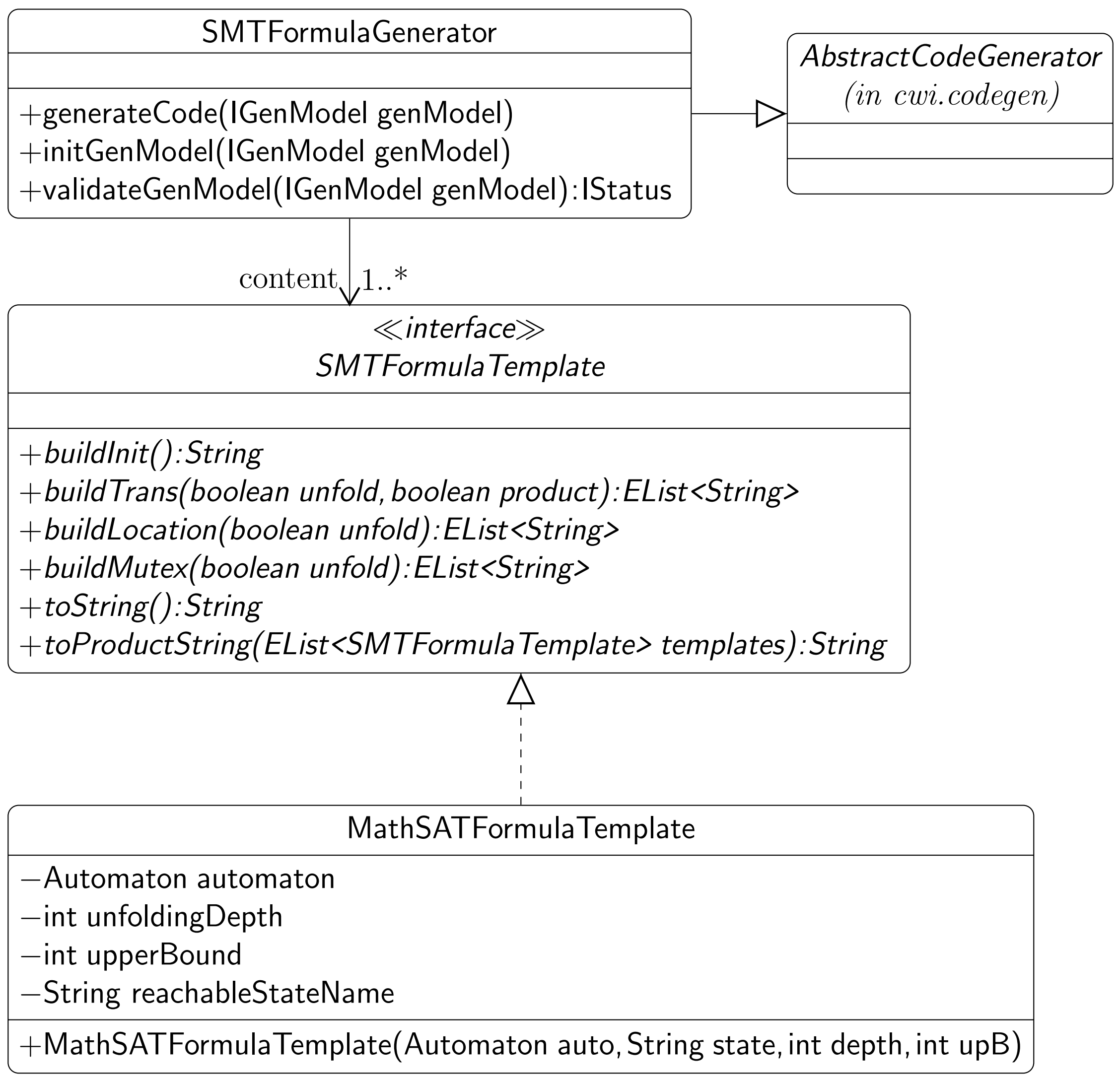

SMTFormulaGenerator
+generateCode(IGenModel genModel)
+initGenModel(IGenModel genModel)
+validateGenModel(IGenModel genModel):IStatus
AbstractCodeGenerator
(in cwi.codegen)
content
1..*
≪interface≫
SMTFormulaTemplate
+buildInit():String
+buildTrans(boolean unfold, boolean product):EList<String>
+buildLocation(boolean unfold):EList<String>
+buildMutex(boolean unfold):EList<String>
+toString():String
+toProductString(EList<SMTFormulaTemplate> templates):String
MathSATFormulaTemplate
−Automaton automaton
−int unfoldingDepth
−int upperBound
−String reachableStateName
+MathSATFormulaTemplate(Automaton auto, String state, int depth, int upB)

Figure 5.6: Package *cwi.ea.extensions.clocks.codegen*

The formula representation (in String format) for a single *SMTFormulaTemplate* object is obtained by calling method *toString()* . Method *toProductString()* is used to generate the product representation for a list of templates. Subclasses of *SMTFormulaTemplate* will typically implement these methods such that they call the *buildX* methods (passing the runtime value of *unfold* to the latter three), add additional information for the intended back-end solver as needed, and possibly sort the formulas in a specific way (cf. Section 4.2). Any implementation of the *toProductString()* method should be robust enough to produce the same result, independently of whether the object on which it is called (`this`) is contained in the list or not.

The constructor of MathSATFormulaTemplate is used to initialise the private fields with the appropriate values. It can be used for both finite and infinite data domains: for finite data domains, a lower bound of 0 is assumed by default. If the fourth parameter, upB ("upper bound"), is 0 as well, the data domain is assumed to be infinite. Otherwise, upB is required to have a value ≥ 1, which then determines the upper bound of the data domain (cf. also Remark 3.1.6 on finite data domains).

Parameter state can be used to specify the name of a location to be checked for k-step reachability, cf. Section 3.2.3; it is stored in field reachableStateName. The implementation of toString() respectively toProductString() generates the corresponding formulas. If parameter state is the empty string, no such property is generated.

Invocation of the formula generation from the editor (cf. Section 5.2.2) creates an instance of class SMTFormulaGenerator, which is a subclass of *AbstractCodeGenerator* from *cwi.codegen*, cf. Figure 5.6. SMTFormulaGenerator implements the methods from *ICodeGenerator* (cf. Figure 5.5) as follows. Method initGenModel() initialises the properties of the genModel required for formula generation: unfolding depth, data domain, location of the resulting output file, the set of automata to translate to formulas (this is a subset of all automata contained in the currently open file), and the target language (currently, only MathSAT format is supported). These settings are determined from user input to the formula generation wizard pages, cf. Figures 5.13, 5.14 and 5.15.

Method validateGenModel() checks that the genModel initialised in this way satisfies a number of additional (with respect to the constraints described in Section 5.1.2) syntactic and semantic requirements. This is needed for the resulting formulas to be well-defined: formula generation can be invoked for any EA automaton, but the results are well-defined only for TCA. The constraints include for example uniqueness of names, appropriate choice of finite data domain with respect to data values used in data constraints, or appropriate choice of enabled extensions (according to Section 5.1.2).

Method generateCode() starts the actual formula generation. It creates an *SMTFormulaTemplate* instance for each TCA selected for formula generation. The target language property determines the runtime type of these objects, that means which subclass of *SMTFormulaTemplate* to use. Actual parameters of the constructor of the appropriate runtime class are determined from the properties of the genModel. generateCode() then calls either toString() or toProductString(), depending on whether one or more TCA were selected for formula generation, and writes the resulting string to a file, using the corresponding property of genModel to determine the location.

5.1.4 Abstraction Refinement

Abstraction and refinement of TCA is not directly part of the ECT, in that it is not implemented as a plugin or extension within the graphical ECT interface. Instead, it is implemented as a standalone program, which performs abstraction and refinement on a file obtained from the ECT plugin (as explained in the preceding Section 5.1.3), and calls the preferred (corresponding) SMT solver for the verification part. The reason is that many SMT solvers already offer a graphical user interface (MathSAT does not, though). If needed, abstraction and refinement can be integrated into ECT on a command line basis, similar to the integration of mCRL2 (cf. [Kra11]), but we consider this less useful. The sources are found in package *cwi.ea.extensions.clocks.absref*, cf. Figure 5.7.

The interface *TCAAbstractor* serves as a base class for all implementations of abstraction functions working on TCA, independent of a specific target language. It defines method signatures that need to be supported by all such abstractors.

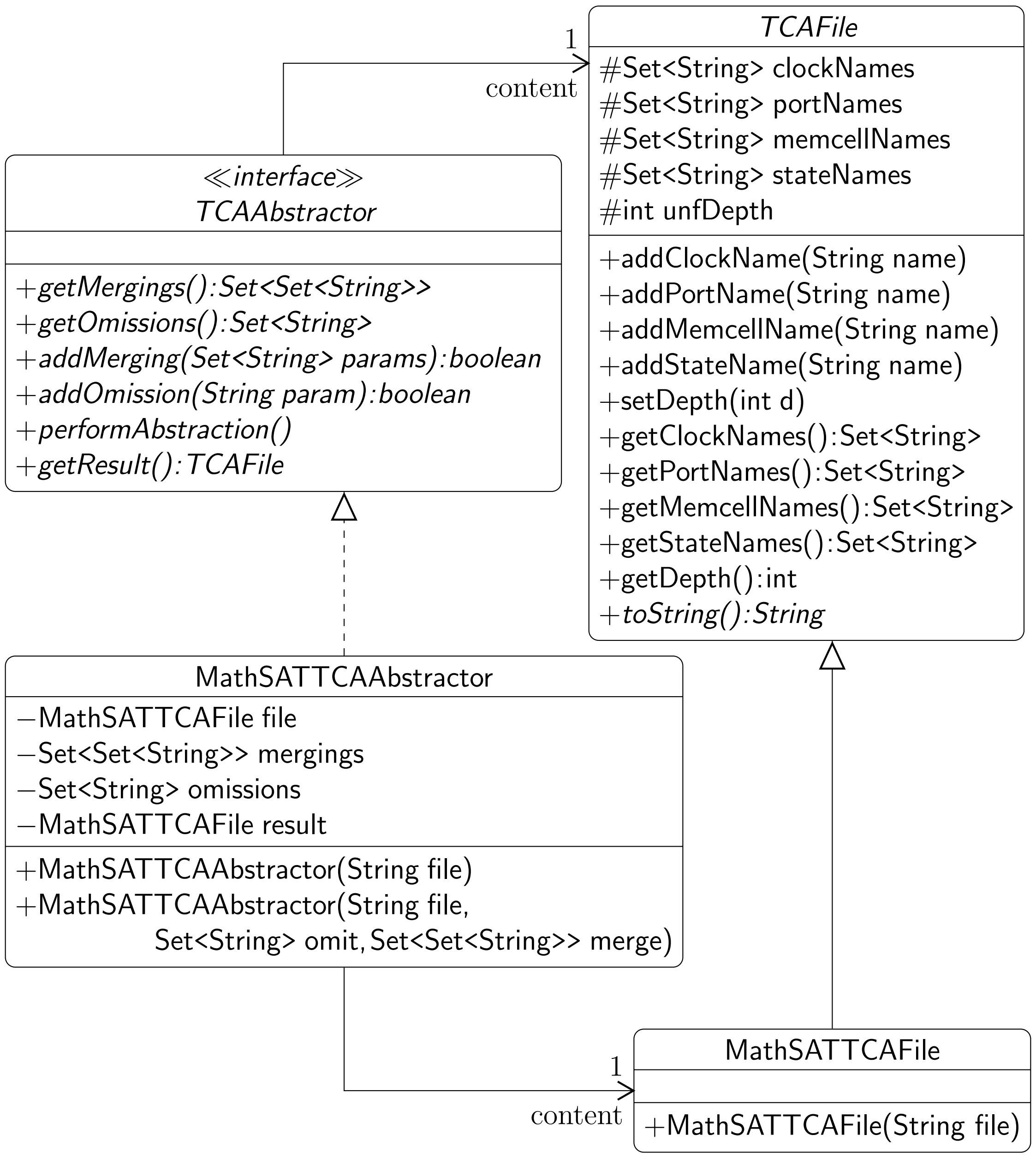

TCAFile
#Set<String> clockNames
#Set<String> portNames
#Set<String> memcellNames
#Set<String> stateNames
#int unfDepth
+addClockName(String name)
+addPortName(String name)
+addMemcellName(String name)
+addStateName(String name)
+setDepth(int d)
+getClockNames():Set<String>
+getPortNames():Set<String>
+getMemcellNames():Set<String>
+getStateNames():Set<String>
+getDepth():int
+toString():String
1
content
≪interface≫
TCAAbstractor
+getMergings():Set<Set<String>>
+getOmissions():Set<String>
+addMerging(Set<String> params):boolean
+addOmission(String param):boolean
+performAbstraction()
+getResult():TCAFile
MathSATTCAAbstractor
−MathSATTCAFile file
−Set<Set<String>> mergings
−Set<String> omissions
−MathSATTCAFile result
+MathSATTCAAbstractor(String file)
+MathSATTCAAbstractor(String file, Set<String> omit, Set<Set<String>> merge)
1
content
MathSATTCAFile
+MathSATTCAFile(String file)

Figure 5.7: Package *cwi.ea.extensions.clocks.absref*

Implementing class MathSATTCAAbstractor is tailored to work on files containing formulas in MathSAT format. In particular, MathSATTCAAbstractor implements the performAbstraction() method, by essentially implementing abstraction function α presented in Figure 4.1 in Section 4.1.

The abstract container class *TCAFile* encapsulates the intermediate representation of TCA in the process of abstraction, independent of a specific target language. It is used for both intermediate and final results.[7] Though *TCAFile* is abstract, it provides implementations for all methods except *toString()* , since this is the only method that depends on the target language, all other operations are performed on the internal/intermediate representation. Class MathSATTCAFile extends *TCAFile*, by implementing toString() such that the resulting String is in proper MathSAT format. Parser class MathSATFormatParser (not shown in Figure 5.7) is used by MathSATTCAAbstractor to parse a text file in MathSAT format and generate a MathSATTCAFile object from it. The text file can be obtained either from ECT, as explained in Section 5.1.3), or be generated by the toString() method of MathSATTCAFile itself. Class MathSATFormatParser is generated from grammar file `MathSATFormat.g` using the parser generator ANTLR [ANT].

Finally, class TCAMain uses the aforementioned classes to provide a little (command-line based) application to perform interactive abstraction refinement. Essentially, the application implements the three steps of the abstraction refinement paradigm (cf. the beginning of Chapter 4), looping through steps two and three. Figure 5.8 shows a conceptual overview of the application, where grey boxes indicate calls to external tools. The implementation is completely interactive, that means the user has to choose the initial abstraction (the application shows the list of abstractable parameters, though) as well as a refinement option and parameter to be refined (the application shows a list of potentially responsible parameters as well as the current counterexample, though). The type of the input file, and thus the SMT solver to be called, are determined from the input file ending. At the moment, only MathSAT is supported (file ending `.msat` or `.mathsat`), but it is easy to extend TCAMain to support other file types and solvers.

Notice that the abstraction refinement loop in Figure 5.8 has two exit points: either the conjunction of property and abstract system is unsatisfiable, in this case, the property holds for k steps (cf. Section 3.2.3); or the conjunction of original system, property and witness run is satisfiable, in this case, a counterexample to the property has been found.

5.2 Workflow

In this Section, we describe the typical workflow when working with the TCA plugin, using the FIFO buffers with expiration from Examples 2.3.3 and 2.3.10. The Section

[7]It would have been possible to extend interface *SMTFormulaTemplate* from package *cwi.ea.extensions.clocks.codegen* (cf. Figure 5.6) in such a way that it can be used for intermediate results of abstraction as well. Yet, this would have prevented us from providing method implementations in abstract class *TCAFile*. Moreover, we prefer to keep the approach modular and flexible, by maintaining two intermediate formats (*SMTFormulaTemplate* for code generation, and *TCAFile* for abstraction).

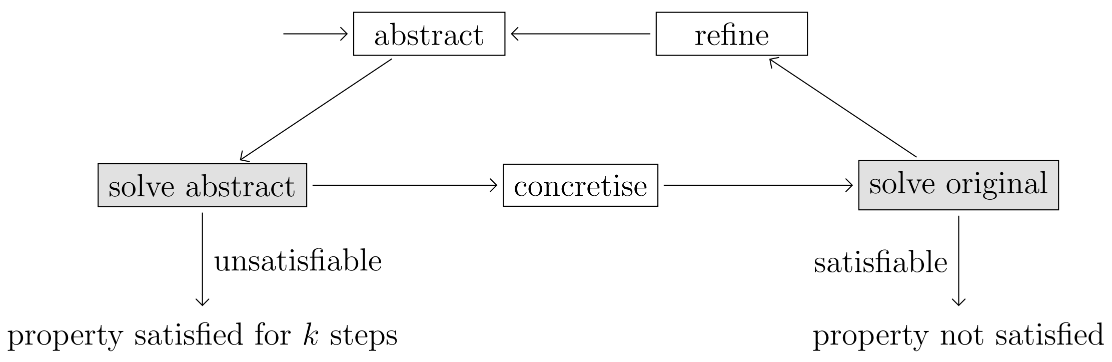


Figure 5.8: Implementation of Abstraction Refinement Loop, Conceptual Overview

can also be seen as a little "Getting started" tutorial to the TCA plugin. We assume that ECT and in particular the EA framework is installed already.[8] For instructions how to install the plugins, please refer to the ECT website `http://reo.project.cwi.nl`.

5.2.1 Editing

The first step is to draw the TCA. From the Eclipse workbench, create a new *General Project* (*File* → *New* → *Project* → *General* → *Project*) with name `Workflow`. In the project, create a new EA automaton file (*File* → *New* → *Other*, scroll down to the *Reo* wizard and choose *Automaton*), and name it `Workflow.ea`. From the palette that appears on the right, choose *Automaton* and left-click on the empty canvas to create a new EA automaton. A menu appears that allows to choose one of the predefined automaton types. If none of the types is chosen, the new automaton has no extensions enabled. Choose *Timed Constraint Automaton* and give it the name `Buffer`. A rectangle, indicating the drawing area for the new automaton, appears on the canvas, cf. Figure 5.9.

The upper part of the drawing area contains general information: the name of the automaton, and information from extensions which are applicable on automaton level. The two symbols below the name indicate that extensions AutomatonPortNames (☰) and AutomatonClocks (🕗) are already enabled for this TCA (these extensions were enabled automatically when automaton type *Timed Constraint Automaton* was chosen). The StateMemoryExtension is not enabled by default. To enable it, right-click on the automaton, choose *Extensions* from the context menu, and check *State Memory*. Since StateMemoryExtension is not applicable to Automaton, there is no visible change. Add two ports `p` and `q` and a clock `x` to the automaton: click on the ☰ respectively 🕗 label until a text field shows up, then enter the text `p,q` respectively `x`.

Next, we add locations. From the palette on the right, choose *State*, and click inside the (lower part of the) rectangle to add a location. Give it the name `empty`. The ingoing arrow indicates that StartStateExtension is enabled for this automaton

[8]All descriptions and images in this section are based on version 3.2.0 of ECT, and version 3.2.9 of the EA framework.

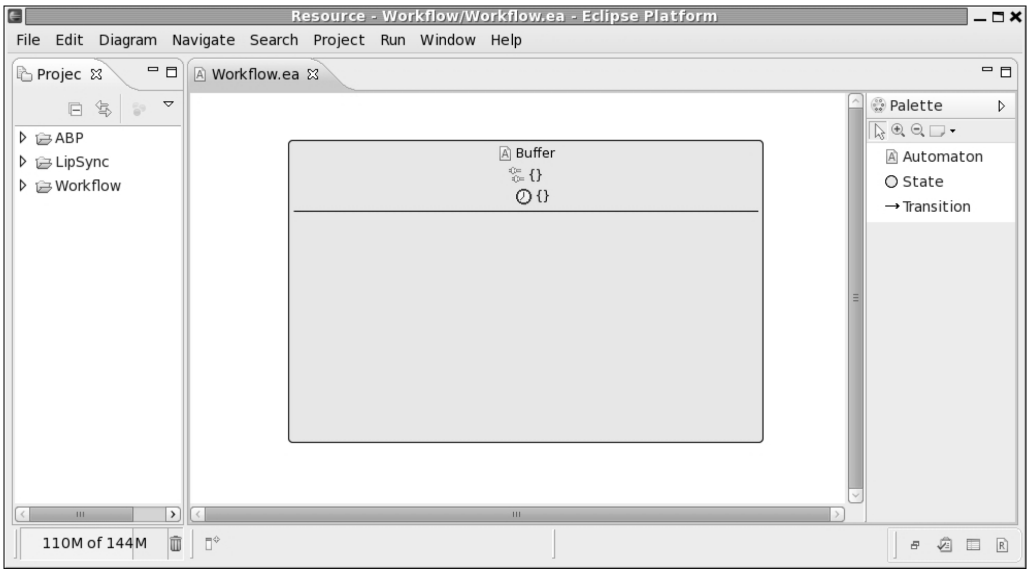

Resource - Workflow/Workflow.ea - Eclipse Platform
File Edit Diagram Navigate Search Project Run Window Help
Projec
ABP
LipSync
Workflow
Workflow.ea
Buffer
{}
{}
Palette
Automaton
State
Transition
110M of 144M

Figure 5.9: Workbench Overview

(again, this extension was enabled automatically when automaton type *Timed Constraint Automaton* was chosen), by default, the first location that is created is set to be the initial location. The initial location can be changed from the context menu of locations. Add a second location with the name `full`. The symbols next to the locations show that StateMemoryExtension (●) and StateInvariant (⧗) are enabled. Add a memory cell `m` to `full`, and set the invariant of `full` to `x<=3`.

To add a transition from `empty` to `full`, choose *Transition* from the palette, click on `empty`, and drag the other end of the transition to `full`. For every (enabled) extension which is applicable to Transition, the new transition has a corresponding label. The four labels are ◷? for TransitionGuard, ◷! for TransitionUpdate, ▣ for TCADataConstraint, and ⚙ for TransitionPortNames. Set the labels pursuant to Figure 2.5, add two transitions from `full` to `empty`, and set the labels accordingly. Note that memory cell references (`s.m` and `t.m`) in the data guards have to be prefixed with an additional `$` in the editor. This is due to liberal naming conventions in ECT which allow dots `.` to appear in names. The final TCA should look similar to the one shown in Figure 5.10 (in order not to clutter up the picture, we have removed empty transition labels).

Repeat the above steps, and create a second instance of the buffer in the same file `Workflow.ea`. Name it `Buffer2`, with locations `empty2` and `full2`, memory cell `m2` (in `full2`), clock `x2` and ports `q` and `r` (cf. Figure 2.6). Make sure to use `q` on the transition from `empty2` to `full2`. The resulting TCA should look similar to the one shown in Figure 5.11.

Throughout the editing process, every modification is checked for syntactic correctness, according to the syntactic requirements explained in Section 5.1.2. If a syntactic error is detected, the label of the extension (for example ◷! or ⧗) in which the error occurred is replaced by the error marker ⊗. A tooltip appears on mouseover which gives information about the error, and in this way helps to resolve it. As an example, while composing the `Buffer` automaton, we could try to use port `p` on a

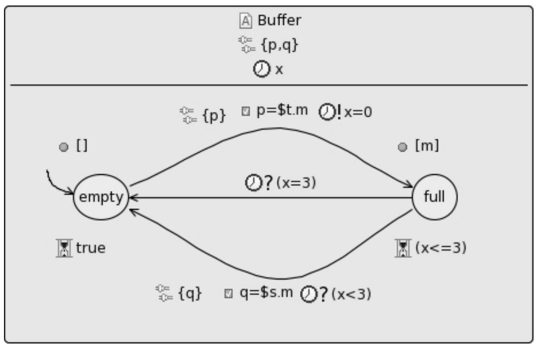


Figure 5.10: First Buffer

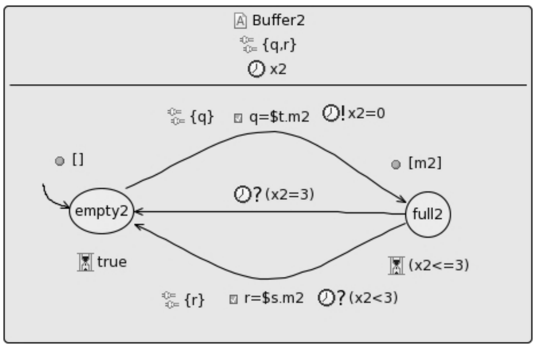


Figure 5.11: Second Buffer

transition *before* declaring it on automaton level. Figure 5.12 shows the resulting error marker and tooltip. To resolve the error, we need to do two things: first, add

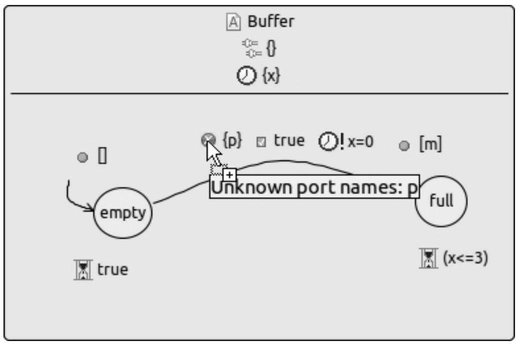


Figure 5.12: Indication of Errors

p on automaton level (i.e., in the field with label field just below the name). After that, the error marker remains active, because the editor only checks the extension that has just been edited (AutomatonPortNames in this case). Therefore, we need to edit the TransitionPortNames extension again (simply click on the error marker until the text field appears, and confirm). The TransitionPortNames extension is checked again, and since p is now declared on automaton level, the error marker disappears.

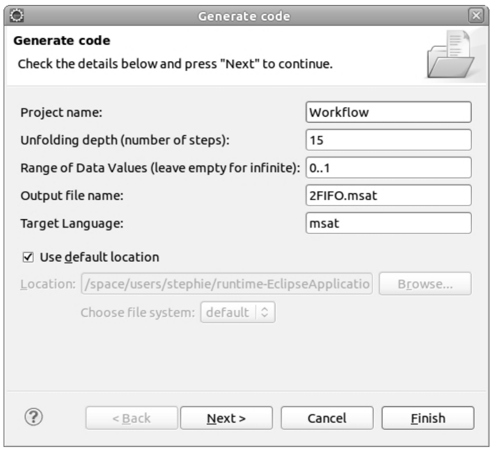
5.2.2 Formula Generation

Formula generation is invoked from the context menu of automata: right-click on one of the automata, and choose *Generate code → SMT Formula Generator* to open the code generation wizard.

The first page of the wizard allows to specify the name and directory of the resulting file, the unfolding depth, the range of data values, and the target language (Figure 5.13). Since currently only the MathSAT solver back-end is supported, the only admissible entry in the target language field is `msat`, which naturally is also the default. Enter `Workflow` as *Project name*, `2FIFO.msat` as *Output file name*, and click `Next>` at the bottom of the wizard page.

The second page (Figure 5.14) allows to choose the (combination of) automata to generate code for (the resulting formulas contain delay transitions, cf. Definition 3.1.8, iff more than one automaton is selected). Select both TCA and click `Next>`.

Finally, the third wizard page (Figure 5.15) allows to select up to one location for each automaton that was selected on the previous (second) wizard page, to check for *k*-step reachability. Unfold the lists of locations, select locations `full` and `full2`, and click `Finish`. Note that if locations are selected for a (non-empty) subset of automata only, then *k*-step reachability is checked for all possible combinations of the selected locations with any location from the other automata. For example, if we would select location `full` only, i.e., not select a location for `Buffer2`, reachability would be checked for any of the product locations `(full,empty2)` and `(full,full2)`.



Generate code

Generate code

Check the details below and press "Next" to continue.

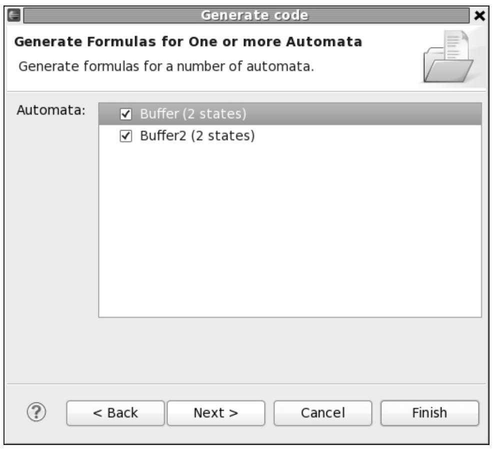
Project name:	Workflow
Unfolding depth (number of steps):	15
Range of Data Values (leave empty for infinite):	0..1
Output file name:	2FIFO.msat
Target Language:	msat

☑ Use default location

Location: /space/users/stephie/runtime-EclipseApplicatio Browse...

Choose file system: default

Figure 5.13: Code Generation Wizard, First Page



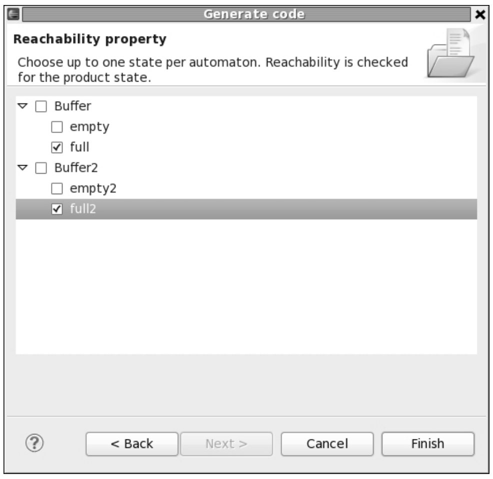
Generate code

Generate Formulas for One or more Automata

Generate formulas for a number of automata.

Automata:
- ☑ Buffer (2 states)
- ☑ Buffer2 (2 states)

Figure 5.14: Code Generation Wizard, Second Page



Generate code

Reachability property

Choose up to one state per automaton. Reachability is checked for the product state.

- ☐ Buffer
  - ☐ empty
  - ☑ full
- ☐ Buffer2
  - ☐ empty2
  - ☑ full2

Figure 5.15: Code Generation Wizard, Third Page

After clicking `Finish`, the resulting file `2FIFO.msat` can be found in the *Project Explorer* view to the left of the canvas, it is located in the *src* subdirectory in the

Workflow project, cf. Figure 5.16.

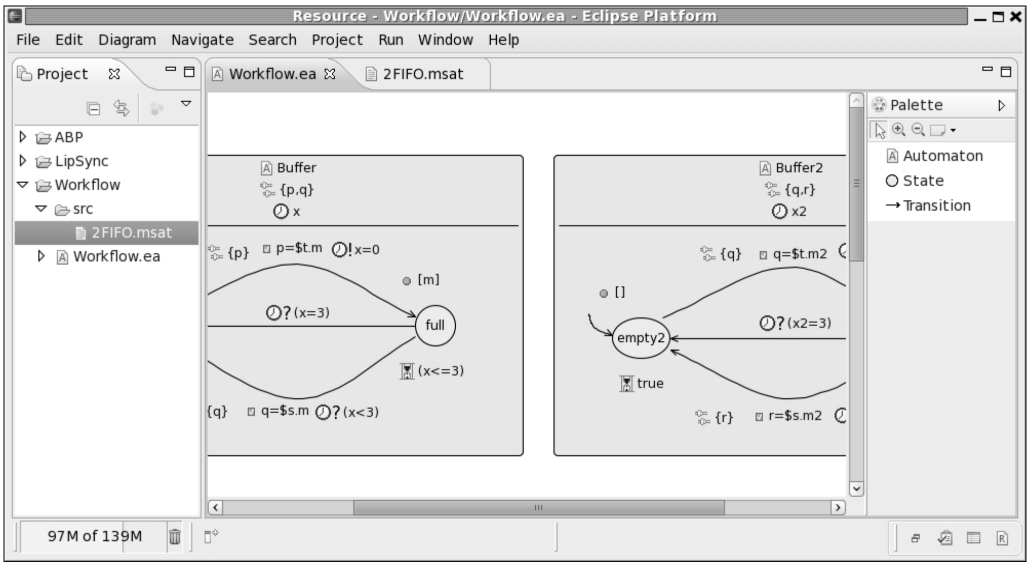


Figure 5.16: Workbench, Project Explorer View

5.2.3 Verification

We now have two options to continue. The first option is to directly call MathSAT on the generated file `2FIFO.msat`. In general, MathSAT is invoked by calling `mathsat [options] input_file` from the command line. In our context, the minimal set of options includes:[9]

`-solve`: tells the solver to solve the formula (other options include for example transformation to CNF)

`-input=msat`: specifies the input format (other options are `smt` or `dimacs`)

`-logic=QF_LRA`: specifies the logic to be used (Quantifier Free Linear Real Arithmetic)

Other useful options include for example

`-print_model`: prints one satisfying valuation (model), if it exists

`-allsat`: prints *all* satisfying valuations, if they exist (only recommended for small problems)

`-outfile=FILE`: redirects the output from stdout to file `FILE`

[9]For a complete and detailed overview and explanation of the available options, please refer to [mat]

The output of a call to MathSAT always starts with some statistical information about the input problem and the involved theory solvers. The essential information can be found at the very end of the output: the last line of the output of the call `mathsat -solve -input=msat -logic=QF_LRA 2FIFO.msat` says `sat`, which means the set of input formulas is satisfiable, and thus the "error state" `(full,full2)` is reachable.

The second option is to call the abstraction refinement application (cf. Section 5.1.4) on the generated file `2FIFO.msat`: `java TCAMain 2FIFO.msat`. As explained in Section 5.1.4, the application first asks the user to create the initial abstraction. For ports, for example, the application outputs

```
Choose ports to abstract/merge (type numbers of ports
to be merged, separated by blanks, type 'X' to end):
1: r
2: p
3: q
```

and for clocks, it outputs

```
Choose clocks to abstract/remove (type numbers,
separated by blanks):
Set of clocks is
1: x
2: x2
```

Type `1` (and confirm) to remove clock `x`. After this initial abstraction is determined, the application internally calls MathSAT on the abstract system. Since the "error state" `(full,full2)` was already reachable in the original system, it is of course reachable in the abstract system as well. The next output of the application is

```
Spurious counterexample found, abstraction needs to be refined.
Choose refinement option (type number):
1: rule out counterexample trace
2: refine a parameter
```

Type `2`, this gives the output

```
Choose parameter to refine (type number):
1: x
```

Now type `1` to refine clock `x`. The application refines the abstraction, and calls MathSAT again on the resulting system (which is the original system already). The next (final) output is

```
Property does not hold.
```

and the application terminates.

5.3 Case Studies

As stated in Chapter 2, TCA are specially tailored to implement coordinating connectors in networks where timed components communicate by exchanging data through multiple channels. In this way, TCA impose a certain communication pattern on associated components, that means, they force the components to behave (communicate) in a certain way. An obvious application area is therefore to use TCA for modelling time- and data-aware communication protocols.

In this section, we describe two such protocols, present experimental results we got from modelling and analysing the protocols with our TCA plugin for ECT, and discuss the benefits of using TCA as underlying formalism for these case studies and in general.

5.3.1 Alternating Bit Protocol

In this section, we present the *alternating bit protocol* (ABP). The ABP is a network protocol, which ensures successful transmission of data elements between a sender and a receiver over unreliable channels. It is one of the standard benchmarks in the context of component based systems and process algebra, and has been discussed in detail for example in [Mil89, Fok00, LM87].

Essentially following the description in [Mil89], we design the protocol from four subcomponents: the Sender, the Receiver, and two unreliable channels Channel1 and Channel2 connecting the former two. Each of these components is modelled with a separate TCA. We assume that the channels may loose, but not corrupt or duplicate, data at random. Yet, it is very easy to change this behaviour, simply by exchanging the TCA that model the channels. For an overview of how to model timed channels with different behaviour, see for example [ABdBR07]. A conceptual overview of the components is shown in Figure 5.17. Arrows indicate the intended direction of dataflow, labels indicate the ports through which the components communicate.

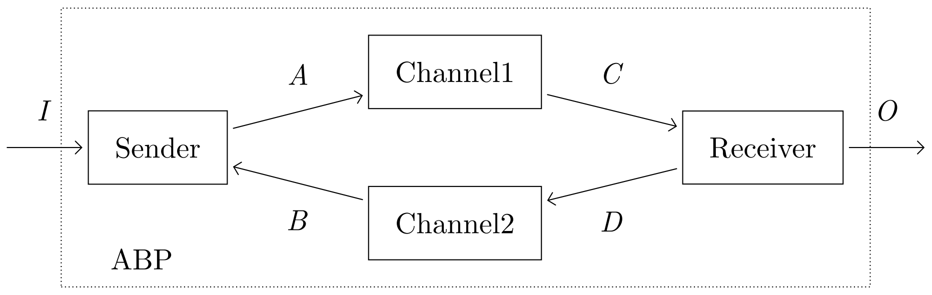


Figure 5.17: ABP Connector, Conceptual Overview

The protocol works as follows: after accepting input (from the environment) through port I, the Sender starts a timer, and sends the message to the Receiver via Channel1, i.e., it sends the message to Channel1 through port A, and Channel1 in turn sends the message to the Receiver via port C. The Sender attaches a control bit b to the message, and expects the Receiver to send back the corresponding control bit b through Channel2 as acknowledgement. After having received the acknowledgement, the Sender is ready to accept another input from the environment, which it sends to the Receiver with attached control bit $\neg b$ (this is where the name *alternating* stems from). If the timer of the Sender expires before it receives the acknowledgement bit b, or if it receives acknowledgement bit $\neg b$ (which it ignores), it assumes an error has occurred, resets the timer and resends the message with bit b.

The Receiver works complementary: it receives a message, together with a control bit b, from the Sender through Channel1. After delivering the message to the environment through port O, the Receiver sets a timer, and sends bit b as acknowledgement to the Sender through Channel2. Next, it expects a message tagged with bit $\neg b$. If the timer expires, or the next message is tagged with b again (which the Receiver ignores), the Receiver assumes an error has occurred, resets the timer and resends the acknowledgement bit b.

We assume an arbitrary but fixed, finite set of messages $\mathcal{M}sg$. The data domain is $\mathcal{D}ata{=}\mathcal{M}sg\cup\mathcal{M}sg\times\{0,1\}\cup\{0,1\}$. The three subsets of $\mathcal{D}ata$ correspond to messages sent from the environment to the Sender, and from the Receiver to the environment ($\mathcal{M}sg$), messages tagged with control bits sent from the Sender to the Receiver ($\mathcal{M}sg\times\{0,1\}$), and acknowledgement bits sent from the Receiver to the Sender ($\{0,1\}$).

The TCA for the Sender, the Receiver and the two channels are shown in Figures 5.18, 5.19 and 5.20. Since ports can only transmit a single data item, we model ports A (between Sender and Channel1) and C (between Channel1 and Receiver) using two ports A_1,A_2 and C_1,C_2, respectively. This is because for a pair $(m,b){\in}(\mathcal{M}sg\times\{0,1\})$, we need to be able to reason about the constituents m and b separately. For both Sender and Receiver, we assume a timeout of 2 for resending the message and acknowledgement, respectively. For example, after the Sender has sent a message and has moved to location *wait*0, it waits for acknowledgement bit 0 before moving to location *idle*1. If this bit does not arrive before clock x has reached value 2, the Sender moves back to location *send*0, where it waits at most one more time unit before it resends the message.

The TCA modelling the entire protocol component—we call it $\mathrm{T}_{\mathrm{ABP}}$—is obtained by composing the TCA of the four subcomponents (cf. Definition 2.3.9), that means

$$\mathrm{T}_{\mathrm{ABP}} = (\text{Sender} \bowtie \text{Receiver} \bowtie \text{Channel1} \bowtie \text{Channel2})$$

5.3.1.1 Verification

While the internal behaviour of the ABP ensures reliable transmission of messages over unreliable channels, from the outside, it behaves as a perfect buffer of capacity one (cf. [Mil89]). That is, it accepts and delivers messages from and to the network (through ports I and O, respectively) alternately, and the order of data elements is not changed.

The alternation is described by the LTL formula

$$\Box((I \rightarrow \bigcirc(\neg I\,\mathcal{U}\,O)) \wedge (O \rightarrow \bigcirc(\neg O\,\mathcal{U}\,I))),$$

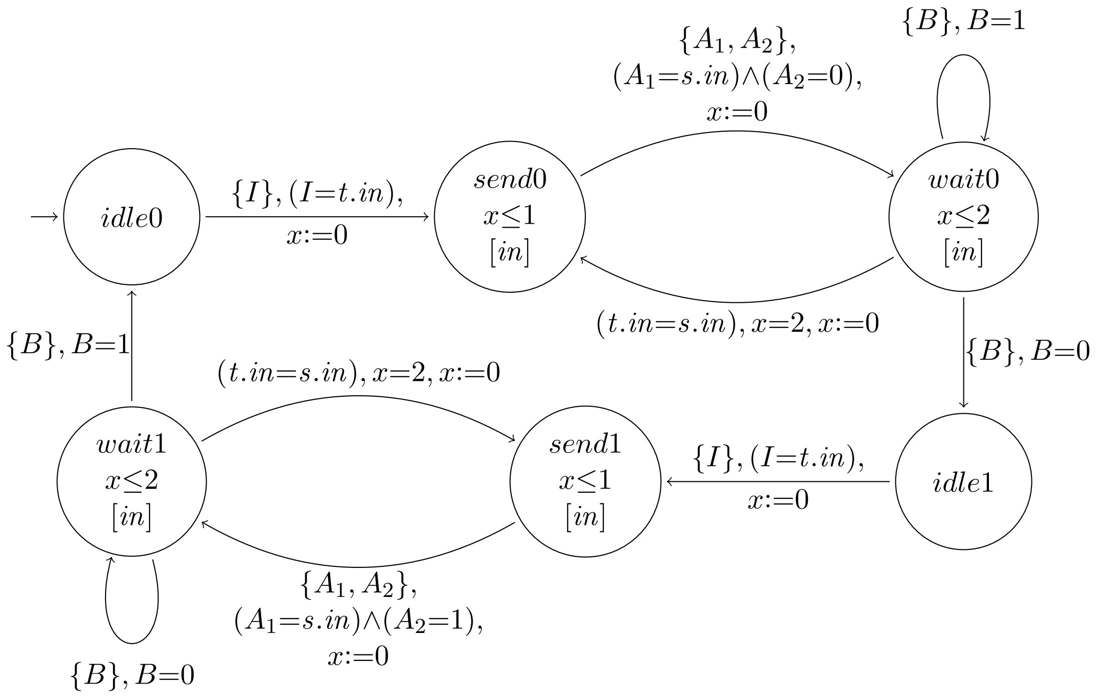


Figure 5.18: ABP, Sender

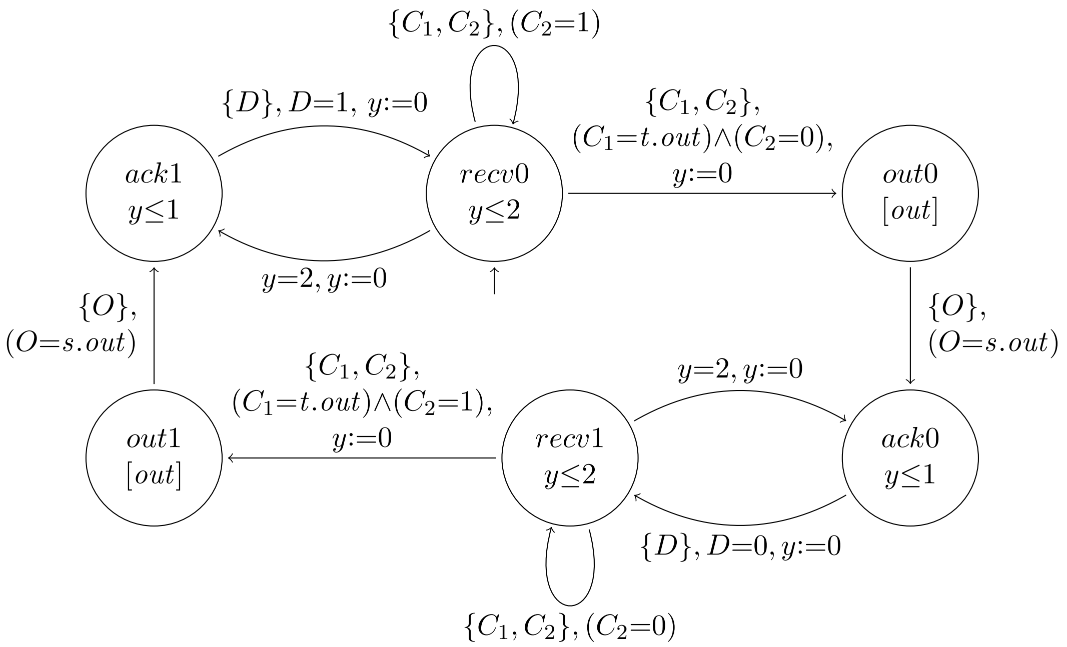


Figure 5.19: ABP, Receiver

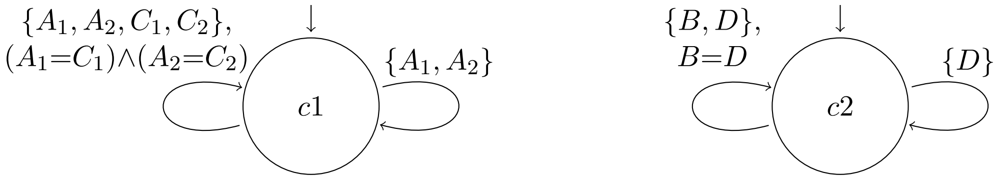


Figure 5.20: ABP, Channel1 (left), Channel2 (right)

which expresses that between any two communications through port I, there is a communication through port O, and vice versa. We call this property `Buffer`.

To check for the correct order of data elements, we identify a set of error locations. First recall that locations in $\mathrm{T}_{\mathrm{ABP}}$ comprise one location for each of the four subcomponents, for example, the initial location of $\mathrm{T}_{\mathrm{ABP}}$ is $(idle0, revc0, c1, c2)$. The error locations are

$$\{(wait\mathrm{i}, out\mathrm{i}, c1, c2) \mid in{\neq}out, i{=}0,1\} \cup \{(send\mathrm{i}, out\mathrm{i}, c1, c2) \mid in{\neq}out, i{=}0,1\}$$

Each of these locations corresponds to a configuration where the Sender subcomponent has received a data item through port I and stored it in memory cell $[in]$, but the Receiver subcomponent has stored a different data item in memory cell *out* which it is about to send to the environment through port O. Note that the formulation of this property relies on the first property of alternating dataflow through I and O. We call the property expressing that none of the error states is reachable `Error`.

We have modelled the TCA of the ABP with the ECT plugin. For performance comparison, and to show that our approach scales very well on reachability properties, we compare three unfolding depths $k{\in}\{20, 50, 100\}$. We have generated the corresponding formula representation from within the editor, as described in Section 5.2.2. To show the performance improvements gained from abstraction, we have identified a tailored abstraction function for each of the properties. First observe that timing information can be considered irrelevant for both properties. Moreover, observe that `Buffer` does not rely on exact data values, but only reasons about activity on ports. For `Error`, we define an abstraction function α_1 that removes all timing information from the system: $\mathcal{O}_1{=}\{x, y\}$, and $\gamma_1{=}id$. For `Buffer`, we define an even coarser abstraction function α_2, which in addition removes all information about the exact data values: $\mathcal{O}_2{=}\{x, y, (I{=}t.in), (A_1{=}s.in), (t.in{=}s.in), (C_1{=}t.out), (O{=}s.out)\}$, and $\gamma_2{=}id$.

Table 5.21 shows our experimental results for the different unfolding depths, properties and abstractions. Note that since `Buffer` describes correct alternation of data flow through ports I and O, we need to check the negation of `Buffer`. Abstraction function α_2 removes all information about concrete data values, therefore, the error states are trivially reachable under this abstraction, and we cannot expect `Error` to hold. We have marked the corresponding entries with a slanted font and the additional entry (S) (for "satisfiable"). All other properties are satisfied, which means that the result of verification is "unsatisfiable".

The results in Table 5.21 clearly show that our approach is tailored to reachability properties, and that on these, it scales very well for large unfolding depths. While

	k=20		k=50		k=100	
	¬`Buffer`	`Error`	¬`Buffer`	`Error`	¬`Buffer`	`Error`
$\varphi(\mathrm{T}_{\mathrm{ABP}})_k$	1.050s 20.79MB	1.013s 18.85MB	52.50s 61.59MB	13.12s 39.19MB	1690s 508.53MB	80.98s 111.62MB
$\alpha_1(\varphi(\mathrm{T}_{\mathrm{ABP}}))_k$	1.704s 20.93MB	0.799s 19.44MB	519.9s 215.02MB	7.082s 32.76MB	segm. fault	32.92s 57.09MB
$\alpha_2(\varphi(\mathrm{T}_{\mathrm{ABP}}))_k$	1.430s 19.23MB	*0.175s (S) 15.99MB*	458.7s 232.60MB	*0.880s (S) 21.215MB*	segm. fault	*3.984s (S) 29.11MB*

All experiments have been carried out with MathSAT, version 4.2.17, on an Intel Core2 Duo CPU E4500, with 2.20GHz and 2.5GB RAM

Table 5.21: Experimental Results for the ABP

for k=20, the two properties take around the same time and memory consumption, checking `Error` is factor 4 faster, with factor 1.5 less memory, than `Buffer` for k=50, and almost factor 20 faster, with factor 4 less memory, for k=100. Comparing the same property on different unfolding depths, time and memory consumption increase by factors 50 and 3 (k=20 to k=50), and factors 30 and 8 (k=50 to k=100) for `Buffer`, while these factors are limited to 13 and 2 (k=20 to k=50) and 6 and 3 (k=50 to k=100) for `Error`.

As a second result, Table 5.21 shows the improved performance for `Error` on the abstract system, resulting in a speed-up of factor 1.2 for k=20, factor 1.8 for k=50, and almost factor 2.5 for k=100. Memory consumption is almost the same for k=20 and k=50, but reduces to half for k=100. Note that verification is very fast in case the reachability property `Error` does not hold (i.e., where the input is satisfiable). In contrast to this, performance of `Buffer` on the abstract system decreases, even leading to a segmentation fault for k=100. Though this might seem surprising at first glance, the reason is obvious: since `Buffer` reasons about all possible runs, and the abstract system permits more runs than the concrete system, checking `Buffer` on the abstract system is more expensive. What can be seen though is a slightly improved performance when comparing the two abstractions.

5.3.2 Lip-Synchronisation Protocol

In this section, we describe a *Lip-synchronisation protocol* (LSP). The LSP was first described in the synchronous language Esterel [SHH92]. Later, specifications were presented amongst others using timed LOTOS [Reg93], LOTOS/QTL [BBBC94, BBBC97], timed CSP [ABSS96], timed automata [BFK$^+$98, KLP10], and the Duration Calculus [MLWZ01].

The problem of lip-synchronisation is the following: a presentation device (for example a media player) receives input from two media sources: a Sound Stream and a Video Stream, and should display the two streams synchronously. Yet, small delays are not human-perceivable, therefore, synchrony needs not be perfect, but certain deviations of the actual presentation times from the optimal presentation times are acceptable. The LSP is used to ensure this degree of synchrony. For this, it has to take into account three major points:

Firstly, the frequencies of the two streams may be different, i.e., there is no (at least not necessarily) one-to-one correspondence between frames of the two streams.[10]

Secondly, the streams may experience a phenomenon called *jitter*. In an ideal scenario, frames are received from the streams with a constant frequency. Yet, the actual arrival times of frames may deviate from these optimal arrival times, for example due to transmission delays. These deviations from the optimal presentation time on *the same* stream are called jitter. Intuitively, the user perceives jitter in case the sound (equivalently video) presentation does not run "smoothly", for example because some parts are skipped or presented too fast.

From [BFK+98], we adopt the notions of *anchored* and *non-anchored* jitter: anchored jitter describes deviations that only depend on the current frame. That means, each frame arrives within a certain interval around *its own* optimal arrival time. Non-anchored jitter, on the other hand, describes deviations that depend on the previous frame. That means, each frame arrives within a certain interval after the *previous* frame.

The last point the LSP has to consider is that the two streams can "drift apart": each frame has an optimal position on the respective other stream. The term *skew* describes deviations from this optimal position on *the other* stream. Intuitively, the user perceives skew in case the sound and video presentation "do not agree", that means if a sound (for example the sound of breaking pottery) is audible significantly before or after the corresponding action (for example a falling mug) is visible.

Summing up, the LSP has to ensure that the two streams are interleaved in the right way, and that the presentation of frames does not violate the acceptable bounds on jitter and skew.

Notation 5.3.1 (Arrival Times of Frames). In the sequel, we use t_i^{opt} to refer to the optimal arrival time of the i-th frame (sound or video), and t_i^{act} to refer to the actual arrival time of the i-th frame.

In line with previous specifications ([SHH92, Reg93, BFK+98]), we design the protocol as follows. A sound frame should be displayed every 30 milliseconds (ms), no jitter is allowed on the Sound Stream, i.e., $t_i^{act}{=}t_i^{opt}$, and $t_i^{act}{=}(t_{i-1}^{act}{+}30)$ for all frames. A video frame should ideally be displayed every 40ms, but we allow non-anchored jitter of ±5ms: $(t_i^{act}{-}5){\leq}(t_{i-1}^{act}{+}40){\leq}(t_i^{act}{+}5)$. That means, to ensure that a human perceives the media presentation as synchronous, it is sufficient to display each video frame within the interval [35ms,45ms] after the previous frame. The video presentation may precede the sound presentation by 15ms (skew), and may lag behind by 150ms. The fact that jitter is not allowed on the Sound Stream actually allows us to

[10]As pointed out in [BFK+98], if there was a one-to-one correspondence, the obvious solution to achieve synchrony would be to multiplex the streams for transmission, and demultiplex them at the presentation device.

interpret skew as anchored jitter on the Video Stream: $(t_i^{act}-15) \leq t_i^{opt} \leq (t_i^{act}+150)$. We restrict the models of the streams to the arrival times of frames received from the streams. Similarly, we do not model the presentation device, but assume that frames are presented when the corresponding signal (from the LSP) occurs. Further, we assume that the presentation of a video frame can be delayed for an arbitrary amount of time if the frame arrives too early.

We model the protocol with five TCA: Sound Manager, Video Manager, Skew Observer, Jitter Observer, and Initialiser. The design of the TCA is inspired by the timed automata in [BFK+98]. Yet, due to the extended modelling power of TCA, we obtain a more concise model (cf. also Section 5.3.3), in particular for the Skew Observer, which we model using a single counter (rather than using multiple clocks in [BFK+98]). A conceptual overview of the protocol components is shown in Figure 5.22. We omit the Initialiser, since its only purpose is to start the protocol components in the right order. The presentation device is added to Figure 5.22 for illustration only, it is not part of the protocol. As for the ABP (Figure 5.17), arrows indicate the intended direction of communication, labels indicate the ports through which the TCA communicate.

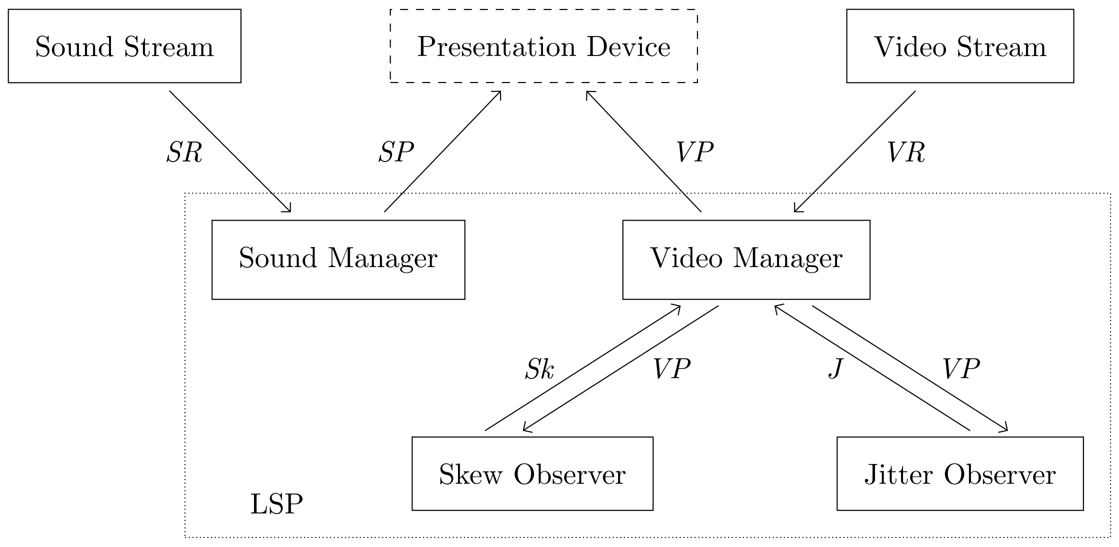


Figure 5.22: LSP: Conceptual Overview

Since the TCA of the LSP (and their interaction) are more complex than the TCA of the ABP, we now describe each TCA in detail. In the end, the TCA modelling the protocol component—we call that TCA T_{LSP}—is obtained by composing the TCA of the five subcomponents (cf. Definition 2.3.9), that means

$$T_{LSP} = (\text{Initialiser} \bowtie \text{SoundManager} \bowtie \text{VideoManager} \quad (5.1)$$

$$\bowtie \text{SkewObserver} \bowtie \text{JitterObserver}) \quad (5.2)$$

5.3.2.1 Sound Manager

The conditions on presentation of sound are very strict: a sound frame should be presented every 30ms, and no jitter is allowed on the Sound Stream. The task of

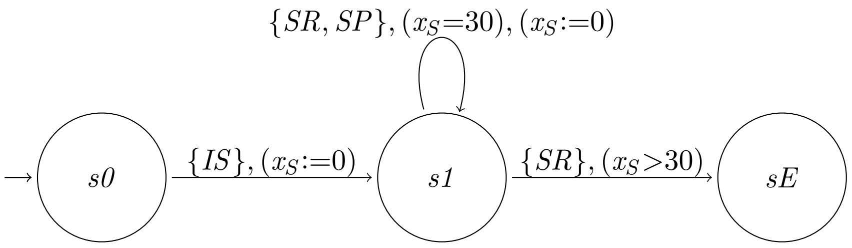


Figure 5.23: LSP, Sound Manager

the Sound Manager (Figure 5.23) is to ensure this behaviour. It uses a clock x_S to measure the time distance between two subsequent sound frames. After being initialised by the Initialiser (cf. Section 5.3.2.5) through port *IS* on presentation of the first sound frame, the Sound Manager starts its cyclic behaviour, waiting for a sound frame to be ready (signalled through port *SR*) every 30ms, and sending to the presentation device the order to present the sound frame (through port *SP*) at the same moment. If the signal that a sound frame is ready arrives late, the Sound Manager moves to its error location *sE*.

For explanatory purposes, in the sequel we use the terms "presentation of a sound frame" and "communication through port *SP*" interchangeably.

5.3.2.2 Video Manager

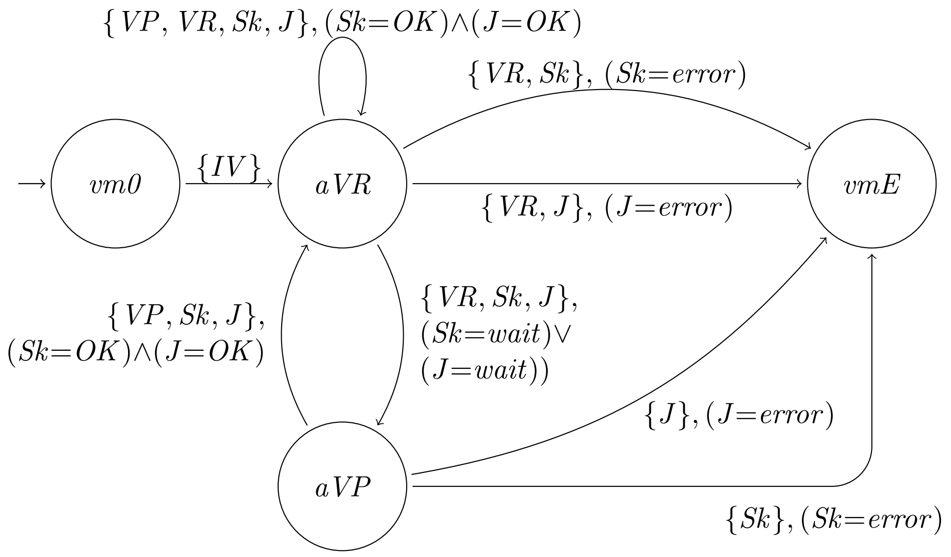
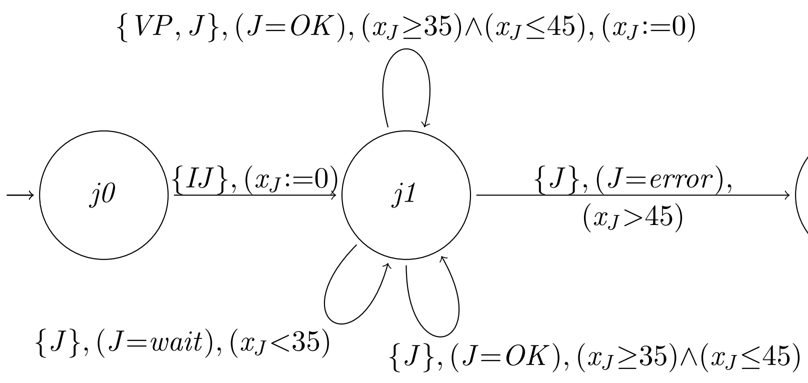

{VP, VR, Sk, J}, (Sk=OK)∧(J=OK)
{VR, Sk}, (Sk=error)
vm0
{IV}
aVR
vmE
{VR, J}, (J=error)
{VP, Sk, J}, (Sk=OK)∧(J=OK)
{VR, Sk, J}, (Sk=wait)∨ (J=wait))
{J}, (J=error)
aVP
{Sk}, (Sk=error)

Figure 5.24: LSP, Video Manager

The Video Manager (Figure 5.24) ensures that the timing of video presentation conforms to the bounds described above. Yet, the Video Manager does not control the timing itself, but for this consults the two "helper" components Jitter Observer (cf. Section 5.3.2.3) and Skew Observer (cf. Section 5.3.2.4). In particular, the Video Manager does not use any clocks. The Video Manager is initialised by the Initialiser (cf. Section 5.3.2.5) through port *IV* when the first video frame is presented. In location *aVR*, it awaits a signal *VR* from the Video Stream, indicating that the next video frame is ready. When receiving this signal, the Video Manager checks the timing conditions with the Skew Observer through port *Sk*, and with the Jitter Observer through port *J*. If both return *OK* (loop in *aVR*), the next video frame can be presented immediately, which is signalled to the presentation device through port *VP*. If either of the observers returns *wait*, video presentation is too early and needs to be delayed. In this case, the Video Manager moves to location *aVP*, and waits until both observers return *OK*. If at any point, either of the observers returns *error*, the Video Manager moves to its error location *vmE*.

For explanatory purposes, in the sequel we use the terms "presentation of a video frame" and "communication through port *VP*" interchangeably.

5.3.2.3 Jitter Observer



$\{VP, J\}, (J{=}OK), (x_J{\geq}35){\wedge}(x_J{\leq}45), (x_J{:=}0)$
j0
$\{IJ\}, (x_J{:=}0)$
j1
$\{J\}, (J{=}error),$
$(x_J{>}45)$
jE
$\{J\}, (J{=}wait), (x_J{<}35)$
$\{J\}, (J{=}OK), (x_J{\geq}35){\wedge}(x_J{\leq}45)$

Figure 5.25: LSP, Jitter Observer

The Jitter Observer (Figure 5.25) checks that non-anchored jitter on video presentation remains within the acceptable bounds, that means, that every video frame is presented within an interval of [35ms,45ms] after the previous frame. It uses clock x_J to measure the time distance between two subsequent frames. When the Jitter Observer is initialised by the Initialiser (cf. Section 5.3.2.5) through port *IJ* on presentation of the first video frame, it resets its clock to zero, and moves to location *j1*. The Video Manager can now request the current status of jitter—i.e, whether presenting a video frame at the current point in time would conform to the bounds—by communicating with the Jitter Observer through port *J*. Depending on the value of x_J, the Jitter Observer returns either *wait* (lower left loop in *j1*), *OK* (lower right loop and upper loop in *j1*) or *error* (transition from *j1* to *jE*). If the Video Manager communicates through both ports *J* and *VP*, it request the status of jitter and simultaneously sends the order to present a video frame, which is only possible if presentation is acceptable. In this case, the Jitter Observer resets its clock x_J to start the timer for the next frame.

Note that by construction, the Jitter Observer can only enter its error location *jE* if the Video Manager enters its error location *vmE* at the same time.

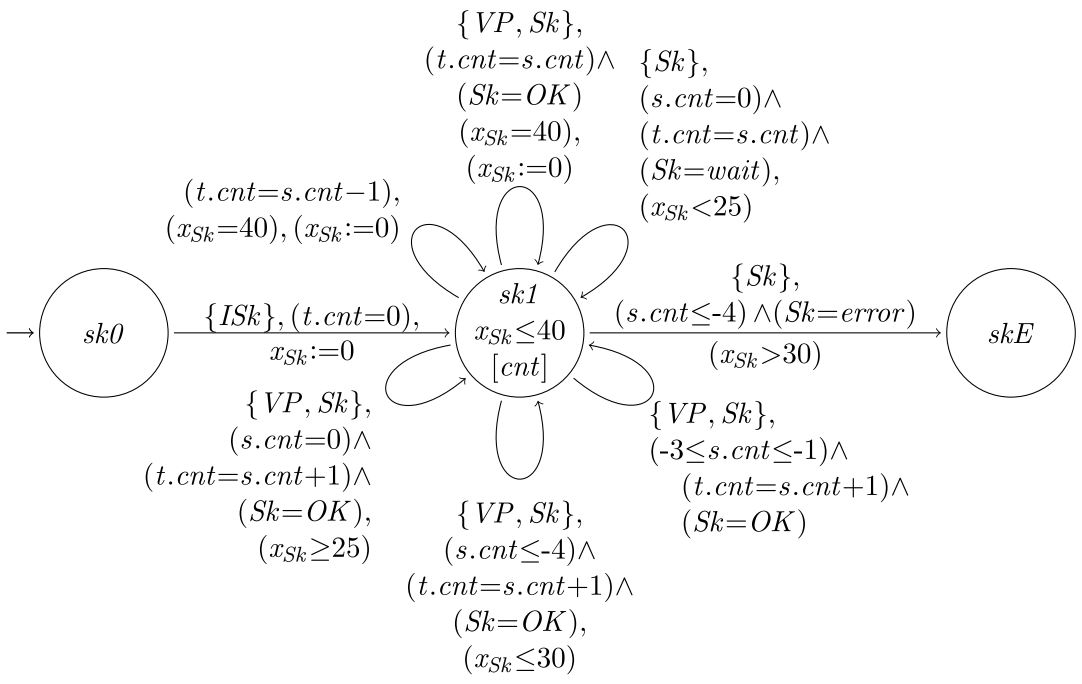

$\{VP, Sk\}$, $(t.cnt{=}s.cnt)\wedge$ $(Sk{=}OK)$ $(x_{Sk}{=}40)$, $(x_{Sk}{:=}0)$
$\{Sk\}$, $(s.cnt{=}0)\wedge$ $(t.cnt{=}s.cnt)\wedge$ $(Sk{=}wait)$, $(x_{Sk}{<}25)$
$(t.cnt{=}s.cnt{-}1)$, $(x_{Sk}{=}40)$, $(x_{Sk}{:=}0)$
sk0
$\{ISk\}$, $(t.cnt{=}0)$, $x_{Sk}{:=}0$
sk1
$x_{Sk}{\le}40$
$[cnt]$
$\{Sk\}$, $(s.cnt{\le}\text{-}4)\wedge(Sk{=}error)$ $(x_{Sk}{>}30)$
skE
$\{VP, Sk\}$, $(s.cnt{=}0)\wedge$ $(t.cnt{=}s.cnt{+}1)\wedge$ $(Sk{=}OK)$, $(x_{Sk}{\ge}25)$
$\{VP, Sk\}$, $(s.cnt{\le}\text{-}4)\wedge$ $(t.cnt{=}s.cnt{+}1)\wedge$ $(Sk{=}OK)$, $(x_{Sk}{\le}30)$
$\{VP, Sk\}$, $(\text{-}3{\le}s.cnt{\le}\text{-}1)\wedge$ $(t.cnt{=}s.cnt{+}1)\wedge$ $(Sk{=}OK)$

Figure 5.26: LSP, Skew Observer

5.3.2.4 Skew Observer

The task of the Skew Observer (Figure 5.26) is to measure the skew (i.e., non-anchored jitter, see above) on video presentation. For this, the Skew Observer uses a clock x_{Sk} and a memory cell *cnt*. Clock x_{Sk} is used to determine the optimal presentation time t_i^{opt} for video frames. The counter *cnt* is used to calculate—together with x_{Sk}—the exact amount of skew at any given point in time. The Skew Observer is initialised by the Initialiser (cf. Section 5.3.2.5) through port *ISk* on presentation of the first sound frame. When this communication occurs, the Skew Observer resets x_{Sk} to zero, sets *cnt* to zero, and moves to location *sk1*.

The general idea of *cnt* can be roughly described as keeping track of the "overflow" of video frame presentations in case the presentation lags behind by more than one period (of 40ms). In detail, this works as follows: every 40ms (measured by x_{Sk}), that means every time a video frame *should* be presented, the value of *cnt* is decreased by one (upper left loop in location *sk1*), and every time a video frame *is* presented, it is increased by one (lower three loops in location *sk1*). In this way, a negative value of *cnt* indicates that $t_i^{act}>t_i^{opt}$ for the last (i-th) video frame, i.e., video lags behind its ideal position; equivalently, a positive value of *cnt* indicates that $t_i^{act}<t_i^{opt}$ for the last (i-th) video frame, i.e., video is ahead of its ideal position. The exact values of clock x_{Sk} and memory cell *cnt* are used to determine *how much* the presentation is ahead of/behind its ideal position. To explain how this works, we look at the following three situations: on presentation of a video frame, *cnt* is (1) incremented to 1, (2) incremented to zero, and (3) incremented to a value <0.

When *cnt* is incremented to 1 on presentation of the i-th video frame at time t_i^{act}, video is ahead of its ideal position (since $cnt{>}0$). This ideal position is at time

t_i^{opt}, which is the next point in time when x_{Sk} reaches 40.[11] The amount of time by which video is ahead of its ideal position at time t_i^{act} is thus given by the difference between the current value of x_{Sk} and 40. Therefore, if a video frame is about to be presented when cnt=0 (i.e., cnt would be set to 1) and x_{Sk}<25, the presentation of the video frame would be more than 15ms (40−25) early, and thus needs to be delayed (upper right loop in location *sk1*). As soon as x_{Sk}≥25, the presentation time is within the acceptable bounds, and the video frame can be presented (lower left loop in location *sk1*). An example for this is depicted in Figure 5.27, showing a situation where each video frame is presented as early as possible: if the fourth video frame would be presented after 140ms, it would be 20ms too early, so the presentation needs to be delayed for at least 5ms.

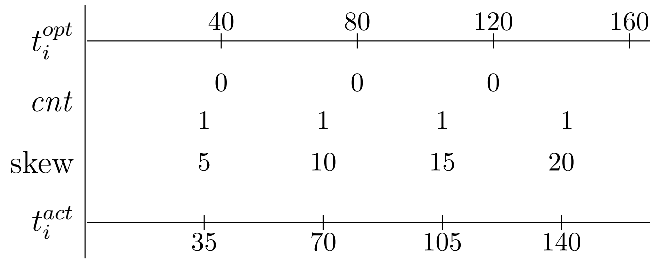


Figure 5.27: Video Presentation ahead: Example

When cnt is incremented to zero on presentation of the i-th video frame at time t_i^{act}, video lags behind its ideal position (since cnt was -1 just before, see above), and the amount by which it lags behind is given by the difference (t_i^{act}−t_i^{opt}), which is equal to the value of x_{Sk} at time t_i^{act}.[12]

Finally, when cnt is incremented to a value <0 on presentation of the i-th video frame at time t_i^{act}, video lags behind its ideal position by more than one period (i.e, more than 40ms). Figure 5.28 shows an illustration of this: if each video frame is presented as late as possible, after 405ms, cnt is incremented to -1, and video is 45ms (more than one period) late by that time. In general, if cnt is incremented to -k, video lags behind its ideal position by k∗40+x_{Sk}. As an example, consider Figure 5.28 again: at time 405, the last time x_{Sk} was reset was at time 400, that means 5ms before, and video presentation lags behind by $k*40 + x_{Sk}=45ms$. The lower loop in location *sk1* thus represents the last possible time where it is still acceptable to present a video frame: on execution of the transition, cnt is incremented to -3, that means video lags behind by -3∗40 + x_{Sk}, and since the clock guard only permits values x_{Sk}≤30, video lags behind by at most 150ms. In case cnt would be set to -3, and x_{Sk}>30 on presentation of a video frame, an out-of-synchronisation error has occurred (transition from location *sk1* to *skE*).

The remaining loops in location *sk1* represent the case where video lags behind, but within the acceptable bounds (lower right loop), and the case where a video

[11]Note that this is indeed t_i^{opt} (and not t_j^{opt}, with j< i), since otherwise cnt would have been set to a value >1.

[12]Note that the last time x_{Sk} was reset was indeed t_i^{opt} (and not t_j^{opt}, with j>i), since otherwise cnt would have been set to a value <0.

frame is presented at an optimal presentation time (upper middle loop), that means $t_i^{act}{=}t_j^{opt}$. Note that apart from $i{=}j$, $i{<}j$ is possible as well in case video presentation lags behind, but $i{>}j$ is not possible due to the bound of 15ms acceptable for video presentation being ahead (cf. Figure 5.27 again).

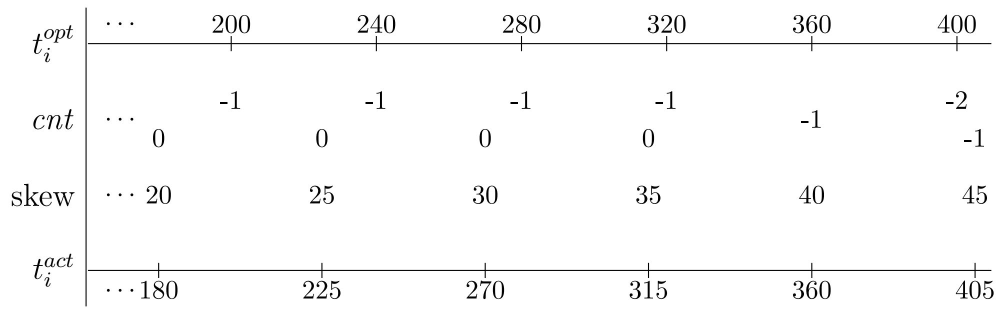


Figure 5.28: Video Presentation behind: Example

Note that by construction, the Skew Observer can only enter its error location *skE* if the Video Manager enters its error location *vmE* at the same time.

5.3.2.5 Initialiser

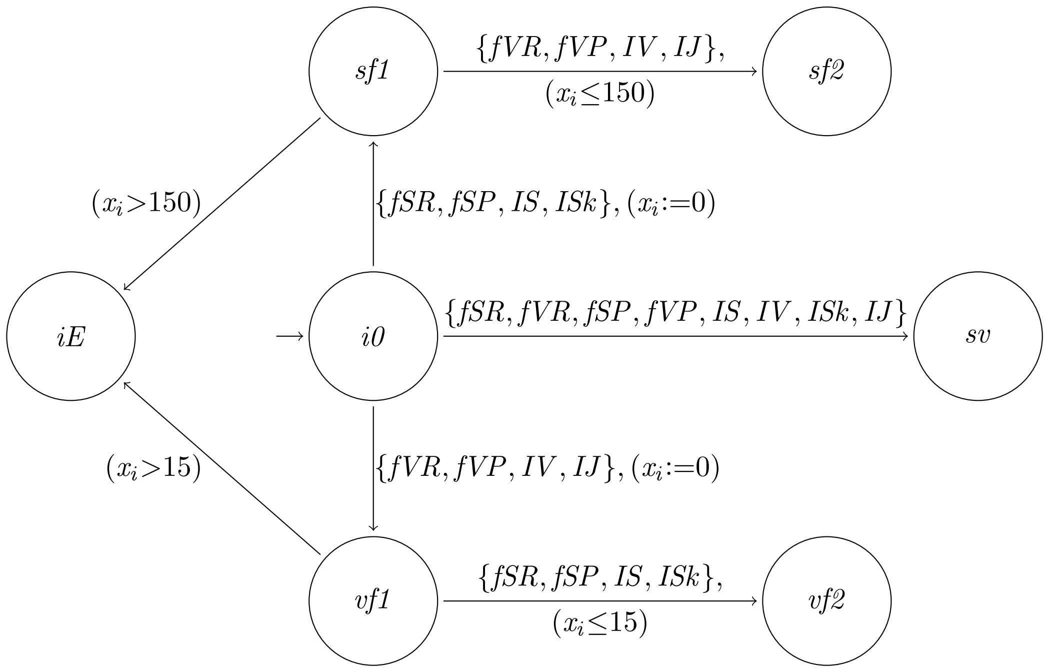

sf1
sf2
$\{fVR, fVP, IV, IJ\}$, $(x_i{\le}150)$
$(x_i{>}150)$
$\{fSR, fSP, IS, ISk\}$, $(x_i{:=}0)$
iE
i0
$\{fSR, fVR, fSP, fVP, IS, IV, ISk, IJ\}$
sv
$(x_i{>}15)$
$\{fVR, fVP, IV, IJ\}$, $(x_i{:=}0)$
vf1
vf2
$\{fSR, fSP, IS, ISk\}$, $(x_i{\le}15)$

Figure 5.29: LSP, Initialiser

The task of the Initialiser (Figure 5.29) is to start the other protocol components in the right order on occurrence of the first frame(s), and to check for initial out-of-synchronisation errors. From its initial location *i0*, there are three options: either a sound frame is ready first (signalled through port *fSR*), in this case, the Initialiser sends to the presentation device the order to present the first sound frame (port *fSP*), starts the Sound Manager (port *IS*) and the Skew Observer (port *ISk*) and a timer x_i, and moves to location *sf1*. If the first video frame is ready (signalled through *fVR*) before the timer reaches 150 (maximum allowed time for video lagging behind), it sends to the presentation device the order to present the first video frame (port *fVP*), and starts the Video Manager (*IV*) and the Jitter Observer (port *IJ*). If x_i reaches 150 and no video frame is ready (initial out-of-synchronisation error), the Initialiser moves to its error location *iE*. In case a video frame arrives first, the behaviour is (mostly) symmetric: the Initialiser moves to location *vf1*, starts the Video Manager and the Jitter Observer, and waits for the first sound frame to be ready. The timeout in this case is 15, which is the maximum amount of time by which the video presentation may precede the sound presentation. The third option from the initial location is the case where both frames (video and sound) arrive at the same time. In this case, all communication (presentation of frames, initialisation of other components) takes place at the same time, and no further check for initial out-of-synchronisation errors is required.

5.3.2.6 Verification

We consider the behaviour of the LSP in different environments, i.e., with different streams. As explained above, the LSP is designed in such a way that no jitter is allowed on the Sound Stream. Consequently, every Sound Stream that allows jitter would immediately cause an out-of-synchronisation error. We therefore only consider a model of an ideal Sound Stream, as shown in Figure 5.30. Every 30ms, the Sound Stream emits a signal that a sound frame is ready, through port *fSR* for the first sound frame, and through port *SR* for all subsequent frames. The extra port for the first sound frame is introduced to ease proper initialisation of the protocol (cf. the Initialiser in Section 5.3.2.5).

For the Video Stream, we consider two different variants, cf. [BFK+98]. Figure 5.31 shows a Video Stream with non-anchored jitter of 5ms, i.e., every video frame is ready at least 35ms and at most 45ms after the previous frame. As a second variant, Figure 5.32 shows a Video Stream with anchored jitter of 5ms, i.e., every video frame is send not earlier than 5ms before its ideal presentation time, and not later than 5ms after its ideal presentation time. As for the Sound Stream, the signal indicating that the first video frame is ready is sent through port *fVR*, for all subsequent frames, it is sent through port *VR*. The TCA modelling the entire system (including the environment) is obtained by composing the TCA $\mathrm{T}_{\mathrm{LSP}}$ (cf. Page 110) with the TCA of the Sound Stream (Figure 5.30) and the TCA of the Video Stream with Anchored Jitter (5.32) or Non-Anchored Jitter (Figure 5.31). For brevity of explanation, we may identify the system by the type of jitter of the included Video Stream, for example, we may refer to the fact that “a property holds in the system including the Video Stream with Anchored Jitter” by simply saying “the property holds in the system with anchored jitter” or “the property holds under anchored jitter”.

To identify out-of-synchronisation errors, we check for reachability of the error locations *sE* in the Sound Manager, *jE* in the Jitter Observer, and *skE* in the Skew Observer component. Note that instead of the latter two, we could also check

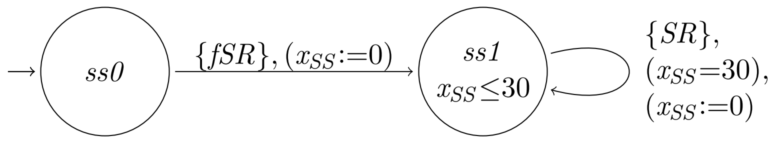


Figure 5.30: LSP, Sound Stream

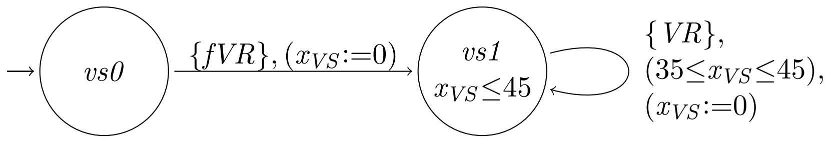


Figure 5.31: LSP, Video Stream, Non-Anchored Jitter

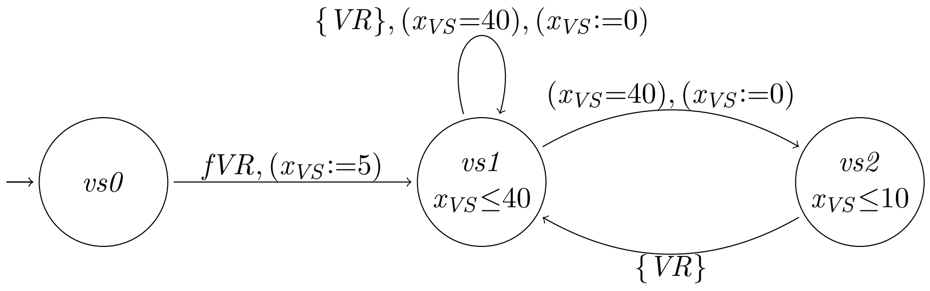


Figure 5.32: LSP, Video Stream, Anchored Jitter

whether the error location *vmE* in the Video Manager component is reachable, since *vmE* is only reachable if either *jE* or *skE* is reachable at the same time, cf. Sections 5.3.2.3 and 5.3.2.4. However, the results would be less meaningful in that case, since we would not be able to identify the exact reason of the error.

For the verification, we further assume that the streams start at the same time, that means the first frames on the two streams are ready at the same time. This is implemented by restricting the runs of the system to those where the Initialiser takes the transition from its initial location *i0* to location *sv* (cf. Section 5.3.2.5). Without this assumption, there can be an arbitrary delay between the first frames on the two streams, and initial out-of-synchronisation errors are trivially possible.

We have modelled the TCA of the LSP with the ECT plugin. As for the ABP (cf. Section 5.3.1.1), we compare three unfolding depths $k \in \{20, 50, 100\}$. Table 5.33 shows the experimental results for the different unfolding depths, error locations and Video Streams (anchored or non-anchored jitter).[13] Again, slanted entries (together with the entry *(S)*) indicate that the input problem is satisfiable. The LSP is modelled in a very concise way already, therefore, we do not provide an abstraction function. In particular, there exists no obvious abstraction function (and probably there exists no non-empty abstraction function at all) that would preserve the results

[13]Note that for k=100, we have added information to the input problem that was obtained from the case k=50, namely that locations *sE* and *skE* are not reachable within 50 steps, and that location *jE* is not reachable within 50 steps under non-anchored jitter. Without this assumption, verification would take much longer for k=100, for example more than 10 hours to detect that location *skE* is not reachable under anchored jitter.

in Table 5.33.

	k=20		k=50		k=100	
	anc.	non-anc.	anc.	non-anc.	anc.	non-anc.
sE	1.145s 22.02MB	0.584s 21.14MB	13.46s 36.77MB	4.171s 32.79MB	43.15s 69.79MB	20.99s 67.22MB
jE	*1.225s (S) 21.91MB*	1.153s 21.67MB	*3.881s (S) 32.98MB*	2.075s 31.31MB	*92.27s (S) 79.06MB*	73.75s 79.90MB
skE	5.250s 22.66MB	4.401s 22.56MB	9.795s 35.22MB	7.769s 33.26MB	1155s 142.44MB	*781.5s (S) 140.05MB*

All experiments have been carried out with MathSAT, version 4.2.17, on an Intel Core2 Duo CPU E4500, with 2.20GHz and 2.5GB RAM

Table 5.33: Experimental Results for the LSP

The first row in Table 5.33 shows that the error location *sE* in the Sound Manager is unreachable under all unfolding depths and for both types of jitter, that means sound frames can never arrive late. This is the expected result, since we have assumed an ideal Sound Stream in Figure 5.30, which does not experience any jitter.

Out-of-synchronisation errors on the Video Stream (jitter) can occur under anchored jitter only, as shown in the second row of Table 5.33. The reason is that under anchored jitter, the maximal distance of two subsequent video frames is 50ms, which is 5ms more than what is allowed by the protocol. The error can occur in case a video frame arrives as early as possible (namely 5ms before its ideal presentation time), and the following video frame arrives as late as possible (namely 5ms after its ideal presentation time). Under non-anchored jitter however, this error is not possible.

In contrast, out-of-synchronisation error between the two streams (skew) can occur under non-anchored jitter only. The reason is that only in this case, the presentation of the video frames can "drift off", that means deviations from the optimal presentation time can sum up, while under anchored jitter, this is not possible. Remember that we allow the video presentation to lag behind by at most 150ms (cf. the beginning of Section 5.3.2). Assuming all video frames arrive as late as possible, i.e., after 45ms, the first time this error can occur is when the protocol has run for 1395ms: after 1350ms, video presentation lags behind by exactly 150ms, and on arrival of the next video frame 45ms later, the limit of 150ms is exceeded. Due to the periodic nature of the Sound Stream and the Video Stream, the protocol needs more than 80 steps to reach this point, which is why the error occurs under unfolding depth 100 only.

As for performance, we make the following observations: comparing the verification times for reachability of different locations, but with the same unfolding depth and Video Streams, it can be seen that checking the reachability of *sE* is mostly faster than checking the reachability of *jE*, which in turn is faster than checking the reachability of *skE*. The reason lies in the complexity of the TCA, and the formula representation generated for them. The transition relation representation (cf. (3.13)) of the Sound Manager contains three elements for source location *s1* (two elements of type (3.11) corresponding to the two visible transitions in location *s1*, and one element of type (3.17) for the delay in location *s1*). For the Jitter Observer, the transition relation representation contains five elements for source location *j1* (four elements of type (3.11), and one element of type (3.17)), and for the Skew Observer, the transition relation representation contains eight elements for source location *sk1* (six elements of type (3.11), one element of type (3.12), and one element of type (3.17)). In particular, the Skew Observer has an invisible transition, which can be executed independently of other TCA (subject to the clock guard being satisfied). Intuitively, this means that the transition relation representation of the Skew Observer is "more nondeterministic" than the transition relation representation of the Jitter Observer, which in turn is more nondeterministic than the transition relation representation of the Sound Manager. Thus, when searching for a system run ending in the respective error location, the MathSAT solver needs to try more options for the Jitter Observer than for the Sound Manager, and in turn more options for the Skew Observer than for the Jitter Observer.

A second observation is the fact that in case the property is satisfied for a certain unfolding depth (i.e., the error location is unreachable) for both types of Video Streams, verification of the system with anchored jitter still takes up to factor 3 longer than verification of the system with non-anchored jitter, though the number of possible executions of a given length in both systems is the same. Again, the reason lies in the complexity of the TCA and the formula representation of the transition relation. For the Video Stream with Non-Anchored Jitter, the transition relation representation contains two elements for source location *vs1*, while for the Video Stream with Anchored Jitter, it contains three element for source location *vs1*, and two more for source location *vs2*.

In contrast to the ABP, where reachable error locations where found comparably fast even for k=100 (less than four seconds, cf. Table 5.21), it takes MathSAT more than 13 minutes to detect that location *skE* is reachable for k=100 (under non-anchored jitter). Equivalently, it takes MathSAT about one and a half minutes to detect that location *jE* is reachable for k=100 (under anchored jitter), which is comparably long given the fact that reachability of location *jE* is detected in a bit more than a second for k=20. The reason is that the LSP is considerably more complex than the ABP. Moreover, even though *jE* is reachable very quickly (in actually less than ten steps), MathSAT still needs to find a valid execution for the remaining steps when checking the system with k=100.

This last point shows the benefits of starting the verification process with comparably small unfolding depths, and successively increasing the bound (cf. Section 3.2). In case the error location is k-step reachable already for small values of k, there is no need to consider larger unfolding depths, since this might take disproportionately longer (in Table 5.33, we have added the results for location *jE* under anchored jitter for k=50 and k=100 to illustrate exactly this point). If, instead, the error location is not k-step reachable for small values of k, this information can be used to reduce verification times in subsequent checks for k'-step reachability (for $k'>k$).

5.3.3 Advantages of using TCA

The case studies presented above have highlighted the advantages of using TCA in the development process. In general, TCA have—just like other automata-based models—a rather shallow learning curve, compared to other formal models like for example process algebra. This is thanks to the intuitive graphic approach, and the state-transition notion in the drawings that closely corresponds to mental models of such systems. This allows for an easy entry into the subject even for unexperienced users, who can quickly design simple models with TCA. One of the main advantages of TCA over other automata-based models is that they allow for true concurrency, while combining the notions of time and data. Thus, once the designer gets accustomed to the formalism, TCA offer a powerful formalism to design concurrent distributed systems.

Comparing our TCA-based models of the ABP and the LSP to models of the protocols found in the literature further illustrates the modelling power of TCA. The model of the ABP presented in [Mil89] is based on the calculus of communicating system (CCS) [Mil82]. The model does not use concrete data values, but only works with the alternating bits. Moreover, the timing for resending of messages is modelled with a timer that only sends a timeout at "some point" after being activated, but the model does not use concrete time values. Our approach allows for concrete time and data values, and therefore provides a more faithful model of the ABP. The process-algebraic model of the ABP presented in [Fok00] handles different data values, but does not include time at all. Instead, components are assumed to emit messages and acknowledgements with bit b periodically until they receive the next message or acknowledgement with bit $\neg b$. Our TCA model provides a more flexible approach, and can be adapted to different environments (where channels have different delays, for example), in that it allows to specify different delays for the resending of the different messages and acknowledgements.

The model of the LSP presented in [BFK$^+$98] is based on TA, and therefore enjoys the general advantages of automata-based models described above. However, we observe a number of advantages when using TCA instead of TA to model the LSP. Remember that TA do not allow for true concurrency, that means only a single event can be executed in every step. As a consequence, the model in [BFK$^+$98] needs a large number of so-called *committed locations*[14] (large compared to the total number of locations) to model the synchronous execution of a set of actions. This becomes particularly evident when considering the initialising component (called "Sync") in [BFK$^+$98]: it is less powerful than the Initialiser presented here (cf. Section 5.3.2.5), in that it does not check for initial out-of-synchronisation errors, yet the "Sync" component needs 11 locations to ensure certain actions happen at the same time, while the Initialiser needs only seven. Even more, the model in [BFK$^+$98] needs two additional helper components "Video Manager" and "Sound Manager" (not to be confused with our Sound Manager and Video Manager), whose only purpose is to ensure that certain (sequences of) events in different automata are executed at the

[14]The notion of *committed location* is taken from UPPAAL [upp], which is the solver used for protocol verification in [BFK$^+$98]. A committed location in a TA must be left immediately after it has been entered, without delay or interleaving of other (instantaneous) actions.

same time. In contrast, with TCA, we simply put all events that are to be executed at the same time on a single transition.

To detect a negative delay on the Video Stream (that means, video lagging behind), [BFK^{+}98] uses a helper component “Sound Clock”. This component models a “backward running” clock, by decreasing a variable by one every time the value of its clock has increased by one. As a consequence, the maximal delay of the system between two subsequent steps is 1, which results in very long execution traces. In our model, we use a memory cell to keep track of the negative delay. This considerable increases the “step length” of the system, since we only need to update the memory cell on optimal presentation times of video frames (i.e, every 40 time units), and on actual presentation of a video frame (cf. Figure 5.26 and Section 5.3.2.4).

Chapter 6

Conclusions

Component connectors that implement real-time coordination patterns are an essential ingredient of component-based software engineering. They are needed to connect, coordinate and orchestrate the distributed components of large real-time systems, and in this way guarantee correct and safe behaviour of the entire system. Adaptation (and extension) of such systems to different needs is facilitated by the encapsulated modular nature of the component connectors, which allows to replace a coordination pattern by another, unnoticed by the other parts of the system. In this thesis, we have established a formal framework for exhaustive modelling and analysis of real-time coordination patterns, with a focus on the formal model of Timed Constraint Automata (TCA) with Data.

In Chapter 2, we have started with the formal definition of the system models, intended to be used to model the real-time coordination patterns. Each of the models is suited to model certain classes of real-time coordination patterns, depending on whether the pattern takes into account data, true concurrency of actions or environmental constraints. We present the models with increasing complexity and modelling power based on these features: Timed Automata (TA) [AD94, Alu99] are well-known (and well-studied), but lack the ability to handle data and true concurrency. As a consequence, we have extended the formal model of TCA [ABdBR07, Kem11] (handling true concurrency) with memory cells to handle data. To incorporate environmental constraints (for example, whether the environment in which a system operates is ready to communicate), we have finally introduced the formal model of Timed Network Automata (TNA) with memory cells and data (which allow for true concurrency as well) as an extension of the model presented in [Kem10]. The intuitive visual representation of each of the models (corresponding to a certain extend to mental models of such systems) allows to quickly and easily sketch and develop systems with the formalism.

In Chapter 3, we define a representation in propositional logic with linear rational arithmetic for each of the system models from Chapter 2. We try to keep the representation as general as possible, by using only those variable types and operations that are compulsory to faithfully represent the underlying formal model. In this way, the representation can serve as an intermediate format and can be translated into the input language of many common SMT-solvers, which in turn can be used to analyse the underlying system. We have discussed how to apply the technique of Bounded Model Checking to the formula representation, and have presented correctness and completeness results.

A major challenge in component-based software engineering is the fact that the systems to be analysed are getting bigger and more complex, while correctness and safety still need to be guaranteed. In Chapter 4, we have presented an approach to (partially) solve this problem: we have defined an abstraction function that can be used to reduce the size of the system representation. The abstraction function works on the formula representation from Chapter 3, and removes parts that are considered irrelevant to the verification of a particular property. We have shown how to undo part of the abstraction in case it has turned out to be too coarse, based on information obtained from so-called spurious counterexamples (counterexamples to the property under test that only exist in the abstract system). We have provided correctness results proving the abstract system to be an over-approximation of the original system.

Theoretical results can only be beneficial if they can be used and applied in practice. In Chapter 5, we have presented in detail the available tool support for the theoretical framework developed in the preceding Chapters (as of now, the implementation only supports the formal model of TCA). Our implementation of the TCA plugin is part of the Extensible Coordination Tools [ECT], an integrated graphical development environment for the Eclipse [Ecl] platform that supports modelling and analysis of component-based systems. We have provided detailed information about the implementation structure of both our plugin and the ECT in general, and we have sketched the typical workflow when using the tool. In this way, Chapter 5 can be seen as a "getting started" tutorial for users, and at the same time as a reference manual for developers. In the end of the Chapter, we have presented two case studies (including experimental results), to show how TCA can be used to model protocol coordination in a concise and understandable way, and we have discussed the benefits of using TCA, for the case studies in particular and for real-time coordination patterns in general.

6.1 Future Directions

In parts of this thesis, some questions have remained unaddressed. For other parts, we see obvious directions for future research and extensions of the work. We now give an overview of the most important points.

In Section 5.3.3, we have discussed the disadvantages of TA when it comes to concurrent execution of actions, and the advantages of TCA regarding this aspect. However, there are situations where a notion similar to committed locations in TA (remember that a committed location in a TA must be left immediately, without delay or interleaving of other actions, cf. Footnote 14 on Page 120) could be beneficial also for TCA: suppose a committed location in a TA, with n ingoing transitions and m outgoing transitions, all with disjoint transition labels. To model the same behaviour with a TCA, we would need $n{*}m$ transitions (while the TA only has $n{+}m$ transitions and one location). We consider it interesting to investigate how a notion similar to committed locations can be defined for TCA, and whether there is a noticeable improvement on performance when it comes to verification.

The model checker Vereofy (`http://www.vereofy.de`) provides tools for model checking (untimed) Constraint Automata (CA). As input, it accepts amongst others CARML (Constraint Automata Reactive Module Language [BBKK09]), which is a textual description language for CA. Extending CARML with the notion of time would yield a textual description language for TCA, which for example could be used as an exchange format for TCA specifications between different tools.

Untimed CA where originally defined as a semantical model for the channel-based coordination language $\mathcal{R}$eo. Consequently, TCA were defined to serve as a semantical model for timed $\mathcal{R}$eo, yet TCA and timed $\mathcal{R}$eo have by now to a certain degree evolved independently of each other. In particular, timed $\mathcal{R}$eo is still restricted to the definitions presented in [ABdBR07]: timed behaviour is incorporated in $\mathcal{R}$eo through special timer channels. These accept any (data) type of input, delay for a certain amount of time, and then emit a special “timeout” signal. In addition, some timer channels can be stopped and/or restarted. It is obvious to see that TCA are more powerful. However, now that syntax and semantics for TCA with data have been defined formally, it should be straightforwardly possible to establish the link between TCA and timed $\mathcal{R}$eo. This of course includes extending the implementation as well: in ECT, timer channels for $\mathcal{R}$eo are already supported, and a translation from $\mathcal{R}$eo to CA already exists. Similar to this translation, a translation from timed $\mathcal{R}$eo to TCA can be implemented once the formal basis for this has been established.

The time that is needed for verification may be a bottleneck for very large systems. For this reason, we consider it worth to investigate how other SAT and SMT solvers perform on our formula generation. A drawback however is the fact that at the moment, there exist only very few SMT solvers that can generate interpolants for unsatisfiable SMT problems. For this reason, it might be worth to decouple interpolant generation from SMT solving. A tailored (re)implementation of interpolant generation would allow to optimise the generation algorithm such that it always generates the strongest interpolant (the notion of *the* strongest interpolant is well-defined, and the strongest interpolant can be computed iteratively, cf. for example [EKS06]). Another approach to decrease verification times is to investigate how the formula order of the input problem influences the performance of different SMT solvers (for example, whether variable valuations of the witness run occur at the beginning or at the end of the input sequence, or are distributed throughout the sequence).

We expect the largest potential for future research in the field of abstraction refinement. A straightforward extension of the work presented in this thesis is to extend the abstraction function to TNA. Assuming that ports in TNA can be merged just like ports in TCA, questions that need to be solved before an abstraction function for TNA can be defined include (1) how to define the colouring of merged ports, and (2) whether read and write ports can be merged, or (if this should not be possible) how to avoid this. From a more practical point of view, for most parts of the formula representation we expect the abstraction function for TNA to work in the same way as it does for TCA. The only exception to this are port colour variables: the explanations after Definition 4.1.5 (positive propositional variables are used to describe the behaviour, negative propositional variables are used to ensure consistency) do not apply to port colour variables, which means they cannot be handled in the same, uniform way (only based on syntactical categories) as other variables. A possibility to solve this is to impose additional constraints on the set of omission and map of merging for TNA, in a similar way as has been done for data constraints involving merged ports in TCA (cf. Definition 4.1.3).

Apart from extending the abstraction function to other system types (like TNA), it is also possible to extend it with other concepts of abstraction. By concept, we mean the way how information is removed from the system. The abstraction function currently features two concepts: merging and omission. Examples of other concepts include weakening, and (variants of) abstract interpretation. By weakening, we mean to replace a constraint by a weaker variant instead of completely removing it. For example, replace a constraint $x{<}5$ by $x{<}10$ (note that convexity is required for this to work for compound constraints). We expect this concept to be applicable to clock constraints. In abstract interpretation (cf. for example [CC77, CC92]), a large (possibly infinite) set of concrete values is restricted to a smaller, finite set of abstract values, using a suitable abstraction function. For example, the set $\mathbb{Z}$ of integer numbers could be abstracted to the set $\{\text{-}1, 0, 1\}$, by mapping all negative integers to -1, all positive integers to 1, and 0 to 0. Abstract interpretation resembles our idea of abstraction by merging, and we expect it to be applicable to the domain of data values of TCA and TNA.

Finally, a question that has not been addressed in this thesis is how to determine the initial abstraction automatically, and how to automate the refinement step. We rely on human experts to provide the initial abstraction and decide which parameter to refine. As an intermediate step towards automatic abstraction refinement, the initial abstraction could be determined automatically based on the property to be verified. A related approach can be found in [CGKS02, CCK$^+$02], where the initial abstraction leaves the parameters contained in the property in the system, and removes all other parameters. In [MA03], the initial abstraction is obtained from a proof of unsatisfiability (in the original system, for some small unfolding depth). We believe these approaches can be adapted to the work presented in this thesis, but it should be one of the major goals for future research to aim towards a framework for fully automatic abstraction refinement.

Appendix A

Proofs

A.1 Correctness of Representation

In this Section, we prove that the formula representation $\varphi(\mathfrak{S})$ of a real-time system $\mathfrak{S}$, as presented in Definitions 3.1.1, 3.1.4 and 3.1.9 in Chapter 3, is correct, that means that $\varphi(\mathfrak{S})$ exhibits the same behaviour as $\mathfrak{S}$. For this, every model of $\varphi(\mathfrak{S})_k$ has to correspond to a run of length k, and vice versa. To prove this, we show that the diagram in Figure A.1 commutes.

$$\begin{tikzcd}
\mathfrak{S} \arrow[r, "\varphi"] \arrow[d, "\text{run}"'] & \varphi(\mathfrak{S})_k \arrow[d, "\text{model}"] \\
Run_{\mathfrak{S},k} \arrow[r, bend left, "\downarrow^r_\sigma"] & \text{models of } \varphi(\mathfrak{S})_k \arrow[l, bend left, "\downarrow^\sigma_r"]
\end{tikzcd}$$

Figure A.1: Correctness of Representation

The commutative property expresses that models of $\varphi(\mathfrak{S})_k$ have a bijective correspondence to runs of the original system $\mathfrak{S}$, denoted by the maps $\downarrow^r_\sigma$ and $\downarrow^\sigma_r$: the run $\downarrow^\sigma_r(\downarrow^r_\sigma(r))$ of the model $\downarrow^r_\sigma(r)$ belonging to a run r again is r, and the model $\downarrow^r_\sigma(\downarrow^\sigma_r(\sigma))$ of a run $\downarrow^\sigma_r(\sigma)$ belonging to a model σ again is σ.

Remark A.1.1 (Notation). In the sequel, we use the notation of representation variables introduced in Section 3.1.1, and we use the symbol $\sim$ to refer to any arithmetic comparison (cf. Definitions 2.1.2 and 2.1.7).

As before, we use $\mathfrak{S}_\mathfrak{S}$ to refer to the associated LTS of a system $\mathfrak{S}$, with $\mathfrak{S}\in\{\mathfrak{A}, \mathfrak{T}, \mathfrak{N}\}$, and we use $Run_{\mathfrak{S},k}$ to refer to the set of all runs of $\mathfrak{S}_\mathfrak{S}$ up to length k.

Further, we use the symbols $\varphi(\mathfrak{S})_k$, σ and $\mathcal{V}(\varphi(\mathfrak{S})_k)$ to refer to the k-unfolding of $\mathfrak{S}$, a model of $\varphi(\mathfrak{S})_k$, and the set of all models of $\varphi(\mathfrak{S})_k$, respectively.

We first show that the formula representation is sound, i.e., that every model $\sigma{\in}\mathcal{V}(\varphi(\mathfrak{S})_k)$ yields a run $r{\in}\mathit{Run}_{\mathfrak{S},k}$.

Definition A.1.2 (Derived Run). For $\sigma{\in}\mathcal{V}(\varphi(\mathfrak{S})_k)$, the *derived run* r_σ is

$$\begin{aligned}
&r_\sigma{=}\langle l_0,\nu_0\rangle\xrightarrow{a_1}\langle l_1,\nu_1\rangle\xrightarrow{a_2}\dots\xrightarrow{a_k}\langle l_k,\nu_k\rangle, && \text{if } \mathfrak{S}=\mathfrak{A},\\
&r_\sigma{=}\langle l_0,\delta_0,\nu_0\rangle\xrightarrow{P_1,\bar\delta_1,t_1}\langle l_1,\delta_1,\nu_1\rangle\xrightarrow{P_2,\bar\delta_2,t_2}\dots\xrightarrow{P_k,\bar\delta_k,t_k}\langle l_k,\delta_k,\nu_k\rangle, && \text{if } \mathfrak{S}=\mathfrak{T},\\
&r_\sigma{=}\langle l_0,\delta_0,\nu_0\rangle\xrightarrow{\mathbb{c}_1,\gamma_1}\langle l_1,\delta_1,\nu_1\rangle\xrightarrow{\mathbb{c}_2,\gamma_2}\dots\xrightarrow{\mathbb{c}_k,\gamma_k}\langle l_k,\delta_k\nu_k\rangle, && \text{if } \mathfrak{S}=\mathfrak{N},
\end{aligned}\tag{i}$$

where we have for all $0{\le}k_0{\le}k$, $1{\le}k_1{\le}k$

$$l_{k_0}=s,\ \text{ iff } \sigma(\mathtt{s}_{\mathtt{k_0}}){=}\mathtt{tt},\tag{ii}$$

$$\nu_{k_0}(x)=\sigma(\mathtt{z}_{\mathtt{k_0}}){-}\sigma(\mathtt{x}_{\mathtt{k_0}})\tag{iii}$$

$$a_{k_1}=\begin{cases}\mathfrak{a},\ \text{iff } \sigma(\alpha_{\mathtt{k_1}}){=}\mathtt{tt},\\ t,\ \text{with } t{=}\sigma(\mathtt{z}_{\mathtt{k_1}}){-}\sigma(\mathtt{z}_{\mathtt{k_1-1}}),\ \text{otherwise}\end{cases}\tag{iv}$$

$$P_{k_1}=\bigcup_{\sigma(\mathtt{p}_{\mathtt{k_1}})=\mathtt{tt}} p,\tag{v}$$

$$\delta_{k_0}(d)=\begin{cases}\Delta^{-1}(\mathtt{n}^{\mathtt{i}}),\ \text{iff } \sigma(\mathtt{Dd}_{\mathtt{k_0}}){=}\mathtt{n}^{\mathtt{i}}{\neq}\mathtt{n}^{\perp}\\ \perp,\ \text{otherwise}\end{cases},d{\in}\mathcal{D}\tag{vi}$$

$$\bar\delta_{k_1}(q)=\begin{cases}\delta(m)_{k_1-1},\ \text{iff } q{=}s.m, m{\in}\mathcal{D}\\ \delta(m)_{k_1},\ \text{iff } q{=}t.m \text{ or } q{=}m, m{\in}\mathcal{D}\\ \Delta^{-1}(\mathtt{n}^{\mathtt{i}}),\ \text{iff } q{=}p, p{\in}\mathcal{P}, \sigma(\mathtt{Dp}_{\mathtt{k_1}}){=}\mathtt{n}^{\mathtt{i}}{\neq}\mathtt{n}^{\perp}\\ \perp,\ \text{otherwise}\end{cases}\tag{vii}$$

$$t_{k_1}=\sigma(\mathtt{z}_{\mathtt{k_1}}){-}\sigma(\mathtt{z}_{\mathtt{k_1-1}})\tag{viii}$$

$$\mathbb{c}_{k_1}(p)=\begin{cases}\text{——}\ \text{iff } \sigma(\mathtt{p}_{\mathtt{k_1}}){=}\mathtt{tt},\\ \text{-?-}\ \text{iff } \sigma(\mathtt{p}_{\mathtt{k_1}}){=}\mathtt{ff} \text{ and } \sigma(\mathtt{cp}_{\mathtt{k_1}}){=}\mathtt{ff}\\ \text{-!-}\ \text{iff } \sigma(\mathtt{p}_{\mathtt{k_1}}){=}\mathtt{ff} \text{ and } \sigma(\mathtt{cp}_{\mathtt{k_1}}){=}\mathtt{tt}\end{cases}\tag{ix}$$

$$\gamma_{k_1}=\begin{cases}\bar\delta_{k_1},\ \text{iff } \sigma(\mathtt{z}_{\mathtt{k_1}}){=}\sigma(\mathtt{z}_{\mathtt{k_1-1}})\\ t_{k_1},\ \text{otherwise}\end{cases}\tag{x}$$

The derived run for a products, that means for $\sigma{\in}\mathcal{V}(\varphi(\mathfrak{S}_1{\bowtie}\mathfrak{S}_2)_k)$ is defined in the same way, except for replacing l_i in (i) by $(l_{i,1},l_{i,2})$, and rewriting (ii) to

$$l_{k_0,i}=s,\qquad \text{iff } \sigma(\mathtt{s}_{\mathtt{k_0}})=\mathtt{tt} \text{ for } s{\in}S_i, i{=}1,2,\tag{ii'}$$

Remark A.1.3 (Derived Run). Note that in (ii), for each k_0, there exists exactly one location s such that $\sigma(\mathtt{s}_{\mathtt{k_0}}){=}\mathtt{tt}$, cf. (3.5), (3.14) and (3.24). Similarly, in (iv), there exists at most one event $\mathfrak{a}$ such that $\sigma(\alpha_{\mathtt{k_1}}){=}\mathtt{tt}$, cf. (3.6).

Since Δ is injective (cf. Section 3.1.1.3), there exists a $d_i{\in}\mathcal{D}\mathit{ata}$ with $\Delta(d_i){=}\mathtt{n}^{\mathtt{i}}$, such that $\Delta^{-1}(\mathtt{n}^{\mathtt{i}})$ is well-defined in (vii), (vi) and (x).

Lemma A.1.4 (Soundness). For $\sigma{\in}\mathcal{V}(\varphi(\mathfrak{S})_k)$, the derived run r_σ is a run of $\mathfrak{S}_\mathfrak{S}$ of length k, i.e., $r_\sigma{\in}Run_{\mathfrak{S},k}$.

Proof. Induction on k.

IA $\sigma{\models}\varphi(\mathfrak{S})_0$: $\sigma(\mathtt{\bar{s}_0})\overset{(3.1),(3.10),(3.20)}{=}\mathtt{tt}$ for the initial location $\bar{s}$, thus $l_0\overset{\text{(ii)}}{=}\bar{s}$. For all clocks x, $\nu_0(x)\overset{\text{(iii)}}{=}\sigma(\mathtt{z_0}){-}\sigma(\mathtt{x_0})\overset{(3.1),(3.10),(3.20)}{=}0$, therefore in particular $\nu_0{\models}I(\bar{s})$, and thus $r_\sigma{=}\langle\bar{s},\mathbf{0}\rangle{\in}Run_{\mathfrak{S},0}$ for $\mathfrak{S}{=}\mathfrak{A}$.

For $\mathfrak{S}{\neq}\mathfrak{A}$, in addition we have $\sigma(\mathtt{Dd_0})\overset{(3.10),(3.20)}{=}\mathtt{n}^\perp$ for $d{\in}\mathcal{D}$, and $\sigma(\mathtt{Dp_0})\overset{(3.10),(3.20)}{=}\mathtt{n}^\perp$ for $p{\in}\mathcal{P}$, thus $r_\sigma{=}\langle\bar{s},\mathbf{0},\mathbf{0}\rangle{\in}Run_{\mathfrak{S},0}$ for $\mathfrak{S}{\in}\{\mathfrak{T},\mathfrak{N}\}$.

IH $\sigma{\models}\varphi(\mathfrak{S})_k$: $r_\sigma{\in}Run_{\mathfrak{S},k}$ for some $k{\geq}0$.

IS $\sigma{\models}\varphi(\mathfrak{S})_{k+1}$: we consider the different systems separately

$\mathfrak{S} = \mathfrak{A}$**:** $r_\sigma{=}\langle l_0,\nu_0\rangle\xrightarrow{a_1}\ldots\xrightarrow{a_k}\langle l_k,\nu_k\rangle\xrightarrow{a_{k+1}}\langle l_{k+1},\nu_{k+1}\rangle$, and either for some $e{\in}E$ $\sigma{\models}\varphi^{action}(e)_{k+1/t}$, or $\sigma{\models}\varphi^{delay}(s)_{k+1/t}$ for some $s{\in}S$ (cf. (3.7) and (3.29)).

Case $\sigma{\models}\varphi^{action}(e)_{k+1/t}$ $(*)$: let $e{=}(s,\mathfrak{a},cc,\lambda,s')$ (cf. (3.2)), then

- $l_k\overset{\text{IH}}{=}s$, $l_{k+1}\overset{\text{(ii)}}{=}s'$
- $a_{k+1}\overset{\text{(iv)}}{=}\mathfrak{a}$
- $\nu_{k+1}{=}\nu_k[\lambda]$: for all clocks x, we have
  $\lambda(x){=}id$: $\nu_{k+1}(x)\overset{\text{(iii)}}{=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_{k+1}})\overset{(*)}{=}\sigma(\mathtt{z_k}){-}\sigma(\mathtt{x_k})\overset{\text{(iii),IH}}{=}\nu_k(x)$
  $\lambda(x){=}x'$: $\nu_{k+1}(x)\overset{\text{(iii)}}{=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_{k+1}})\overset{(*)}{=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x'_{k+1}})\overset{\text{(iii)}}{=}\nu_{k+1}(x')$
  $\lambda(x){=}n$: $\nu_{k+1}(x)\overset{\text{(iii)}}{=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_{k+1}})\overset{(*)}{=}\sigma(\mathtt{z_{k+1}}){-}(\sigma(\mathtt{z_{k+1}}){-}n){=}n$
- $\nu_k{\models}cc$: for $cc{=}x{\sim}n$,[1] we have $\mathtt{cc_k}{=}(\mathtt{z_k}{-}\mathtt{x_k}){\sim}n$. Because of $(*)$, we have $\sigma{\models}\mathtt{cc_k}$, that means $(\sigma(\mathtt{z_k}){-}\sigma(\mathtt{x_k})){\sim}n$ holds, and since $\nu_k(x)\overset{\text{IH,(iii)}}{=}(\sigma(\mathtt{z_k}){-}\sigma(\mathtt{x_k}))$ for all clocks x, we have $\nu_k{\models}cc$. The argumentation for $\nu_{k+1}{\models}I(s')$ is similar

Thus, $\langle s,\nu_k\rangle\xrightarrow{\mathfrak{a}}\langle s',\nu_k[\lambda]\rangle$ is gained from e using (2.1)

Case $\sigma{\models}\varphi^{delay}(s)_{k+1/t}$ $(**)$: let $s{\in}S$ (cf. (3.3)), then

- $l_k\overset{\text{IH}}{=}s$, $l_{k+1}\overset{\text{(ii)}}{=}s$
- $a_{k+1}{=}t$, with $t\overset{\text{(iv)}}{=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{z_k})$
- $\nu_{k+1}{=}\nu_k{+}t$: for all clocks x, we have
  $\nu_{k+1}(x)\overset{\text{(iv)}}{=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_{k+1}})\overset{(**)}{=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_k}){=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_k}){+}$
  $\sigma(\mathtt{z_k}){-}\sigma(\mathtt{z_k}){=}(\sigma(\mathtt{z_k}){-}\sigma(\mathtt{x_k})){+}(\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{z_k}))\overset{\text{(iii),(iv),IH}}{=}\nu_k(x){+}t$
- $\nu_{k+1}{\models}I(s)$ as above, $\nu_k{+}t'{\models}I(s)$ for all $t'{\leq}t$ because of convexity (cf. Remark 2.1.6)

Thus, $\langle s,\nu_k\rangle\xrightarrow{t}\langle s,\nu_k{+}t\rangle$ is gained from s using (2.2).

Together, we get $r_\sigma{\in}Run_{\mathfrak{S},k+1}$ for $\mathfrak{S}{=}\mathfrak{A}$.

[1]Here and in the remainder of the proof, we only show the basic cases for simple clock constraints (without clock differences) and simple data constraints (without addition/subtraction), but the results directly carry over to constraints involving arithmetic operations, and to Boolean combinations of these.

$\mathfrak{S} = \mathfrak{T}$**:** $r_\sigma{=}\langle l_0, \delta_0,\nu_0\rangle \xrightarrow{P_1,\bar{\delta}_1,t_1} \dots \xrightarrow{P_k,\bar{\delta}_k,t_k} \langle l_k, \delta_k, \nu_k\rangle \xrightarrow{P_{k+1},\bar{\delta}_{k+1},t_{k+1}} \langle l_{k+1}, \delta_{k+1},\nu_{k+1}\rangle$, and either $\sigma{\models}\varphi^{visible}(e)_{k+1/t}$ or $\sigma{\models}\varphi^{invisible}(e)_{k+1/t}$ for some $e{\in}E$ (cf. (3.16) and (3.29)).

Case $\sigma{\models}\varphi^{visible}(e)_{k+1/t}$ (†): let $e{=}(s, P, dc, cc, \lambda, s')$ (cf. (3.11)), then

- $l_k \overset{\text{IH}}{=} s$, $l_{k+1} \overset{\text{(ii)}}{=} s'$, $t_{k+1} \overset{\text{(viii)}}{=} \sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{z_k})$
- $\nu_{k+1}{=}\nu_k{+}t_{k+1}[\lambda]$: for all clocks x, we have
  $\lambda(x){=}id$: $\nu_{k+1}(x) \overset{\text{(iii)}}{=} \sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_{k+1}}) \overset{(\dagger)}{=} \sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_k}){=}$
  $\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_k}){+}\sigma(\mathtt{z_k}){-}\sigma(\mathtt{z_k}){=}$
  $(\sigma(\mathtt{z_k}){-}\sigma(\mathtt{x_k})){+}(\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{z_k})) \overset{\text{(iii),(viii),IH}}{=} \nu_k(x){+}t_{k+1}$
  $\lambda(x){=}x'$: $\nu_{k+1}(x) \overset{\text{(iii)}}{=} \sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_{k+1}}) \overset{\dagger}{=} \sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x'_{k+1}}){=}$
  $\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x'_{k+1}}){+}\sigma(\mathtt{z_k}){-}\sigma(\mathtt{z_k}){=}$
  $(\sigma(\mathtt{z_k}){-}\sigma(\mathtt{x'_{k+1}})){+}(\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{z_k})) \overset{\text{(iii),(viii),IH}}{=} \nu_{k+1}(x'){+}t_{k+1}$
  $\lambda(x){=}n$: $\nu_{k+1}(x) \overset{\text{(iii)}}{=} \sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_{k+1}}) \overset{(\dagger)}{=} \sigma(\mathtt{z_{k+1}}){-}(\sigma(\mathtt{z_{k+1}}){-}n){=}n$
- $\nu_k{+}t_{k+1}{\models}cc$: for $cc{=}x{\sim}n$, we have $\mathtt{cc_k}{=}(\mathtt{z_k}{-}\mathtt{x_k}){\sim}n$. Because of (†), we have $\sigma{\models}\mathtt{cc_k}$, that means $(\sigma(\mathtt{z_k}){-}\sigma(\mathtt{x_k})){\sim}n$ holds, and since $\nu_k(x){+}t_{k+1} \overset{\text{IH,(iii),(viii)}}{=} \sigma(\mathtt{z_k}){-}\sigma(\mathtt{x_k}){+}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{z_k}){=}\sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{x_k})$ for all clocks x, we have $\nu_k{+}t_{k+1}{\models}cc$.
- $P_{k+1} \overset{\text{(v)}}{=} \bigcup_{\sigma(\mathtt{p_{k+1}})=\mathtt{tt}} p \overset{(\dagger)}{=} P$
- $\bar{\delta}_{k+1}{\models}dc$: let $dc{=}(\mathbf{D}{\simeq}\mathbf{D}')$, $\simeq\,\in\{=,\leqslant\}$ (cf. Section 3.1.1.5). For the constituents of dc, we have
  for $p{\in}\mathcal{P}|_{dc}$: $\bar{\delta}_{k+1}(p) \overset{\text{(vii)}}{=} \Delta^{-1}(\mathtt{n^i}) \overset{def.\Delta}{=} d_i$ if $\sigma(\mathtt{Dp_{k+1}}){=}d_i$
  for $s.m{\in}\mathcal{D}|_{dc}$: $\bar{\delta}_{k+1}(s.m) \overset{\text{(vii)}}{=} \Delta^{-1}(\mathtt{n^i}) \overset{def.\Delta}{=} d_i$ if $\sigma(\mathtt{Dm_k}){=}d_i$
  for $t.m{\in}\mathcal{D}|_{dc}$: $\bar{\delta}_{k+1}(t.m) \overset{\text{(vii)}}{=} \Delta^{-1}(\mathtt{n^i}) \overset{def.\Delta}{=} d_i$ if $\sigma(\mathtt{Dm_{k+1}}){=}d_i$
  for $m{\in}\mathcal{D}|_{dc}$: $\bar{\delta}_{k+1}(m) \overset{\text{(vii)}}{=} \Delta^{-1}(\mathtt{n^i}) \overset{def.\Delta}{=} d_i$ if $\sigma(\mathtt{Dm_{k+1}}){=}d_i$
  Because of (†), we have $\sigma{\models}\mathtt{dc_{k+1}}$, thus for all possible instantiations of $\mathbf{D}$ and $\mathbf{D}'$ with $\mathtt{Dp_{k+1}}$, $\mathtt{Dm_k}$, $\mathtt{Dm_{k+1}}$ or elements of $\mathcal{D}ata$ (cf. Section 3.1.1.5), we have $\bar{\delta}_{k+1}{\models}dc$
- $\delta_{k+1}(m){=}\bot$ if $m{\notin}\#(s')$: because of (†), in particular $\sigma{\models}(\mathtt{Dm_{t+1}}{=}\mathtt{n^\bot})$ (cf. (3.11)), and thus $\delta_{k+1}(m) \overset{\text{(vi)}}{=} \bot$ for all $m{\notin}\#(s')$

Thus, $\langle s,\delta_k,\nu_k\rangle \xrightarrow{P,\bar{\delta}_{k+1},t_{k+1}} \langle s',\delta_{k+1},\nu_k{+}t_{k+1}[\lambda]\rangle$ is gained from e using (3.11).

Case $\sigma{\models}\varphi^{invisible}(e)_{k+1/t}$ (‡): let $e{=}(s, \emptyset, dc, cc, \lambda, s')$ (cf. (3.12)), then

- $l_k \overset{\text{IH}}{=} s$, $l_{k+1} \overset{\text{(ii)}}{=} s'$, $t_{k+1} \overset{\text{(viii)}}{=} \sigma(\mathtt{z_{k+1}}){-}\sigma(\mathtt{z_k})$,
- $\nu_{k+1}{=}\nu_k{+}t_{k+1}[\lambda]$, $\nu_k{+}t_{k+1}{\models}cc$, $\delta_{k+1}(m){=}\bot$ if $m{\notin}\#(s')$, and $\bar{\delta}_{k+1}{\models}dc$ as before
- $P_{k+1} \overset{\text{(v)}}{=} \bigcup_{\sigma(\mathtt{p_{k+1}})=\mathtt{tt}} p \overset{(\ddagger)}{=} \emptyset$
- $\delta_{k+1}(p) \overset{\text{(vii)}}{=} \bot$ for all $p{\in}\mathcal{P}$, because $\sigma(\mathtt{p_{k+1}}) \overset{(\ddagger)}{=} \mathtt{ff}$, and $\sigma(\mathtt{Dp_{k+1}}) \overset{(\ddagger)}{=} \mathtt{n^\bot}$ for all $p{\in}\mathcal{P}$

Thus, $\langle s, \delta_k, \nu_k\rangle \xrightarrow{\emptyset,\bar{\delta}_{k+1},t_{k+1}} \langle s', \delta_{k+1}, \nu_k{+}t_{k+1}[\lambda]\rangle$ is gained from e using (3.11) in case $t_{k+1}{>}0$, and using (3.12) in case $t_{k+1}{=}0$.

Together, we get $r_\sigma \in Run_{\mathfrak{S},k+1}$ for $\mathfrak{S}=\mathfrak{T}$.

$\mathfrak{S} = \mathfrak{N}$**:** $r_\sigma = \langle l_0, \delta_0, \nu_0\rangle \xrightarrow{\mathbb{c}_1,\gamma_1} \ldots \xrightarrow{\mathbb{c}_k,\gamma_k} \langle l_k, \delta_k, \nu_k\rangle \xrightarrow{\mathbb{c}_{k+1},\gamma_{k+1}} \langle l_{k+1}, \delta_{k+1}, \nu_{k+1}\rangle$, and either $\sigma \models \varphi^{commu}(e)_{k+1/t}$ or $\sigma \models \varphi^{delay}(e)_{k+1/t}$ for some $e \in E$ (cf. (3.26) and (3.29))

Case $\sigma \models \varphi^{commu}(e)_{k+1/t}$ $(\star)$: let $e=(s, \mathbb{c}, dc, cc, \lambda, s')$ (cf. (3.21)), then

- $l_k \overset{\text{IH}}{=} s$, $l_{k+1} \overset{\text{(ii)}}{=} s'$
- $\nu_{k+1}=\nu_k[\lambda]$, $\nu_k \models cc$, $\nu_{k+1} \models I(s')$: equivalent to the respective cases for $\mathfrak{S}=\mathfrak{A}$ above
- $\delta_{k+1}(m)=\perp$ if $m \notin \#(s')$: equivalent to the respective case for $\mathfrak{S}=\mathfrak{T}$ above
- $\gamma_{k+1} \overset{(x),(\star)}{=} \bar{\delta}_{k+1}$
- $\bar{\delta}_{k+1} \models dc$: equivalent to the respective case for $\mathfrak{S}=\mathfrak{T}$ above
- $\bar{\delta}_{k+1}(p)=\perp$ iff $\mathbb{c}_{k+1}(p) \neq$ ——: because of $(\star)$, in particular $\sigma \models \langle \mathtt{p}_{\mathbb{c}} \rangle_{\mathtt{k+1}}$ for all $p \in \mathcal{P}$. If $\sigma \models \neg \mathtt{p}_{\mathtt{k+1}}$, then either $\mathbb{c}(p) \overset{\text{(ix)}}{=} \texttt{-!-}$ or $\mathbb{c}(p) \overset{\text{(ix)}}{=} \texttt{-?-}$, and $\sigma(\mathtt{Dp}_{\mathtt{k+1}}) \overset{(3.25),(\star)}{=} \mathtt{n}^{\perp}$

Thus, $\langle s, \delta_k, \nu_k\rangle \xrightarrow{\mathbb{c}_{k+1},\bar{\delta}_{k+1}} \langle s, \delta_{k+1}, \nu_k[\lambda]\rangle$ is gained from e using (2.15).

Case $\sigma \models \varphi^{delay}(e)_{k+1/t}$ $(\star\star)$: let $e=(s, \mathbb{c}, dc, cc, id, s)$ (cf. (3.22)), then

- $l_k \overset{\text{IH}}{=} s$, $l_{k+1} \overset{\text{(ii)}}{=} s'$
- $\gamma_{k+1} \overset{(x),(\star\star)}{=} t_{k+1}$
- $\nu_{k+1}=\nu_k+t_{k+1}$: equivalent to the respective case for $\mathfrak{S}=\mathfrak{A}$ above
- $\nu_k \models cc$, $\nu_{k+1} \models I(s)$: equivalent to the respective cases for $\mathfrak{S}=\mathfrak{A}$ above, $\nu_k+t' \models cc$ and $\nu_k+t' \models I(s)$ for all $t' \leq t_{k+1}$ because of convexity (cf. Remark 2.1.6)
- $\delta_k \models dc$: equivalent to the respective case above, with δ_k instead of $\bar{\delta}_{k+1}$

Thus, $\langle s, \delta_k, \nu_k\rangle \xrightarrow{\mathbb{c}_{k+1},t_{k+1}} \langle s, \delta_k, \nu_k+t_{k+1}\rangle$ is gained from e using (2.16).

Together, we get $r_\sigma \in Run_{\mathfrak{S},k+1}$ for $\mathfrak{S}=\mathfrak{N}$.

Finally, we get $r_\sigma \in Run_{\mathfrak{S},k+1}$ for all systems $\mathfrak{S} \in \{\mathfrak{A}, \mathfrak{T}, \mathfrak{N}\}$, and we define the map $\downarrow_r^\sigma : \mathcal{V}(\varphi(\mathfrak{S})_k) \to Run_{\mathfrak{S},k}$ such that for every interpretation $\sigma \in \mathcal{V}(\varphi(\mathfrak{S})_k)$, we have that $\downarrow_r^\sigma(\sigma) = r_\sigma \in Run_{\mathfrak{S},k}$ is the derived run. $\square$

Proposition A.1.5 (Derived Run, Product). For $\sigma \in \mathcal{V}(\varphi(\mathfrak{S}_1 \bowtie \mathfrak{S}_2)_k)$, the derived run r_σ is a run of $\mathfrak{S}_{\mathfrak{T}_1 \bowtie \mathfrak{T}_2}$ of length k, i.e., $r_\sigma \in Run_{\mathfrak{S}_1 \bowtie \mathfrak{S}_2,k}$.

Proof (Idea). The proof is along the same lines as the proof of Lemma A.1.4. In **IS**, we first show that for i=1, 2, reducing a transition of the product $\mathfrak{S}_1 \bowtie \mathfrak{S}_2$ (of the form $\langle (l_{k,1}, l_{k,2}), \nu_k\rangle \xrightarrow{a_k} \langle (l_{k+1,1}, l_{k+1,2} \nu_{k+1}\rangle$ for $\mathfrak{S}=\mathfrak{A}$, for example) to the constituents of $\mathfrak{S}_i$ yields a transition e_i in $\mathfrak{S}_i$. We then argue that all possible combinations of e_1 and e_2 correspond to a valid execution in the product automaton (cf. Definitions 2.2.8, 2.3.9 and 2.4.14). In end, we get $r_\sigma \in Run_{\mathfrak{S}_1 \bowtie \mathfrak{S}_2,k}$. $\square$

We now show that the formula representation is complete, i.e, for every run $r{\in}Run_{\mathfrak{S},k}$, we can find a model $\sigma{\in}\mathcal{V}(\varphi(\mathfrak{S})_k)$.

Definition A.1.6 (Derived Interpretation). For $r{\in}Run_{\mathfrak{S},k}$, the *derived interpretation σ_r over (the variables in) $\varphi(\mathfrak{S})_k$* is (we use the notation of (i))

$$\sigma_r(\mathtt{s}_{\mathtt{k}_0}) = \mathtt{tt},\ \text{iff } s{=}l_{k_0} \tag{xi}$$

$$\sigma_r(\mathtt{z}_{\mathtt{k}_0}) = \begin{cases} 0,\ \text{if } k_0{=}0 \\ \sigma_r(\mathtt{z}_{\mathtt{k}_0-\mathtt{1}}){+}t,\ \text{if } \mathfrak{S}{=}\mathfrak{A} \text{ and } a_{k_0}{=}t \\ \sigma_r(\mathtt{z}_{\mathtt{k}_0-\mathtt{1}}){+}t_{k_0},\ \text{if } (\mathfrak{S}{=}\mathfrak{T}) \text{ or } (\mathfrak{S}{=}\mathfrak{N} \text{ and } \gamma_{k_0}{=}t_{k_0}) \\ \sigma_r(\mathtt{z}_{\mathtt{k}_0-\mathtt{1}}),\ \text{otherwise} \end{cases} \tag{xii}$$

$$\sigma_r(\mathtt{x}_{\mathtt{k}_0}) = \sigma_r(\mathtt{z}_{\mathtt{k}_0}){-}\nu_{k_0}(x) \tag{xiii}$$

$$\sigma_r(\alpha_{k_0}) = \begin{cases} \mathtt{ff},\ \text{if } k_0{=}0 \\ \mathtt{tt},\ \text{iff } a_{k_0}{=}\mathfrak{a} \\ \mathtt{ff},\ \text{otherwise} \end{cases} \tag{xiv}$$

$$\sigma_r(\mathtt{p}_{\mathtt{k}_0}) = \begin{cases} \mathtt{ff},\ \text{if } k_0{=}0 \\ \mathtt{tt},\ \text{if } (p{\in}P_{k_0} \text{ and } \mathfrak{S}{=}\mathfrak{T}) \text{ or } (\mathbb{c}_{k_0}(p){=}\text{——} \text{ and } \mathfrak{S}{=}\mathfrak{N}) \\ \mathtt{ff},\ \text{otherwise} \end{cases} \tag{xv}$$

$$\sigma_r(\mathtt{Dp}_{\mathtt{k}_0}) = \begin{cases} \mathtt{n}^{\perp},\ \text{if } k_0{=}0 \\ \Delta(\bar{\delta}_{k_0}(p)),\ \text{if } (p{\in}P_{k_0} \text{ and } \mathfrak{S}{=}\mathfrak{T}) \text{ or } (\mathbb{c}_{k_0}(p){=}\text{——} \text{ and } \mathfrak{S}{=}\mathfrak{N}) \\ \mathtt{n}^{\perp},\ \text{otherwise} \end{cases} \tag{xvi}$$

$$\sigma_r(\mathtt{Dd}_{\mathtt{k}_0}) = \begin{cases} \mathtt{n}^{\perp},\ \text{if } k_0{=}0 \\ \Delta(\delta_{k_0}(d)),\ \text{if } d{\in}\#(l_{k_0}) \\ \Delta(\bar{\delta}_{k_0}(d)),\ \text{otherwise} \end{cases} \tag{xvii}$$

$$\sigma_r(\mathtt{d}_{\mathtt{k}_0}) = \begin{cases} \mathtt{ff},\ \text{if } \sigma_r(\mathtt{Dd}_{\mathtt{k}_0}){=}\mathtt{n}^{\perp} \\ \mathtt{tt},\ \text{otherwise} \end{cases} \tag{xviii}$$

$$\sigma_r(\mathtt{cp}_{\mathtt{k}_0}) = \begin{cases} \mathtt{tt},\ \text{if } (k_0{=}0) \text{ or } (\mathbb{c}_{k_0}(p){=}\mathtt{-!-}) \\ \mathtt{ff},\ \text{if } \mathbb{c}_{k_0}(p){=}\mathtt{-?-} \\ \text{unspecified, otherwise} \end{cases} \tag{xix}$$

for all $0{\leq}k_0{\leq}k$. The derived interpretation for a run of the product, that means for $r{\in}Run_{\mathfrak{S}_1\bowtie\mathfrak{S}_2,k}$, is defined in the same way, except for rewriting (xi) to

$$\sigma_r(\mathtt{s}_{\mathtt{k}_0}) = \mathtt{tt}, \qquad \text{iff } s{=}l_{k_0,i} \text{ for } s{\in}S_i, i{=}1,2 \tag{xi'}$$

Lemma A.1.7 (Completeness). For $r{\in}Run_{\mathfrak{S},k}$, the derived interpretation σ_r is a model of the k-unfolding of $\mathfrak{S}$, that means $\sigma_r{\models}\varphi(\mathfrak{S})_k$.

Proof. Induction on k.

IA $r{=}\langle l_0, \nu_0\rangle$ respectively $r{=}\langle l_0, \delta_0, \nu_0\rangle$ (cf. (i) again). We have $\sigma_r(\bar{\mathtt{s}}_0)\overset{\text{(xi)}}{=}\mathtt{tt}$ for the initial location $\bar{s}{=}l_0$, and $\sigma_r(\mathtt{s}_0)\overset{\text{(xi)}}{=}\mathtt{ff}$ otherwise. For clocks, we have $\sigma_r(\mathtt{z}_0)\overset{\text{(xii)}}{=}0$, and $\sigma_r(\mathtt{x}_0)\overset{\text{(xiii)}}{=}\sigma_r(\mathtt{z}_0)-\nu_0(x)=0$ for all other clocks x. For $I(\bar{s})=x\sim c$,[2] we have $\mathtt{I}(\bar{\mathtt{s}})_0=\mathtt{z}_0-\mathtt{x}_0\sim n$. Because $\nu_0\models I(\bar{s})$ (cf. Definitions 2.2.4, 2.3.5 and 2.4.6), in particular $0=\nu_0(x)\sim n$, therefore $\sigma_r(\mathtt{x}_0)\sim n$ holds, and thus $\sigma_r\models\mathtt{I}(\bar{\mathtt{s}})_0$.

We have $\sigma_r(\mathtt{p}_0)\overset{\text{(xv)}}{=}\mathtt{ff}$, $\sigma_r(\mathtt{Dp}_0)\overset{\text{(xvi)}}{=}\mathtt{n}^\perp$, and $\sigma_r(\mathtt{cp}_0)\overset{\text{(xix)}}{=}\mathtt{tt}$ for all ports p, and for all data variables d we have $\sigma_r(\mathtt{Dd}_0)\overset{\text{(xvii)}}{=}\mathtt{n}^\perp$ and $\sigma_r(\mathtt{d}_0)\overset{\text{(xviii)}}{=}\mathtt{ff}$.

Thus, $\sigma_r\models\varphi^{init}(\mathfrak{S})$ (cf. (3.1), (3.10) and (3.20)), and therefore $\sigma_r\models\varphi(\mathfrak{S})_0$.

IH $r\in Run_{\mathfrak{S},k}$: $\sigma_r\models\varphi(\mathfrak{S})_k$, for some $k\geq 0$.

IS $r\in Run_{\mathfrak{S},k+1}$: we again consider the different systems separately

$\mathfrak{S}=\mathfrak{A}$: $r=\langle l_0,\nu_0\rangle\xrightarrow{a_1}\dots\xrightarrow{a_k}\langle l_k,\nu_k\rangle\xrightarrow{a_{k+1}}\langle l_{k+1},\nu_{k+1}\rangle\in Run_{\mathfrak{A},k+1}$, and either the last step $\langle l_k,\nu_k\rangle\xrightarrow{a_{k+1}}\langle l_{k+1},\nu_{k+1}\rangle$ is an action transition (2.1) resulting from execution of a transition $e=(s,\mathfrak{a},cc,\lambda,s')$, or it is a delay transition (2.2) in location s.

In case of an action transition, we have $l_k=s$, $l_{k+1}=s'$, $a_{k+1}=\mathfrak{a}$ for some $\mathfrak{a}\in\Sigma$, and $\nu_{k+1}=\nu_k$. Then

- $\sigma_r(\mathtt{s}_{\mathtt{k+1}})\overset{\text{(xi)}}{=}\mathtt{tt}$ for $s=l_{k+1}$, and $\mathtt{ff}$ otherwise
- $\sigma_r(\mathtt{z}_{\mathtt{k+1}})\overset{\text{(xii)}}{=}\sigma_r(\mathtt{z}_\mathtt{k})$
- $\sigma_r(\mathtt{x}_{\mathtt{k+1}})\overset{\text{(xiii)}}{=}\sigma_r(\mathtt{z}_{\mathtt{k+1}})-\nu_{k+1}(x)=\begin{cases}\overset{\lambda(x)=id,\text{(xii)}}{=}\sigma_r(\mathtt{z}_\mathtt{k})-\nu_k(x)=\sigma_r(\mathtt{x}_\mathtt{k})\\ \overset{\lambda(x)=x'}{=}\sigma_r(\mathtt{z}_{\mathtt{k+1}})-\nu_{k+1}(x')\overset{\text{(xiii)}}{=}\sigma_r(\mathtt{x}'_{\mathtt{k+1}})\\ \overset{\lambda(x)=n}{=}\sigma_r(\mathtt{z}_{\mathtt{k+1}})-n\end{cases}$
- $\sigma_r(\alpha_{\mathtt{k+1}})\overset{\text{(xiv)}}{=}\mathtt{tt}$ for $\mathfrak{a}=a_{k+1}$, and $\mathtt{ff}$ otherwise
- For $cc=x\sim n$, we have $\mathtt{cc}_\mathtt{k}=(\mathtt{z}_\mathtt{k}-\mathtt{x}_\mathtt{k})\sim n$. Because $\nu_k\models(x\sim n)$ (Definition 2.2.4), we have $\sigma_r(\mathtt{z}_\mathtt{k})-\sigma_r(\mathtt{x}_\mathtt{k})\overset{\text{(xiii),(IH)}}{\sim}n$, thus $\sigma_r\models\mathtt{cc}_\mathtt{k}$. The argumentation for $\sigma_r\models\mathtt{I}(\mathtt{s}')_{\mathtt{k+1}}$ is similar.

From the above, we get $\sigma_r\models\varphi^{action}(e)_{k+1/t}$ (3.2) (so $\sigma_r\models\varphi^{trans}(\mathfrak{A})_{k+1/t}$ (3.4)), $\sigma_r\models\varphi^{location}(\mathfrak{A})_{k+1/t}$ (3.5), and $\sigma_r\models\varphi^{mutex}(\mathfrak{A})_{k+1/t}$ (3.6).

In case of a delay transition, we have $l_k=s=l_{k+1}$, $a_{k+1}=t$ for some $t\in\mathtt{Time}$, and $\nu_{k+1}=\nu_k+t$. Then

- $\sigma_r(\mathtt{s}_{\mathtt{k+1}})\overset{\text{(xi)}}{=}\mathtt{tt}$ for $s=l_{k+1}$, and $\mathtt{ff}$ otherwise
- $\sigma_r(\mathtt{z}_{\mathtt{k+1}})\overset{\text{(xii)}}{=}\sigma_r(\mathtt{z}_\mathtt{k})+t$
- $\sigma_r(\mathtt{x}_{\mathtt{k+1}})\overset{\text{(xiii)}}{=}\sigma_r(\mathtt{z}_{\mathtt{k+1}})-\nu_{k+1}(x)\overset{\text{(xii)}}{=}\sigma_r(\mathtt{z}_\mathtt{k})+t-(\nu_k(x)+t)=$ $\sigma_r(\mathtt{z}_\mathtt{k})-\nu_k(x)\overset{\text{(xiii)}}{=}\sigma_r(\mathtt{x}_\mathtt{k})$
- $\sigma_r(\alpha_{\mathtt{k+1}})\overset{\text{(xiv)}}{=}\mathtt{ff}$ for all $\mathfrak{a}\in\Sigma$
- $\sigma_r\models\mathtt{I}(\mathtt{s})_{\mathtt{k+1}}$: similar to the argumentation for $\sigma_r\models\mathtt{cc}_\mathtt{k}$ above

From the above, we get $\sigma_r\models\varphi^{delay}(e)_{k+1/t}$ (3.3) (so $\sigma_r\models\varphi^{trans}(\mathfrak{A})_{k+1/t}$ (3.4)), $\sigma_r\models\varphi^{location}(\mathfrak{A})_{k+1/t}$ (3.5), and $\sigma_r\models\varphi^{mutex}(\mathfrak{A})_{k+1/t}$ (3.6).

Together, we get $\sigma_r\models\varphi(\mathfrak{S})_{k+1}$ for $\mathfrak{S}=\mathfrak{A}$

[2]Here and in the remainder of the proof, again we only show the basic cases for simple clock constraints (without clock differences) and simple data constraints (without addition/subtraction).

$\mathfrak{S} = \mathfrak{T}$: $r{=}\langle l_0,\delta_0,\nu_0\rangle \xrightarrow{P_1,\bar{\delta}_1,t_1} \dots \xrightarrow{P_k,\bar{\delta}_k,t_k} \langle l_k,\delta_k,\nu_k\rangle \xrightarrow{P_{k+1},\bar{\delta}_{k+1},t_{k+1}} \langle l_{k+1},\delta_{k+1},\nu_{k+1}\rangle$ $\in Run_{\mathfrak{T},k+1}$, and the last step $\langle l_k,\delta_k,\nu_k\rangle \xrightarrow{P_{k+1},\bar{\delta}_{k+1},t_{k+1}} \langle l_{k+1},\delta_{k+1},\nu_{k+1}\rangle$ results from following either a visible transition (2.7) or an invisible transition transition (2.7), (2.8).

In case of a visible transition $e{=}(s,P,dc,cc,\lambda,s')$, we have $l_k{=}s$, $l_{k+1}{=}s'$, $P_{k+1}{=}P$, and $\nu_{k+1}{=}\nu_k{+}t_{k+1}$, with $t_{k+1}{>}0$. Then

- $\sigma_r(\mathtt{s_{k+1}})$, $\sigma_r(\mathtt{x_{k+1}})$: equivalent to the respective cases for $\mathfrak{S}{=}\mathfrak{A}$ above
- $\sigma_r(\mathtt{z_{k+1}})\overset{(xii)}{=}\sigma_r(\mathtt{z_k}){+}t_{k+1}$
- $\sigma_r(\mathtt{p_{k+1}})\overset{(xv)}{=}\mathtt{tt}$ for $p{\in}P$, $\mathtt{ff}$ otherwise
- $\sigma_r(\mathtt{Dp_{k+1}})\overset{(xvi)}{=}\Delta(\bar{\delta}_{k+1})$ for $p{\in}P$, $\mathtt{n}^{\perp}$ otherwise
- $\sigma_r(\mathtt{Dd_{k+1}})\overset{(xvii)}{=}\Delta(\delta_{k+1}(d))$ if $d{\in}\#(s')$, $\Delta(\bar{\delta}_{k+1}(d))$ otherwise
- $\sigma_r(\mathtt{d_{k+1}})\overset{(xviii)}{=}\mathtt{ff}$ if $\sigma_r(\mathtt{Dd_{k+1}}){=}\mathtt{n}^{\perp}$, $\mathtt{tt}$ otherwise
- For $I(s){=}(x{\sim}n)$, we have $\mathtt{I(s)_{k+1\Delta}}{=}(\mathtt{z_{k+1}}{-}\mathtt{x_k}){\sim}n$. Since $\nu_k{+}t{\models}I(s)$ for all $0{\leq}t{\leq}t_{k+1}$ (Definition 2.3.5), in particular $\nu_k(x){+}t_{k+1}{\models}(x{\sim}n)$; so $\sigma_r(\mathtt{z_{k+1}}){-}\sigma_r(\mathtt{x_k})\overset{\text{IH},(xiii)}{=}(\sigma_r(\mathtt{z_k}){+}t_{k+1}){-}(\sigma_r(\mathtt{z_k}){-}\nu_k(x)){=}\nu_k(x){+}t_{k+1}$, which means that $\sigma_r(\mathtt{z_{k+1}}){-}\sigma_r(\mathtt{x_k}){\sim}n$ holds. Therefore $\sigma_r{\models}\mathtt{I(s)_{k+1\Delta}}$. The argumentation for $\sigma_r{\models}\mathtt{cc_{k+1\Delta}}$ is similar, and the argumentation for $\sigma_r{\models}\mathtt{I(s')_{k+1}}$ is equivalent to the respective case for $\mathfrak{S}{=}\mathfrak{A}$ above
- For $dc{=}(\mathbf{D}{\simeq}\mathbf{D}')$, ${\simeq}{\in}\{=,\leqslant\}$ (cf. Definition 2.1.7 and Section 3.1.1.5), we have $\mathtt{dc_{k+1}}{=}(\mathtt{D}{\simeq}\mathtt{D}')$, where $\mathtt{D}$ and $\mathtt{D}'$ are either port data variables $\mathtt{Dp_{k+1}}$ or data content variables $\mathtt{Dd_k}$, $\mathtt{Dd_{k+1}}$ (cf. Section 3.1.1.5). Because $\bar{\delta}_{k+1}{\models}dc$ (Definition 2.3.5), in particular $\bar{\delta}(\mathbf{D})$ and $\bar{\delta}(\mathbf{D}')$ such that $\bar{\delta}(\mathbf{D}){\simeq}\bar{\delta}(\mathbf{D}')$ holds. Therefore, $\sigma_r(\mathtt{D}){\simeq}\sigma_r(\mathtt{D}')$ holds as well ((xvi), (xvii)),[3] and thus $\sigma_r{\models}\mathtt{dc_{k+1}}$.
  The case where $\mathbf{D}{\in}\mathcal{D}ata$ and/or $\mathbf{D}'{\in}\mathcal{D}ata$, that means where $\mathtt{D}$ or $\mathtt{D}'$ are data element representations $\mathtt{n^i}$, is a simplification of the above.

From the above, we get $\sigma_r{\models}\varphi^{visible}(e)_{k+1/t}$ (3.11) (so $\sigma_r{\models}\varphi^{trans}(\mathfrak{T})_{k+1/t}$ (3.16), $\sigma_r{\models}\varphi^{location}(\mathfrak{T})_{k+1/t}$ (3.14), and $\sigma_r{\models}\varphi^{mutex}(\mathfrak{T})_{k+1/t}$ (3.15).

In case of an invisible transition $e{=}(s,\emptyset,dc,cc,\lambda,s')$, we have $l_k{=}s$, $l_{k+1}{=}s'$, $P_{k+1}{=}\emptyset$, and $\nu_{k+1}{=}\nu_k{+}t_{k+1}$, with $t_{k+1}{\geq}0$. Then

- $\sigma_r(\mathtt{s_{k+1}})$, $\sigma_r(\mathtt{z_{k+1}})$, $\sigma_r(\mathtt{x_{k+1}})$ as above
- $\sigma_r(\mathtt{p_{k+1}})\overset{(xv)}{=}\mathtt{ff}$ for all $p{\in}\mathcal{P}$
- $\sigma_r(\mathtt{Dd_{k+1}})$, $\sigma_r(\mathtt{d_{k+1}})$, $\sigma_r{\models}\mathtt{I(s)_{k+1\Delta}}$, $\sigma_r{\models}\mathtt{cc_{k+1\Delta}}$, $\sigma_r{\models}\mathtt{I(s')_{k+1}}$ $\sigma_r{\models}\mathtt{dc_{k+1}}$: as above

From the above, we get $\sigma_r{\models}\varphi^{invisible}(e)_{k+1/t}$ (3.12) (so $\sigma_r{\models}\varphi^{trans}(\mathfrak{T})_{k+1/t}$ (3.16), $\sigma_r{\models}\varphi^{location}(\mathfrak{T})_{k+1/t}$ (3.14), and $\sigma_r{\models}\varphi^{mutex}(\mathfrak{T})_{k+1/t}$ (3.15).

Together, we get $\sigma_r{\models}\varphi(\mathfrak{S})_{k+1}$ for $\mathfrak{S}{=}\mathfrak{T}$

$\mathfrak{S} = \mathfrak{N}$: $r{=}\langle l_0,\delta_0,\nu_0\rangle \xrightarrow{\mathbb{c}_1,\gamma_1} \dots \xrightarrow{\mathbb{c}_k,\gamma_k} \langle l_k,\delta_k,\nu_k\rangle \xrightarrow{\mathbb{c}_{k+1},\gamma_{k+1}} \langle l_{k+1},\delta_{k+1},\nu_{k+1}\rangle$, where $r{\in}Run_{\mathfrak{N},k+1}$, and either the last step $\langle l_k,\delta_k,\nu_k\rangle \xrightarrow{\mathbb{c}_{k+1},\gamma_{k+1}} \langle l_{k+1},\delta_{k+1},\nu_{k+1}\rangle$

[3]Note that (Definition 2.3.5) $\delta(d)$ and $\bar{\delta}(d)$ coincide in case $d{\in}\#(s)$ for any location s.

is an action transition (2.15) resulting from executing a communication, or it is a delayed action transition (2.16) resulting from executing a delay.

In case of a communication $e=(s, \mathbb{c}, dc, cc, \lambda, s')$, we have $l_k=s$, $l_{k+1}=s'$, and $\gamma_{k+1}=\bar{\delta}_{k+1}$. Then

- $\sigma_r(\mathtt{s_{k+1}})$, $\sigma_r(\mathtt{x_{k+1}})$, $\sigma_r(\mathtt{z_{k+1}})$, $\sigma_r(\mathtt{d_{k+1}})$: equivalent to the respective cases for $\mathfrak{S}=\mathfrak{T}$ above
- $\sigma_r(\mathtt{p_{k+1}})\overset{(\mathrm{xv})}{=}\mathtt{tt}$ if $\mathbb{c}_{k+1}(p)=$—, $\mathtt{ff}$ otherwise
- $\sigma_r(\mathtt{cp_{k+1}})\overset{(\mathrm{xix})}{=}\mathtt{tt}$ if $\mathbb{c}_{k+1}(p)=$-!-, $\sigma_r(\mathtt{cp_{k+1}})=\mathtt{ff}$ if $\mathbb{c}_{k+1}(p)=$-?-, unspecified otherwise
- $\sigma_r(\mathtt{Dp_{k+1}})\overset{(\mathrm{xvi})}{=}\Delta(\bar{\delta}_{k+1}(p))$ if $\mathbb{c}_{k+1}(p)=$—, $\mathtt{n}^{\perp}$ otherwise
- $\sigma_r(\mathtt{Dd_{k+1}})$, $\sigma_r(\mathtt{d_{k+1}})$, $\sigma_r\models\mathtt{cc_k}$, $\sigma_r\models\mathtt{dc_{k+1}}$, $\sigma_r\models\mathtt{I(s')_{k+1}}$: equivalent to the respective cases for $\mathfrak{S}=\mathfrak{T}$ above

From the above, we get $\sigma_r\models\varphi^{commu}(e)_{k+1/t}$ (3.21) (so $\sigma_r\models\varphi^{trans}(\mathfrak{N})_{k+1/t}$ (3.26)), $\sigma_r\models\varphi^{location}(\mathfrak{N})_{k+1/t}$ (3.24), and $\sigma_r\models\varphi^{mutex}(\mathfrak{N})_{k+1}$ (3.25).

The case of a delay $e=(s, \mathbb{c}, dc, cc, id, s)$ is essentially equivalent to the case of a communication, and needs not be considered separately. For a delay, we get $\sigma_r\models\varphi^{delay}(e)_{k+1/t}$ (3.22) (so $\sigma_r\models\varphi^{trans}(\mathfrak{N})_{k+1/t}$ (3.26)), $\sigma_r\models\varphi^{location}(\mathfrak{N})_{k+1/t}$ (3.24), and $\sigma_r\models\varphi^{mutex}(\mathfrak{N})_{k+1}$ (3.25).

Together, we get $\sigma_r\models\varphi(\mathfrak{S})_{k+1}$ for $\mathfrak{S}=\mathfrak{N}$

Finally, we get $\sigma_r\models\varphi(\mathfrak{S})_{k+1}$ for all systems $\mathfrak{S}\in\{\mathfrak{A}, \mathfrak{T}, \mathfrak{N}\}$, and we define the map $\downarrow^r_\sigma: Run_{\mathfrak{S},k}\to\mathcal{V}(\varphi(\mathfrak{S})_k)$ such that for every run $r\in Run_{\mathfrak{S},k}$, $\downarrow^r_\sigma(r)=\sigma_r\in\mathcal{V}(\varphi(\mathfrak{S})_k)$ is the derived interpretation. □

Proposition A.1.8 (Derived Interpretation, Product). For $r\in Run_{\mathfrak{S}_1\bowtie\mathfrak{S}_2,k}$, the derived interpretation σ_r is a model of $\varphi(\mathfrak{S}_1\bowtie\mathfrak{S}_2)_k$, i.e. $\sigma\in\mathcal{V}(\varphi(\mathfrak{S}_1\bowtie\mathfrak{S}_2)_k)$.

Proof (Idea). The proof is along the same lines as the proof of Lemma A.1.7: in **IS**, we show that for $i=1,2$, the derived interpretation σ_r for a run $r\in Run_{\mathfrak{S}_1\bowtie\mathfrak{S}_2,k+1}$, reduced to the variables of $\varphi(\mathfrak{S}_i)_k$, is a model of $\varphi(\mathfrak{S}_i)_k$. □

Using the above, the proof of Theorem 3.2.4 (found on Page 61) is straightforward:

Proof of Theorem 3.2.4. This follows directly from Lemma A.1.4 and Lemma A.1.7. □

Theorem A.1.9 (Soundness, Completeness). The formula representation $\varphi(\mathfrak{S})$ of a real-time system $\mathfrak{S}$, as defined in Definitions 3.1.1, 3.1.4 and 3.1.9, is correct, that means $\varphi(\mathfrak{S})$ exhibits the same behaviour as $\mathfrak{S}$.

Proof. This follows directly from Lemma A.1.4 and Lemma A.1.7. □

A.2 Correctness of Abstraction

In this Section, we prove that the abstraction function α, as presented in Section 4.1, yields *a correct over-approximation.* To yield an over-approximation, every finite run of the concrete system $\mathfrak{S}$ (represented by a model of $\varphi(\mathfrak{S})_k$, see Theorem A.1.9) has to be reproducible in the abstract case.[4] This is captured in Lemma 4.1.7. Here, we prove an even stronger correctness result, which in particular emphasises the structural relationships between concrete and abstract formula. We show that the diagram in Figure A.2 commutes, which allows us to conclude the existence of a homomorphism h_R between concrete and abstract set of runs.

$$\begin{tikzcd}
\arrow[r, phantom, "\overbrace{\hspace{4cm}}^{i}"] & \arrow[r, phantom, "\overbrace{\hspace{4cm}}^{ii}"] & \arrow[r, phantom, "\overbrace{\hspace{4cm}}^{iii}"] & \\
\mathfrak{S} \arrow[r, "\varphi"] \arrow[d, "\text{run}"'] & \varphi(\mathfrak{S})_k \arrow[r, "\alpha"] \arrow[d, "\text{model}"] & \varphi(\widetilde{\mathfrak{S}})_k \arrow[d, "\text{model}"] & \widetilde{\mathfrak{S}} \arrow[l, "\varphi"'] \arrow[d, "\text{run}"] \\
\mathit{Run}_{\mathfrak{S},k} \arrow[r, bend left, "\downarrow^{r}_{\sigma}"] \arrow[rrr, bend right=40, "h_R"'] & \mathcal{V}(\varphi(\mathfrak{S})_k) \arrow[l, bend left, "\downarrow^{\sigma}_{r}"] \arrow[r, "\subseteq"] & \mathcal{V}(\varphi(\widetilde{\mathfrak{S}})_k) \arrow[r, bend left, "\downarrow^{\sigma}_{r}"] & \mathit{Run}_{\widetilde{\mathfrak{S}},k} \arrow[l, bend left, "\downarrow^{r}_{\sigma}"]
\end{tikzcd}$$

Figure A.2: Strong Correctness of Abstraction

The idea of the proof is as follows: since α works locally, it retains the formula structure of $\varphi(\mathfrak{S})$ if $\mathfrak{S}{=}\mathfrak{A}$ (cf. (3.7)), and it retains the formula structure of $\varphi(\mathfrak{S})$ up to data constraints if $\mathfrak{S}{=}\mathfrak{T}$ (cf. (3.16)).[5] Therefore, there exists some system $\widetilde{\mathfrak{S}}$ of the same representation $\varphi(\widetilde{\mathfrak{S}})_k = \alpha(\varphi(\mathfrak{S})_k)$ (up to logical equivalence and data constraints). With this, subdiagrams (*i*) and (*iii*) in Figure A.2 commute according to Theorem A.1.9. Moreover, subdiagram (*ii*) in Figure A.2 commutes according to Lemma 4.1.7 (since every model of $\varphi(\widetilde{\mathfrak{S}})_k$ is a model of $\alpha(\varphi(\mathfrak{S})_k)$), such that the whole diagram commutes.

Notation A.2.1 (Notation of Systems). If not stated otherwise, we shall assume the constituents of a TA $\mathfrak{A}$ to be denoted as $\mathfrak{A}{=}(S, s_0, \Sigma, \mathcal{X}, I, E)$, and of a TCA $\mathfrak{T}$ as $\mathfrak{T}{=}(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$. We use the general notion $\mathfrak{S}$, with $\mathfrak{S}{\in}\{\mathfrak{A}, \mathfrak{T}\}$, whenever possible, and if applicable, we may refer to common constituents (i.e., S, s_0, $\mathcal{X}$, I, E) without explicitly mentioning $\mathfrak{A}$ or $\mathfrak{T}$. For a system with identifier $\widetilde{\mathfrak{S}}$, we add the symbol $\widetilde{\ }$ to all constituents, equivalently, for a system with identifier $\mathfrak{S}_i$, we add index i to all constituents.

We use the notation of representation variables introduced in Section 3.1.1.

[4]Note that unlike in Section A.1, where we had $\mathfrak{S}{\in}\{\mathfrak{A}, \mathfrak{T}, \mathfrak{N}\}$, here we only have $\mathfrak{S}{\in}\{\mathfrak{A}, \mathfrak{T}\}$, cf. Section 4.1.

[5]To guarantee that α yields an over-approximation, we may retain only those data constraints that reason about ports not merged by γ, cf. Definition 4.1.3 and the explanations thereafter. Therefore, we cannot expect that the formula structure of data constraints is preserved.

Definition A.2.2 (Homomorphism of Runs). Let $\mathfrak{A}$, $\widetilde{\mathfrak{A}}$ be TA, $\mathfrak{T}$, $\widetilde{\mathfrak{T}}$ be TCA, both with $\mathcal{X}{\supseteq}\widetilde{\mathcal{X}}$ and $|S|{\geq}|\widetilde{S}|$. Let $|\Sigma|{\geq}|\widetilde{\Sigma}|$, $|\mathcal{P}|{\geq}|\widetilde{\mathcal{P}}|$, and $|\mathcal{D}|{\geq}|\widetilde{\mathcal{D}}|$. Let $\mathfrak{S}_{\mathfrak{A}}$, $\mathfrak{S}_{\mathfrak{T}}$, $\mathfrak{S}_{\widetilde{\mathfrak{A}}}$ and $\mathfrak{S}_{\widetilde{\mathfrak{T}}}$ be the associated transition systems, and let $Run_{\mathfrak{A}}$, $Run_{\mathfrak{T}}$, $Run_{\widetilde{\mathfrak{A}}}$ and $Run_{\widetilde{\mathfrak{T}}}$ be the sets of runs. Let $\gamma_S{:}S{\to}\widetilde{S}$, $\gamma_\Sigma{:}\Sigma{\to}\widetilde{\Sigma}$, $\gamma_P{:}\mathcal{P}{\to}\widetilde{\mathcal{P}}$ and $\gamma_{\mathcal{D}}{:}\mathcal{D}{\to}\widetilde{\mathcal{D}}$ be total, surjective mappings.

A function $h_R{:}Run_{\mathfrak{A}}{\to}Run_{\widetilde{\mathfrak{A}}}$ is called a *homomorphism of runs (between $Run_{\mathfrak{A}}$ and $Run_{\widetilde{\mathfrak{A}}}$)* iff for each run

$$r{=}\langle l_0,\nu_0\rangle\xrightarrow{a_1}\langle l_1,\nu_1\rangle\xrightarrow{a_2}\langle l_2,\nu_2\rangle{\in}Run_{\mathfrak{A}},$$

there exists a run $h(r){=}\widetilde{r}$,

$$\widetilde{r}{=}\langle\widetilde{l_0},\widetilde{\nu_0}\rangle\xrightarrow{\widetilde{a_1}}\langle\widetilde{l_1},\widetilde{\nu_1}\rangle\xrightarrow{\widetilde{a_2}}\langle\widetilde{l_2},\widetilde{\nu_2}\rangle\ldots{\in}Run_{\widetilde{\mathfrak{A}}},$$

with $\gamma_S(l_i){=}\widetilde{l_i}$, $\widetilde{\nu_i}{=}\nu_i|_{\widetilde{\mathcal{X}}}$, and $\gamma_\Sigma(\gamma_i){=}\widetilde{\gamma_i}$ for all $i{\geq}0$.

A function $h_R{:}Run_{\mathfrak{T}}{\to}Run_{\widetilde{\mathfrak{T}}}$ is called a *homomorphism of runs (between $Run_{\mathfrak{T}}$ and $Run_{\widetilde{\mathfrak{T}}}$)* iff for each run

$$r{=}\langle l_0,\delta_0,\nu_0\rangle\xrightarrow{P_1,\bar{\delta}_1,t_1}\langle l_1,\delta_1,\nu_1\rangle\xrightarrow{P_2,\bar{\delta}_2,t_2}\langle l_2,\delta_2,\nu_2\rangle{\in}Run_{\mathfrak{T}},$$

there exists a run $h(r){=}\widetilde{r}$,

$$\widetilde{r}{=}\langle\widetilde{l_0},\widetilde{\delta_0},\widetilde{\nu_0}\rangle\xrightarrow{\widetilde{P_1},\widetilde{\bar{\delta}_1},\widetilde{t_1}}\langle\widetilde{l_1},\widetilde{\delta_1},\widetilde{\nu_1}\rangle\xrightarrow{\widetilde{P_2},\widetilde{\bar{\delta}_2},\widetilde{t_2}}\langle\widetilde{l_2},\widetilde{\delta_2},\widetilde{\nu_2}\rangle{\in}Run_{\widetilde{\mathfrak{T}}},$$

with $\gamma_S(l_i){=}\widetilde{l_i}$, $\widetilde{\nu_i}{=}\nu_i|_{\widetilde{\mathcal{X}}}$, $\gamma_P(P_i){=}\widetilde{P_i}$, $\widetilde{\bar{\delta}_i}(\widetilde{p}){=}n$ only if $\delta_i(p){=}n$ for some $p{\in}\gamma_P^{-1}(\widetilde{p})$, $\widetilde{\delta_i}(\widetilde{d}){=}n$ only if $\delta_i(d){=}n$ for some $d{\in}\gamma_{\mathcal{D}}^{-1}(\widetilde{d})$, and $\widetilde{\bar{\delta}_i}(\widetilde{d}){=}n$ only if $\bar{\delta}_i(d){=}n$ for some $d{\in}\gamma_{\mathcal{D}}^{-1}(\widetilde{d})$, for all $i{\geq}0$.

For the sets of finite runs $Run_{\mathfrak{A},k}$, $Run_{\mathfrak{T},k}$, $Run_{\widetilde{\mathfrak{A}},k}$ and $Run_{\widetilde{\mathfrak{T}},k}$, h_R is defined analogously.

Intuitively speaking, an abstraction is correct if the semantics of the abstract system is not reduced with respect to the semantics of the concrete system. That means, every behaviour that is possible in the concrete system has to be possible in the abstract system as well. Since we have defined the semantics of a real-time system $\mathfrak{S}$ via sets of runs (Definitions 2.2.4 and 2.3.5), an abstraction of $\mathfrak{S}$ is correct if for all systems $\mathfrak{S}$ and $\widetilde{\mathfrak{S}}$, such that $\widetilde{\mathfrak{S}}$ is obtained from $\mathfrak{S}$ by abstraction, there exists a homomorphism of runs $h_R{:}Run_{\mathfrak{S}}{\to}Run_{\widetilde{\mathfrak{S}}}$, as defined in definition A.2.2. To prove the existence of h_R, we show that Figure A.2 is a commuting diagram.

The general proof idea is shown in Figure A.3: let $\mathfrak{S}$ be a real-time system, with k-unfolding $\varphi(\mathfrak{S})_k$. The abstraction function α preserves the structure of $\varphi(\mathfrak{S})_k$, that means the abstraction $\alpha(\varphi(\mathfrak{S})_k)$ of $\varphi(\mathfrak{S})_k$ is the k-unfolding $\varphi(\widetilde{\mathfrak{S}})_k$ of some system $\widetilde{\mathfrak{S}}$. Though the abstraction function α is defined on formulas rather than on systems, the system $\widetilde{\mathfrak{S}}$ can be "derived" from the formula representation $\alpha(\varphi(\mathfrak{S})_k)$ (Proposition A.2.7). For $r{\in}Run_{\mathfrak{S},k}$ and $\widetilde{r}{\in}Run_{\widetilde{\mathfrak{S}},k}$, such that $h_R(r){=}\widetilde{r}$, there exists an interpretation $\sigma{\in}\mathcal{V}(\varphi(\widetilde{\mathfrak{S}})_k)$, such that $\widetilde{r}$ is the derived run r_σ of σ.

In other words, the commutative property can be summarised as follows: the possible behaviour of the abstract system $\widetilde{\mathfrak{S}}$, given by the set of runs $Run_{\widetilde{\mathfrak{S}},k}$, is

$$\begin{tikzcd}
\mathfrak{S} \arrow[r, "\varphi"] \arrow[d, "\text{run}"'] & \varphi(\mathfrak{S})_k \arrow[r, "\alpha"] \arrow[d, "\text{model}"'] & \varphi(\widetilde{\mathfrak{S}})_k \arrow[d, "\text{model}"] & \widetilde{\mathfrak{S}} \arrow[l, "\varphi"'] \arrow[d, "\text{run}"] \\
Run_{\mathfrak{S},k} \arrow[r, bend left, "\downarrow^r_\sigma"] \arrow[rrr, bend right=20, "h_R"'] & \mathcal{V}(\varphi(\mathfrak{S})_k) \arrow[l, bend left, "\downarrow^\sigma_r"] \arrow[r, "\subseteq"] & \mathcal{V}(\varphi(\widetilde{\mathfrak{S}})_k) \arrow[r, bend left, "\downarrow^\sigma_r"] & Run_{\widetilde{\mathfrak{S}},k} \arrow[l, bend left, "\downarrow^r_\sigma"]
\end{tikzcd}$$

Figure A.3: Abstraction by Omission: Basic proof idea

obtained from the possible behaviour of the original system $Run_{\mathfrak{S},k}$ and the homomorphism of runs h_R (lower path in Figure A.3). $Run_{\widetilde{\mathfrak{S}},k}$ is also obtained from the k-unfolding $\varphi(\mathfrak{S})_k$ of $\mathfrak{S}$, the abstraction function α, the set of models of $\alpha(\varphi(\mathfrak{S})_k)$, and the set of derived runs for these interpretations (upper path in Figure A.3).

We can already state that

Proposition A.2.3 (Commuting Subdiagrams). The subdiagrams (i) and (iii) in Figure A.2 are a commuting diagram each.

Proof. This follows directly from Theorem A.1.9. □

The subdiagram (iii) in Figure A.2 is a commuting diagram when considered separately. Yet, with respect to the overall context of Figure A.2, the fact that MO is not defined on systems $\mathfrak{S}$ but on formulas has to be taken into account. However, Proposition A.2.7 below will show the existence of such an abstract system $\widetilde{\mathfrak{S}}$.

We first show that the abstract formula $\alpha(\varphi(\mathfrak{S}))$ is weaker than the concrete formula $\varphi(\mathfrak{S})$ (cf. Lemma 4.1.7 on Page 73).

Proof of Lemma 4.1.7. Let L be a literal. The proof is done inductively on the structure of the formula F:

IA: We need to consider the different cases in (4.1)

- If $F{=}L$, $Conts(L){\cap}^\bullet\alpha{=}\emptyset$, then $\alpha(F)\overset{(4.1a)}{=}L$. $(L \to L)$ holds trivially.
- If $F{=}L$, $Conts(L){\cap}^\bullet\alpha{\neq}\emptyset$, $L{=}p{\in}\mathtt{P}$, then $\alpha(F)\overset{(4.1b)}{=}\gamma(p)$. By definition of γ, $\gamma(p){=}q$ for some $q{\in}\mathtt{P}'$. By definition of γ_α (4.3), $(p \to q)$ holds.
- For all other literals L, $\alpha(L)\overset{(4.1e)}{=}\mathtt{true}$. $(L \to \mathtt{true})$ holds trivially.
- If $F{=}\neg p{\wedge}p'$, with $p, p'{\in}\mathtt{P}$ and $\gamma(p){=}\gamma(p'){=}q$ (basic case of (4.1c)), then $\alpha(F)\overset{(4.1c),(4.2a)}{=}\alpha(\neg p){\wedge}\alpha(p')\overset{(4.1c),(4.1b)}{=}q{\wedge}q{=}q$. By definition of γ_α, $((\neg p{\wedge}p'){\to}q)$ holds.
- If $F{=}\neg p{\wedge}\neg p''$, with $p, p''{\in}\mathtt{P}$ and $\gamma(p){=}\gamma(p''){=}q$ (basic case of (4.1d)), then $\alpha(F)\overset{(4.1d),(4.2a)}{=}\alpha(\neg p){\wedge}\alpha(\neg p'')\overset{(4.1d),(4.1b)}{=}\neg q{\wedge}\neg q = \neg q$. By definition of γ_α, $((\neg p{\wedge}\neg p'') \to \neg q)$ holds.

IH: For formulas F_1 and F_2, $(F_1 \to \alpha(F_1))$ and $(F_2 \to \alpha(F_2))$ holds.

IS:
- If $F{=}F_1{\wedge}F_2$, then $\alpha(F)\overset{(4.2a)}{=}\alpha(F_1){\wedge}\alpha(F_2)$. $((F_1{\wedge}F_2) \rightarrow (\alpha(F_1){\wedge}\alpha(F_2)))$ holds by IH and propositional logic.
- If $F{=}F_1{\vee}F_2$, then $\alpha(F)\overset{(4.2b)}{=}\alpha(F_1){\vee}\alpha(F_2)$. $((F_1{\vee}F_2) \rightarrow (\alpha(F_1){\vee}\alpha(F_2)))$ holds by IH and propositional logic.
- If $F{=}F_1{\wedge}\gamma_\alpha$, then $\alpha(F)\overset{(4.4)}{=}\alpha(F_1) \wedge \gamma_\alpha$. $((F_1{\wedge}\gamma_\alpha) \rightarrow (\alpha(F_1){\wedge}\gamma_\alpha))$ holds by IH and propositional logic.

□

We want to show that MO preserves the formula representation. Since MO is defined for formulas in NNF (cf. Definition 4.1.5), we first show that transformation to NNF preserves the formula representation.

Remark A.2.4 (NNF preserves the Formula Representation). Let $\mathfrak{S}$ be a real-time system, with formula representation $\varphi(\mathfrak{S})$ and k-unfolding $\varphi(\mathfrak{S})_k$. The *transformation to NNF of $\varphi(\mathfrak{S})$ and $\varphi(\mathfrak{S})_k$ preserves the formula structure*, that means, $NNF(\varphi(\mathfrak{S}))$ is a formula of the form (3.7) respectively (3.16), and similarly, $NNF(\varphi(\mathfrak{S})_k)$ is a formula of the form (3.29).

Proof. For $\mathfrak{S}{=}\mathfrak{A}$, the formulas $\varphi(\mathfrak{A})$ and $\varphi(\mathfrak{A})_k$ are in NNF already, so nothing needs to be shown.

For $\mathfrak{S}{=}\mathfrak{T}$, the only parts of $\varphi(\mathfrak{T})$ and $\varphi(\mathfrak{T})_k$ which are not yet in NNF are the representations of data constraints. It is easy to see that for a data constraint $dc{\in}\mathcal{DC}(\mathcal{P},\mathcal{D})$, with representation $\mathtt{dc}{\in}\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D})$, the transformation $NNF(\mathtt{dc})$ to NNF is a well-formed data constraint according to Definition 2.1.7, too, that means $NNF(\mathtt{dc}){\in}\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D})$. □

Next, we show that MO preserves the structure of data and clock constraints.

Lemma A.2.5 (MO preserves Data and Clock Constraints). Let $\mathcal{P}$ be a set of ports, $\mathcal{D}$ a set of data variables, and $\mathcal{X}$ a set of clocks. Let $dc{\in}\mathcal{DC}(\mathcal{P},\mathcal{D})$ be a data constraint (cf. Definition 2.1.7), $cc{\in}\mathcal{CC}(\mathcal{X})$ a clock constraint (cf. Definition 2.1.2). Let $\mathtt{dc}{\in}\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$ be the representation of dc, and $\mathtt{cc}{\in}\mathcal{CC}(\mathtt{X})$ the representations of cc (cf. Section 3.1.1). The abstraction function α *preserves the structure of data and clock constraints*, that means $\alpha(\mathtt{dc})$ and $\alpha(\mathtt{cc})$ are also valid representations of data and clock constraints.

Proof. By definition, α changes only literals. Since $\mathtt{dc}$ and $\mathtt{cc}$ do not contain propositional variables, neither of (4.1b), (4.1c) or (4.1d) is applicable. Therefore, α preserves the logical structure,[6] and literals are either kept unchanged (4.1a) or mapped to $\mathtt{true}$ (4.1e). Thus, $\alpha(\mathtt{dc}){\in}\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$, and $\alpha(\mathtt{cc}){\in}\mathcal{CC}(\mathtt{X})$. □

[6]The logical structure of a formula is the order of its literals and the logical operators $\wedge$, $\vee$ and $\neg$. For example, for a formula $F = (p \vee \neg q) \wedge \neg(r \wedge \neg(x = 5))$, with $p, q, r \in \mathcal{P}$ being atomic propositions and $x \in \mathcal{V}$ being a variable, the logical structure is $F = (l_1 \vee l_2) \wedge \neg(l_3 \wedge l_4)$ (for literals l_i). Note that an occurrence of $\neg$ is part of the logical structure only if it is not part of a literal.

Remark A.2.6 (Lifting of MO). For argumentation purposes, we lift α in the straightforward way to reason about constituents of systems rather than formulas. For example, for a clock x with representation $\mathtt{x}$, we may write $x{\in}\mathcal{O}$ instead of $\mathtt{x}{\in}\mathcal{O}$.

Similarly, we lift α to reason about sets rather than single variables. For example, for the set of locations S and the set of clocks $\mathcal{X}$, we may write $\alpha(S)$ and $\alpha(\mathcal{X})$ to denote the set of locations respectively clocks in the abstract system, that means $\alpha(S){=}\{s' \mid s{\in}S, \gamma(s){=}s'\}$, and $\alpha(\mathcal{X}){=}\{x \mid x{\notin}\mathcal{O}\}{=}\mathcal{X}\backslash\mathcal{O}$. By $\alpha(\lambda)$, we denote the update map λ, reduced to the clocks of the abstract system. That is, $\alpha(\lambda){=}\lambda|_{\alpha(\mathcal{X})}$.

We are now ready to show that MO preserves the formula representation of TA, and preserves the formula representation of TCA up to data constraints.

Proposition A.2.7 (MO preserves the Formula Representation). Let $\mathfrak{A} = (S, s_0, \Sigma, \mathcal{X}, I, E)$ be a TA, $\mathfrak{T}{=}(S, s_0, \mathcal{P}, \mathcal{X}, I, \mathcal{D}, \#, E)$ a TCA, with formula representations $\varphi(\mathfrak{A})$, $\varphi(\mathfrak{T})$, and k-unfoldings $\varphi(\mathfrak{A})_k$, $\varphi(\mathfrak{T})_k$ in NNF (cf. Remark A.2.4). Let α be an abstraction function, with γ and $\mathcal{O}$ as in Definition 4.1.5.

The abstraction by merging omission *preserves the formula representation and k-unfolding of* $\mathfrak{A}$, and it *preserves the formula representation and k-unfolding of* $\mathfrak{T}$ *up to data constraints*. That means, there exists a TA $\widetilde{\mathfrak{A}}$, with formula representation $\varphi(\widetilde{\mathfrak{A}})$ and k-unfolding $\varphi(\widetilde{\mathfrak{A}})_k$, such that

$$\begin{aligned} \varphi(\widetilde{\mathfrak{A}}) &= \alpha(\varphi(\mathfrak{S})), \text{ and} \\ \varphi(\widetilde{\mathfrak{A}})_k &= \alpha(\varphi(A)_k), \end{aligned} \tag{xx}$$

and there exists a TCA $\mathfrak{T}$, with formula representation $\varphi(\widetilde{\mathfrak{T}})$ and k-unfolding $\varphi(\widetilde{\mathfrak{T}})_k$, such that

$$\begin{aligned} \varphi(\widetilde{\mathfrak{T}})\backslash_{dc} &= \alpha(\varphi(\mathfrak{T}))\backslash_{dc}, \text{ and} \\ \varphi(\widetilde{\mathfrak{T}})_k\backslash_{dc} &= \alpha(\varphi(T)_k)\backslash_{dc}, \end{aligned} \tag{xxi}$$

where $\backslash_{dc}$ is a function that replaces all literals of the form $(\mathtt{D}{\simeq}\mathtt{D}')$ or $\neg(\mathtt{D}{\simeq}\mathtt{D}')$, with $\simeq\,{\in}\{=, \leqslant\}$, and $\mathtt{D}$, $\mathtt{D}'$ either port data variables $\mathtt{Dp_t}$, data content variables $\mathtt{Dd_t}$, or data element representations $\mathtt{n^i}$ (cf. Definition 2.1.7 and Section 3.1.1.5), and $t{\in}\mathbb{N}$, in a formula by $\mathtt{true}$.

Proof of (xx). (for the proof of (xxi), please refer to Page 143).
Let $\mathfrak{A}'{=}(S', s_0', \Sigma', \mathcal{X}', I', E')$ be a TA, with $S'{=}\alpha(S)$, $s_0'{=}\alpha(s_0)$, $\Sigma'{=}\alpha(\Sigma)$, $\mathcal{X}'{=}\alpha(\mathcal{X})$, $I'(s){=}\alpha(I(s))$ for all $s{\in}S'$, and $E'{=}\{(\alpha(s), \alpha(\mathfrak{a}), \alpha(cc), \alpha(\lambda), \alpha(s')) \mid (s, \mathfrak{a}, cc, \lambda, s') \in E\}$. Let $\varphi(\mathfrak{A}')$ and $\varphi(\mathfrak{A}')_k$ be the formula representation and k-unfolding of $\mathfrak{A}'$. Observe that we have

$$S'{=}\alpha(S){=}\{s|s{\in}S, \alpha(s){=}id\}{\cup}\{s'|s{\in}S, \alpha(s){=}s'\}, \text{and} \tag{*}$$
$$\Sigma'{=}\alpha(\Sigma){=}\{\mathfrak{a}|\mathfrak{a}{\in}\Sigma, \alpha(\mathfrak{a}){=}id\}{\cup}\{\mathfrak{a}'|\mathfrak{a}{\in}\Sigma, \alpha(\mathfrak{a}){=}\mathfrak{a}'\} \tag{**}$$

We first show that $\varphi(\mathfrak{A}'){=}\alpha(\varphi(\mathfrak{A}))$. By Definitions 3.1.1 and 4.1.5, we have

$$\begin{aligned} \alpha(\varphi(\mathfrak{A})) &= \alpha(\varphi^{init}(\mathfrak{A}) \wedge \varphi^{trans}(\mathfrak{A}) \wedge \varphi^{location}(\mathfrak{A}) \wedge \varphi^{mutex}(\mathfrak{A})) \\ &= \alpha(\varphi^{init}(\mathfrak{A})) \wedge \alpha(\varphi^{trans}(\mathfrak{A})) \wedge \alpha(\varphi^{location}(\mathfrak{A})) \wedge \alpha(\varphi^{mutex}(\mathfrak{A})) \\ \varphi(\mathfrak{A}') &= \varphi^{init}(\mathfrak{A}') \wedge \varphi^{trans}(\mathfrak{A}') \wedge \varphi^{location}(\mathfrak{A}') \wedge \varphi^{mutex}(\mathfrak{A}') \end{aligned}$$

Consider the corresponding parts in $\alpha(\varphi(\mathfrak{A}))$ and $\varphi(\mathfrak{A}')$ separately

1. Initial constraints φ^{init}:

$$\begin{aligned}
\alpha(\varphi^{init}(\mathfrak{A})) &= \alpha\big(\bar{\mathtt{s}}_0 \wedge \bigwedge_{s\in S, s\neq\bar{s}} \neg\mathtt{s}_0 \wedge \mathtt{I}(\bar{\mathtt{s}})_0 \wedge \bigwedge_{\mathfrak{a}\in\Sigma} (\neg\alpha_0) \wedge (\mathtt{z}_0{=}0) \wedge \bigwedge_{x\in\mathcal{X}} (\mathtt{x}_0{=}0)\big)\\
&= \alpha(\bar{\mathtt{s}}_0) \wedge \bigwedge_{s\in S, s\neq\bar{s}} \alpha(\neg\mathtt{s}_0) \wedge \alpha(\mathtt{I}(\bar{\mathtt{s}})_0) \wedge \bigwedge_{\mathfrak{a}\in\Sigma} \alpha((\neg\alpha_0)) \wedge\\
&\quad \alpha((\mathtt{z}_0{=}0)) \wedge \bigwedge_{x\in\mathcal{X}} \alpha((\mathtt{x}_0{=}0))\\
&= \alpha(\bar{\mathtt{s}}_0) \wedge \bigwedge_{\substack{s\in S, s\neq\bar{s},\\ \alpha(s)=id}} \neg\mathtt{s}_0 \wedge \bigwedge_{\substack{s\in S, s\neq\bar{s},\\ \alpha(s)=s'\neq\alpha(\bar{s})}} \neg\mathtt{s}'_0 \wedge \alpha(\mathtt{I}(\bar{\mathtt{s}})_0) \wedge \bigwedge_{\substack{\mathfrak{a}\in\Sigma,\\ \alpha(\mathfrak{a})=id}} (\neg\mathfrak{a}_0) \wedge\\
&\quad \bigwedge_{\substack{\mathfrak{a}\in\Sigma,\\ \alpha(\mathfrak{a})=\mathfrak{a}'}} (\neg\alpha'_0) \wedge (\mathtt{z}_0{=}0) \wedge \bigwedge_{x\in\mathcal{X}\setminus\mathcal{O}} (\mathtt{x}_0{=}0)
\end{aligned}$$

$$\varphi^{init}(\mathfrak{A}') = \bar{\mathtt{s}}'_0 \wedge \bigwedge_{s\in S', s\neq\bar{s}'} \neg\mathtt{s}_0 \wedge \mathtt{I}(\bar{\mathtt{s}}')_0 \wedge \bigwedge_{\mathfrak{a}\in\Sigma'} (\neg\alpha_0) \wedge (\mathtt{z}_0{=}0) \wedge \bigwedge_{x\in\mathcal{X}'} (\mathtt{x}_0{=}0)$$

By definition of $\mathfrak{A}'$, we have $\alpha(\bar{\mathtt{s}}_0){=}\bar{\mathtt{s}}'_0$, and $\alpha(\mathtt{I}(\bar{\mathtt{s}})_0){=}\mathtt{I}(\bar{\mathtt{s}}')_0$. Because of (*), (**), and the fact that $\mathcal{X}'{=}\mathcal{X}\setminus\mathcal{O}$, we finally get

$$\alpha(\varphi^{init}(\mathfrak{A})) = \varphi^{init}(\mathfrak{A}')$$

2. Transition relation φ^{trans}:

$$\begin{aligned}
\alpha(\varphi^{trans}(\mathfrak{A})) &= \alpha\big(\bigvee_{e\in E} \varphi^{action}(e) \vee \bigvee_{s\in S} \varphi^{delay}(s)\big)\\
&= \bigvee_{e\in E} \alpha(\varphi^{action}(e)) \vee \bigvee_{s\in S} \alpha(\varphi^{delay}(s))
\end{aligned}$$

$$\varphi^{trans}(\mathfrak{A}') = \bigvee_{e\in E'} \alpha(\varphi^{action}(e)) \vee \bigvee_{s\in S'} \alpha(\varphi^{delay}(s))$$

Consider an action transition $e{=}(s, \mathfrak{a}, cc, \lambda, s'){\in}E$:

$$\begin{aligned}
\alpha(\varphi^{action}(e)) &= \alpha\big(\mathtt{s}_\mathtt{t} \wedge \alpha_{\mathtt{t+1}} \wedge \mathtt{cc}_\mathtt{t} \wedge (\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}}) \wedge \bigwedge_{\lambda(x)=id} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x}_\mathtt{t}) \wedge\\
&\quad \bigwedge_{\lambda(x)=x'} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x'}_{\mathtt{t+1}}) \wedge \bigwedge_{\lambda(x)=n} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{z}_{\mathtt{t+1}}{-}n) \wedge \mathtt{s'}_{\mathtt{t+1}} \wedge \mathtt{I}(\mathtt{s'})_{\mathtt{t+1}}\big)\\
&= \alpha(\mathtt{s}_\mathtt{t}) \wedge \alpha(\alpha_\mathtt{t}) \wedge \alpha(\mathtt{cc}_\mathtt{t}) \wedge \alpha(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}}) \wedge \bigwedge_{\lambda(x)=id} \alpha(\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x}_\mathtt{t}) \wedge\\
&\quad \bigwedge_{\lambda(x)=x'} \alpha(\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x'}_{\mathtt{t+1}}) \wedge \bigwedge_{\lambda(x)=n} \alpha(\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{z}_{\mathtt{t+1}}{-}n) \wedge \alpha(\mathtt{s'}_{\mathtt{t+1}}) \wedge \mathtt{I}(\mathtt{s'})_{\mathtt{t+1}}\\
&= \alpha(\mathtt{s}_\mathtt{t}) \wedge \alpha(\alpha_{\mathtt{t+1}}) \wedge \alpha(\mathtt{cc}_\mathtt{t}) \wedge (\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}}) \wedge \bigwedge_{\substack{\lambda(x)=id,\\ x\in\mathcal{X}\setminus\mathcal{O}}} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x}_\mathtt{t}) \wedge\\
&\quad \bigwedge_{\substack{\lambda(x)=x',\\ x,x'\in\mathcal{X}\setminus\mathcal{O}}} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x'}_{\mathtt{t+1}}) \wedge \bigwedge_{\substack{\lambda(x)=n,\\ x\in\mathcal{X}\setminus\mathcal{O}}} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{z}_{\mathtt{t+1}}{-}n) \wedge \alpha(\mathtt{s'}_{\mathtt{t+1}}) \wedge \mathtt{I}(\mathtt{s'})_{\mathtt{t+1}}
\end{aligned}$$

and its counterpart $e'{=}(\alpha(s), \alpha(\mathfrak{a}), \alpha(cc), \alpha(\lambda), \alpha(s')){\in}E'$:

$$\varphi^{action}(e') = \alpha(\mathtt{s}_\mathtt{t}) \wedge \alpha(\alpha_{\mathtt{t+1}}) \wedge \alpha(\mathtt{cc}_\mathtt{t}) \wedge (\mathtt{z}_\mathtt{t}{=}\mathtt{z}_{\mathtt{t+1}}) \wedge \bigwedge_{\alpha(\lambda)(x)=id} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x}_\mathtt{t}) \wedge$$
$$\bigwedge_{\alpha(\lambda)(x)=x'} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{x'}_{\mathtt{t+1}}) \wedge \bigwedge_{\alpha(\lambda)(x)=n} (\mathtt{x}_{\mathtt{t+1}}{=}\mathtt{z}_{\mathtt{t+1}}{-}n) \wedge \alpha(\mathtt{s'}_{\mathtt{t+1}}) \wedge \alpha(\mathtt{I(s')}_\mathtt{t})$$

Because $\mathcal{X}'{=}\mathcal{X}\backslash\mathcal{O}$, we have

$$\alpha(\varphi^{action}(e)) = \varphi^{action}(e')$$

For a delay transition in s, we have

$$\begin{aligned}
\alpha(\varphi^{delay}(s)) &= \alpha(\mathtt{s}_\mathtt{t}\wedge \bigwedge_{\mathfrak{a}\in\Sigma}\neg\alpha_{\mathtt{t+1}}\wedge(\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_{\mathtt{t+1}})\wedge \bigwedge_{x\in\mathcal{X}}(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_{\mathtt{t+1}})\wedge\mathtt{s}_{\mathtt{t+1}}\wedge\mathtt{I(s)}_{\mathtt{t+1}})\\
&= \alpha(\mathtt{s}_\mathtt{t})\wedge \bigwedge_{\mathfrak{a}\in\Sigma}\alpha(\neg\alpha_{\mathtt{t+1}})\wedge\alpha(\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_{\mathtt{t+1}})\wedge \bigwedge_{x\in\mathcal{X}}\alpha(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_{\mathtt{t+1}})\wedge\alpha(\mathtt{s}_{\mathtt{t+1}})\wedge\alpha(\mathtt{I(s)}_{\mathtt{t+1}})\\
&= \alpha(\mathtt{s}_\mathtt{t}) \wedge \bigwedge_{\substack{\mathfrak{a}\in\Sigma,\\ \alpha(\mathfrak{a})=id}}\neg\alpha_{\mathtt{t+1}}\wedge \bigwedge_{\substack{\mathfrak{a}\in\Sigma,\\ \alpha(\mathfrak{a})=\mathfrak{a}'}}(\neg\alpha'_{\mathtt{t+1}})\wedge(\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_{\mathtt{t+1}})\wedge\\
&\quad \bigwedge_{\substack{x\in\mathcal{X},\\ x\in\mathcal{X}\backslash\mathcal{O}}}(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_{\mathtt{t+1}})\wedge\alpha(\mathtt{s}_{\mathtt{t+1}})\wedge\alpha(\mathtt{I(s)}_{\mathtt{t+1}})
\end{aligned}$$

and for the corresponding delay transition in s', we have

$$\varphi^{delay}(s') = \alpha(\mathtt{s}_\mathtt{t})\wedge \bigwedge_{\mathfrak{a}\in\Sigma'}\neg\alpha_{\mathtt{t+1}}\wedge(\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_{\mathtt{t+1}})\wedge \bigwedge_{x\in\mathcal{X}'}(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_{\mathtt{t+1}})\wedge\alpha(\mathtt{s}_{\mathtt{t+1}})\wedge\alpha(\mathtt{I(s)}_{\mathtt{t+1}})$$

Because of (**), we have

$$\alpha(\varphi^{delay}(s)) = \varphi^{delay}(s')$$

Since there is a one-to-one relation between transitions in E and E', and by definition of $\vee$, we finally have

$$\alpha(\varphi^{trans}(\mathfrak{A})) = \varphi^{trans}(\mathfrak{A}')$$

3. Mutual exclusion of locations $\varphi^{location}$:

$$\begin{aligned}
\alpha(\varphi^{location}(\mathfrak{A})) &= \alpha(\bigvee_{s\in S}(\mathtt{s}_{\mathtt{t+1}}\wedge \bigwedge_{s'\in S,s'\neq s}\neg\mathtt{s}'_{\mathtt{t+1}}))\\
&= \bigvee_{s\in S}(\alpha(\mathtt{s}_{\mathtt{t+1}})\wedge \bigwedge_{s'\in S,s'\neq s}\alpha(\neg\mathtt{s}'_{\mathtt{t+1}}))\\
&= \bigvee_{\substack{s\in S\\ \alpha(s)=id}}(\mathtt{s}_\mathtt{t}\wedge \bigwedge_{\substack{s'\in S,s'\neq s\\ \alpha(s')=id}}\neg\mathtt{s'}_\mathtt{t}\wedge \bigwedge_{\substack{s'\in S,s'\neq s\\ \alpha(s')=\bar{s}\neq s}}\neg\bar{\mathtt{s}}_\mathtt{t})\vee\\
&\quad \bigvee_{\substack{s\in S\\ \alpha(s)=\hat{s}}}(\hat{\mathtt{s}}_\mathtt{t}\wedge \bigwedge_{\substack{s'\in S,s'\neq s\\ \alpha(s')=id}}\neg\mathtt{s'}_\mathtt{t}\wedge \bigwedge_{\substack{s'\in S,s'\neq s\\ \alpha(s')=\bar{s}\neq s}}\neg\bar{\mathtt{s}}_\mathtt{t})
\end{aligned}$$

$$\varphi^{location}(\mathfrak{A}')) = \bigvee_{s\in S'}(\mathtt{s}_{\mathtt{t+1}}\wedge \bigwedge_{s'\in S',s'\neq s}\neg\mathtt{s}'_{\mathtt{t+1}})$$

Because of (*), we have

$$\alpha(\varphi^{location}(\mathfrak{A})) = \varphi^{location}(\mathfrak{A}'))$$

4. Mutual exclusion of events φ^{mutex}:

$$\begin{aligned}
\alpha(\varphi^{mutex}(\mathfrak{A})) &= \alpha(\bigvee_{\mathfrak{a}\in\Sigma}(\alpha_{\mathtt{t+1}} \wedge \bigwedge_{\mathfrak{a}'\in\Sigma,\mathfrak{a}'\neq\mathfrak{a}} \neg\alpha'_{\mathtt{t+1}}) \vee \bigwedge_{\mathfrak{a}\in\Sigma}(\neg\alpha_{\mathtt{t+1}})) \\
&= \bigvee_{\mathfrak{a}\in\Sigma}(\alpha(\alpha_{\mathtt{t+1}}) \wedge \bigwedge_{\mathfrak{a}'\in\Sigma,\mathfrak{a}'\neq\mathfrak{a}} \alpha(\neg\alpha'_{\mathtt{t+1}})) \vee \bigwedge_{\mathfrak{a}\in\Sigma}\alpha(\neg\alpha_{\mathtt{t+1}}) \\
&= \bigvee_{\substack{\mathfrak{a}\in\Sigma\\ \alpha(\mathfrak{a})=id}} (\alpha_{\mathtt{t}} \wedge \bigwedge_{\substack{\mathfrak{a}'\in\Sigma,\mathfrak{a}'\neq\mathfrak{a}\\ \alpha(\mathfrak{a}')=id}} \neg\alpha'_{\mathtt{t}} \wedge \bigwedge_{\substack{\mathfrak{a}'\in\Sigma,\mathfrak{a}'\neq\mathfrak{a}\\ \alpha(\mathfrak{a}')=\bar{\mathfrak{a}}\neq\mathfrak{a}}} \neg\bar{\alpha}_{\mathtt{t}}) \vee \\
&\quad \bigvee_{\substack{\mathfrak{a}\in\Sigma\\ \alpha(\mathfrak{a})=\hat{\mathfrak{a}}}} (\hat{\alpha}_{\mathtt{t}} \wedge \bigwedge_{\substack{\mathfrak{a}'\in\Sigma,\mathfrak{a}'\neq\mathfrak{a}\\ \alpha(\mathfrak{a}')=id}} \neg\alpha'_{\mathtt{t}} \wedge \bigwedge_{\substack{\mathfrak{a}'\in\Sigma,\mathfrak{a}'\neq\mathfrak{a}\\ \alpha(\mathfrak{a}')=\bar{\mathfrak{a}}\neq\mathfrak{a}}} \neg\bar{\alpha}_{\mathtt{t}}) \vee \\
&\quad \bigwedge_{\substack{\mathfrak{a}\in\Sigma,\\ \alpha(\mathfrak{a})=id}} (\neg\mathfrak{a}) \wedge \bigwedge_{\substack{\mathfrak{a}\in\Sigma,\\ \alpha(\mathfrak{a})=\bar{\mathfrak{a}}}} (\neg\bar{\alpha}_{\mathtt{t+1}}) \\
\varphi^{mutex}(\mathfrak{A}')) &= \bigvee_{\mathfrak{a}\in\Sigma'}(\alpha_{\mathtt{t+1}} \wedge \bigwedge_{\mathfrak{a}'\in\Sigma',\mathfrak{a}'\neq\mathfrak{a}} \neg\alpha'_{\mathtt{t+1}}) \vee \bigwedge_{\mathfrak{a}\in\Sigma'}(\neg\alpha_{\mathtt{t+1}})
\end{aligned}$$

Because of (**), we have

$$\alpha(\varphi^{mutex}(\mathfrak{A})) = \varphi^{mutex}(\mathfrak{A}')),$$

From the four cases above, we get

$$\alpha(\varphi(\mathfrak{A})) = \varphi(\mathfrak{A}')$$

The argumentation for

$$\alpha(\varphi(\mathfrak{A})_k) = \varphi(\mathfrak{A}')_k$$

is similar.

Thus, the TA $\mathfrak{A}'$ satisfies the conditions (xx), and we have shown that MO preserves the formula representation and k-unfolding of TA. $\square$

Proof of (xxi). Let $\mathfrak{T}' = (S', s_0', \mathcal{P}, \mathcal{X}', I', \mathcal{D}', \#', E')$ be a TCA, with $S'{=}\alpha(S)$, $s_0'{=}\alpha(s_0)$, $\mathcal{P}'{=}\alpha(\mathcal{P})$, $\mathcal{X}'{=}\alpha(\mathcal{X})$, $I'(s){=}\alpha(I(s))$ for all $s{\in}S'$, $\mathcal{D}'{=}\alpha(\mathcal{D})$, $\#'(s){=}\alpha(\#(s))$ for all $s{\in}S'$, and $E'{=}\{(\alpha(s), \alpha(P), \alpha(dc), \alpha(cc), \alpha(\lambda), \alpha(s')) | (s, P, dc, cc, \lambda, s'){\in}E\}$. Let $\varphi(\mathfrak{T}')$ and $\varphi(\mathfrak{T}')_k$ be the formula representation and k-unfolding of $\mathfrak{T}'$. Observe that we have

$$\begin{aligned}
&S'{=}\alpha(S){=}\{s|s{\in}S, \alpha(s){=}id\}\cup\{s'|s{\in}S, \alpha(s){=}s'\}, && (\dagger)\\
&\Sigma'{=}\alpha(\Sigma){=}\{\mathfrak{a}|\mathfrak{a}{\in}\Sigma, \alpha(\mathfrak{a}){=}id\}\cup\{\mathfrak{a}'|\mathfrak{a}{\in}\Sigma, \alpha(\mathfrak{a}){=}\mathfrak{a}'\}, \text{ and} && (\ddagger)\\
&\mathcal{D}'{=}\alpha(\mathcal{D}){=}\{d|d{\in}\mathcal{D}, \alpha(d){=}id\}\cup\{d'|d{\in}\mathcal{D}, \alpha(d){=}d'\} && (\dagger\dagger)
\end{aligned}$$

We first show that $\varphi(\mathfrak{T}')\backslash_{dc}=\alpha(\varphi(\mathfrak{T}))\backslash_{dc}$. By definition of $\backslash_{dc}$, and Definitions 3.1.4 and 4.1.5, we have

$$\begin{aligned}
\alpha(\varphi(\mathfrak{T}))\backslash_{dc} &= \alpha(\varphi^{init}(\mathfrak{T}) \wedge \varphi^{trans}(\mathfrak{T}) \wedge \varphi^{location}(\mathfrak{T}) \wedge \varphi^{mutex}(\mathfrak{T}))\backslash_{dc}\\
&= \alpha(\varphi^{init}(\mathfrak{T}))\backslash_{dc} \wedge \alpha(\varphi^{trans}(\mathfrak{T}))\backslash_{dc} \wedge\\
&\quad\ \alpha(\varphi^{location}(\mathfrak{T}))\backslash_{dc} \wedge \alpha(\varphi^{mutex}(\mathfrak{T}))\backslash_{dc}\\
\varphi(\mathfrak{T}')\backslash_{dc} &= \varphi^{init}(\mathfrak{T}')\backslash_{dc} \wedge \varphi^{trans}(\mathfrak{T}')\backslash_{dc} \wedge \varphi^{location}(\mathfrak{T}')\backslash_{dc} \wedge \varphi^{mutex}(\mathfrak{T}')\backslash_{dc}
\end{aligned}$$

Consider the corresponding parts in $\alpha(\varphi(\mathfrak{T}))\backslash_{dc}$ and $\varphi(\mathfrak{T}')\backslash_{dc}$ separately

1. Initial constraints φ^{init}:

$$\begin{aligned}
\alpha(\varphi^{init}(\mathfrak{T}))\backslash_{dc} &= \alpha\big(\bar{\mathtt{s}}_0 \wedge \bigwedge_{s\in S, s\neq\bar{s}} \neg\mathtt{s}_0 \wedge \mathtt{I}(\bar{\mathtt{s}})_0 \wedge \bigwedge_{p\in\mathcal{P}} (\neg\mathtt{p}_0 \wedge (\mathtt{Dp}_0{=}\mathtt{n}^{\perp})) \wedge\\
&\quad \bigwedge_{d\in\mathcal{D}} (\neg\mathtt{d}_0 \wedge (\mathtt{Dd}_0{=}\mathtt{n}^{\perp})) \wedge (\mathtt{z}_0{=}0) \wedge \bigwedge_{x\in\mathcal{X}} (\mathtt{x}_0{=}0)\big)\backslash_{dc}\\
&= \alpha(\bar{\mathtt{s}}_0)\backslash_{dc} \wedge \bigwedge_{s\in S, s\neq\bar{s}} \alpha(\neg\mathtt{s}_0)\backslash_{dc} \wedge \alpha(\mathtt{I}(\bar{\mathtt{s}})_0)\backslash_{dc} \wedge \bigwedge_{p\in\mathcal{P}} \alpha(\neg\mathtt{p}_0)\backslash_{dc} \wedge\\
&\quad \bigwedge_{p\in\mathcal{P}} \alpha((\mathtt{Dp}_0{=}\mathtt{n}^{\perp}))\backslash_{dc} \wedge \bigwedge_{d\in\mathcal{D}} \alpha(\neg\mathtt{d}_0)\backslash_{dc} \wedge \bigwedge_{d\in\mathcal{D}} \alpha((\mathtt{Dd}_0{=}\mathtt{n}^{\perp}))\backslash_{dc} \wedge\\
&\quad \alpha((\mathtt{z}_0{=}0))\backslash_{dc} \wedge \bigwedge_{x\in\mathcal{X}} \alpha((\mathtt{x}_0{=}0))\backslash_{dc}\\
&= \alpha(\bar{\mathtt{s}}_0) \wedge \bigwedge_{\substack{s\in S, s\neq\bar{s},\\ \alpha(s)=id}} \neg\mathtt{s}_0 \wedge \bigwedge_{\substack{s\in S, s\neq\bar{s},\\ \alpha(s)=s'\neq\alpha(\bar{s})}} \neg\mathtt{s}'_0 \wedge \alpha(\mathtt{I}(\bar{\mathtt{s}})_0) \wedge \bigwedge_{\substack{p\in\mathcal{P},\\ \alpha(p)=id}} (\neg\mathtt{p}_0) \wedge\\
&\quad \bigwedge_{\substack{p\in\mathcal{P},\\ \alpha(p)=p'}} (\neg\mathtt{p'}_0) \wedge \bigwedge_{\substack{d\in\mathcal{D},\\ \alpha(d)=id}} (\neg\mathtt{d}_0) \wedge \bigwedge_{\substack{d\in\mathcal{D},\\ \alpha(d)=d'}} (\neg\mathtt{d'}_0) \wedge\\
&\quad (\mathtt{z}_0{=}0) \wedge \bigwedge_{x\in\mathcal{X}\backslash\mathcal{O}} (\mathtt{x}_0{=}0)
\end{aligned}$$

$$\begin{aligned}
\varphi^{init}(\mathfrak{T}')\backslash_{dc} &= \bar{\mathtt{s}}'_0\backslash_{dc} \wedge \bigwedge_{s\in S', s\neq\bar{s}'} \neg\mathtt{s}_0\backslash_{dc} \wedge \mathtt{I}(\bar{\mathtt{s}}')_0\backslash_{dc} \wedge \bigwedge_{p\in\mathcal{P}'} (\neg\mathtt{p}_0 \wedge (\mathtt{Dp}_0{=}\mathtt{n}^{\perp}))\backslash_{dc} \wedge\\
&\quad \bigwedge_{d\in\mathcal{D}'} (\neg\mathtt{d}_0 \wedge (\mathtt{Dd}_0{=}\mathtt{n}^{\perp}))\backslash_{dc} \wedge (\mathtt{z}_0{=}0)\backslash_{dc} \wedge \bigwedge_{x\in\mathcal{X}'} (\mathtt{x}_0{=}0)\backslash_{dc}\\
&= \bar{\mathtt{s}}'_0 \wedge \bigwedge_{s\in S', s\neq\bar{s}'} \neg\mathtt{s}_0 \wedge \mathtt{I}(\bar{\mathtt{s}}')_0 \wedge \bigwedge_{p\in\mathcal{P}'} (\neg\mathtt{p}_0) \wedge\\
&\quad \bigwedge_{d\in\mathcal{D}'} (\neg\mathtt{d}_0) \wedge (\mathtt{z}_0{=}0) \wedge \bigwedge_{x\in\mathcal{X}'} (\mathtt{x}_0{=}0)
\end{aligned}$$

By definition of $\mathfrak{T}'$, we have $\alpha(\bar{\mathtt{s}}_0) = \bar{\mathtt{s}}'_0$, and $\alpha(\mathtt{I}(\bar{\mathtt{s}})_0) = \mathtt{I}(\bar{\mathtt{s}}')_0$. Because of (†), (‡) and (††), and the fact that $\mathcal{X}'{=}\mathcal{X}\backslash\mathcal{O}$, we finally get

$$\alpha(\varphi^{init}(\mathfrak{T}))\backslash_{dc} = \varphi^{init}(\mathfrak{T}')\backslash_{dc}$$

2. Transition relation φ^{trans}:

$$\begin{aligned}
\alpha(\varphi^{trans}(\mathfrak{T}))\backslash_{dc} &= \alpha(\bigvee_{e\in E, e \text{ visible}} \varphi^{visible}(e) \vee \bigvee_{e\in E, e \text{ invisible}} \varphi^{invisible}(e))\backslash_{dc}\\
&= \bigvee_{e\in E, e \text{ visible}} \alpha(\varphi^{visible}(e))\backslash_{dc} \vee \bigvee_{e\in E, e \text{ invisible}} \alpha(\varphi^{invisible}(e))\backslash_{dc}
\end{aligned}$$

$$\varphi^{trans}(\mathfrak{T}')\backslash_{dc} = \bigvee_{e\in E',e\text{ visible}} \varphi^{visible}(e)\backslash_{dc} \vee \bigvee_{e\in E',e\text{ invisible}} \varphi^{invisible}(e)\backslash_{dc}$$

Consider a visible transition $e{=}(s, P, dc, cc, \lambda, s') \in E$:

$$\begin{aligned}
\alpha(\varphi^{visible}(e))\backslash_{dc} = {} & \alpha(\mathtt{s_t}\wedge \mathtt{I(s)_{t\Delta}}\wedge \bigwedge_{p\in P} \mathtt{p_{t+1}}\wedge \bigwedge_{p\notin P}\neg\mathtt{p_{t+1}}\wedge \bigwedge_{d\notin\#(s')} \neg\mathtt{d_{t+1}}\wedge\mathtt{dc_{t+1}}\wedge \\
& \mathtt{cc_{t\Delta}}\wedge(\mathtt{z_t}<\mathtt{z_{t+1}})\wedge \bigwedge_{\lambda(x)=id} (\mathtt{x_{t+1}}=\mathtt{x_t})\wedge \bigwedge_{\lambda(x)=x'} (\mathtt{x_{t+1}}=\mathtt{x'_{t+1}})\wedge \\
& \bigwedge_{\lambda(x)=n} (\mathtt{x_{t+1}}=\mathtt{z_{t+1}}-n)\wedge\mathtt{s'_{t+1}}\wedge\mathtt{I(s')_{t+1}})\backslash_{dc} \\
= {} & \alpha(\mathtt{s_t})\wedge\alpha(\mathtt{I(s)_{t\Delta}})\wedge \bigwedge_{\substack{p\in P,\\ \alpha(p)=id}} \mathtt{p_{t+1}} \wedge \bigwedge_{\substack{p\in P,\\ \alpha(p)=p'}} \mathtt{p'_{t+1}} \wedge \bigwedge_{\substack{p\notin P,\\ \alpha(p)=id}} \neg\mathtt{p_{t+1}} \wedge \\
& \bigwedge_{\substack{p\notin P,\\ \alpha(p)=p'\notin\alpha(P)}} \neg\mathtt{p'_{t+1}} \wedge \bigwedge_{\substack{d\notin\#(s'),\\ \alpha(d)=id}} (\neg\mathtt{d_{t+1}}) \wedge \bigwedge_{\substack{d\notin\#(s'),\\ \alpha(d)=d'}} (\neg\mathtt{d'_{t+1}}) \wedge \\
& \alpha(\mathtt{cc_{t\Delta}}) \wedge(\mathtt{z_t}<\mathtt{z_{t+1}}) \wedge \bigwedge_{\substack{\lambda(x)=id,\\ x\in\mathcal{X}\backslash\mathcal{O}}} (\mathtt{x_{t+1}}=\mathtt{x_t}) \wedge \\
& \bigwedge_{\substack{\lambda(x)=x',\\ x,x'\in\mathcal{X}\backslash\mathcal{O}}} (\mathtt{x_{t+1}}=\mathtt{x'_{t+1}}) \wedge \bigwedge_{\substack{\lambda(x)=n,\\ x\in\mathcal{X}\backslash\mathcal{O}}} (\mathtt{x_{t+1}}=\mathtt{z_{t+1}}-n) \wedge \alpha(\mathtt{s'_{t+1}}) \wedge \mathtt{I(s')_{t+1}}
\end{aligned}$$

and its counterpart $e'{=}(\alpha(s), \alpha(P), \alpha(dc), \alpha(cc), \alpha(\lambda), \alpha(s'))\in E'$:

$$\begin{aligned}
\varphi^{visible}(e')\backslash_{dc} = {} & \alpha(\mathtt{s_t})\backslash_{dc}\wedge\alpha(\mathtt{I(s)_{t\Delta}})\backslash_{dc}\wedge \bigwedge_{p\in\alpha(P)} \mathtt{p_{t+1}}\backslash_{dc}\wedge \bigwedge_{p\notin\alpha(P)} (\neg\mathtt{p_{t+1}})\backslash_{dc}\wedge \\
& \bigwedge_{d\notin\#(\alpha(s'))} (\neg\mathtt{d_{t+1}})\backslash_{dc}\wedge\alpha(\mathtt{dc_{t+1}})\backslash_{dc}\wedge\alpha(\mathtt{cc_{t\Delta}})\backslash_{dc}\wedge\alpha(\mathtt{z_t}<\mathtt{z_{t+1}})\backslash_{dc}\wedge \\
& \bigwedge_{\alpha(\lambda)(x)=id} (\mathtt{x_{t+1}}=\mathtt{x_t})\backslash_{dc}\wedge \bigwedge_{\alpha(\lambda)(x)=x'} (\mathtt{x_{t+1}}=\mathtt{x'_{t+1}})\backslash_{dc}\wedge \\
& \bigwedge_{\alpha(\lambda)(x)=n} (\mathtt{x_{t+1}}=\mathtt{z_{t+1}}-n)\backslash_{dc}\wedge\alpha(\mathtt{s'_{t+1}})\backslash_{dc}\wedge\alpha(\mathtt{I(s')_{t+1}})\backslash_{dc} \\
= {} & \alpha(\mathtt{s_t})\wedge\alpha(\mathtt{I(s)_{t\Delta}})\wedge \bigwedge_{p\in\alpha(P)} \mathtt{p_{t+1}}\wedge \bigwedge_{p\notin\alpha(P)} (\neg\mathtt{p_{t+1}})\wedge \\
& \bigwedge_{d\notin\#(\alpha(s'))} (\neg\mathtt{d_{t+1}})\wedge\alpha(\mathtt{cc_{t\Delta}})\wedge(\mathtt{z_t}<\mathtt{z_{t+1}})\wedge \\
& \bigwedge_{\alpha(\lambda)(x)=id} (\mathtt{x_{t+1}}=\mathtt{x_t})\wedge \bigwedge_{\alpha(\lambda)(x)=x'} (\mathtt{x_{t+1}}=\mathtt{x'_{t+1}})\wedge \\
& \bigwedge_{\alpha(\lambda)(x)=n} (\mathtt{x_{t+1}}=\mathtt{z_{t+1}}-n)\wedge\alpha(\mathtt{s'_{t+1}})\wedge\alpha(\mathtt{I(s')_{t+1}})
\end{aligned}$$

Because of (‡) and (††), and the fact that $\mathcal{X}'{=}\mathcal{X}\backslash\mathcal{O}$, we have

$$\alpha(\varphi^{visible}(e))\backslash_{dc} = \varphi^{visible}(e')\backslash_{dc}$$

Equivalently, we can show for an invisible transition $e{=}(s, \emptyset, \texttt{true}, cc, \lambda, s')$ and its counterpart $e'{=}(\alpha(s), \emptyset, \texttt{true}, \alpha(cc), \alpha(\lambda), \alpha(s'))$ that

$$\alpha(\varphi^{invisible}(e)\backslash_{dc} = \varphi^{invisible}(e')\backslash_{dc}$$

Since there is a one-to-one relation between transitions in E and E', and by definition of $\vee$, we finally have

$$\alpha(\varphi^{trans}(\mathfrak{T}))\backslash_{dc} = \varphi^{trans}(\mathfrak{T}')\backslash_{dc}$$

3. Mutual exclusion of locations $\varphi^{location}$: because $\backslash_{dc}$ does not change $\varphi^{mutex}(\mathfrak{T}')$ or $\alpha(\varphi^{mutex}(\mathfrak{T}))$, equivalently to the case for TA above, we have

$$\alpha(\varphi^{location}(\mathfrak{T}))\backslash_{dc} = \varphi^{location}(\mathfrak{T}')\backslash_{dc}$$

4. Data consistency constraints φ^{mutex}: trivially,

$$\alpha(\varphi^{mutex}(\mathfrak{T})))\backslash_{dc} = \texttt{true} = \varphi^{mutex}(\mathfrak{T}')\backslash_{dc}$$

From the four cases above, we get

$$\alpha(\varphi(\mathfrak{T}))\backslash_{dc} = \varphi(\mathfrak{T}')\backslash_{dc}$$

With a similar argumentation, we get

$$\alpha(\varphi(\mathfrak{T})_k)\backslash_{dc}{=}\varphi(\mathfrak{T}')_k\backslash_{dc}$$

Thus, the TCA $\mathfrak{T}'$ satisfies the conditions (xxi), and we have shown that MO preserves the formula representation and k-unfolding of TCA, up to data constraints.
□

Proposition A.2.8 (Commuting Subdiagram). The subdiagram (*ii*) of Figure A.2 is *a partially commuting diagram.*[7]

Proof. This follows directly from Proposition A.2.7. □

We now have all the results to give the proof of Theorem 4.1.8.

Proof of Theorem 4.1.8. For the abstraction by omission to be correct, every finite run in the original system $\mathfrak{S}$ has to be reproducible in the abstract system $\widetilde{\mathfrak{S}}$. We show this by defining a homomorphism h_R between original and abstract sets of runs $Run_{\mathfrak{S},k}$ and $Run_{\widetilde{\mathfrak{S}},k}$, such that Figure A.2 commutes.

[7]Here, "partially commuting" means that every model $\sigma{\in}\mathcal{V}(\varphi(\mathfrak{T})_k)$ is also a model of $\mathcal{V}(\varphi(\widetilde{\mathfrak{T}})_k)$, but not necessarily vice versa.

Let $\mathfrak{S}$ be a real-time system, with formula representation $\varphi(\mathfrak{T})$ and k-unfolding $\varphi(\mathfrak{S})_k$, let $\mathcal{V}(\varphi(\mathfrak{S})_k)$ be the set of models of $\varphi(\mathfrak{S})_k$, let $Run_{\mathfrak{S},k}$ be the set of finite runs of length k of $\mathfrak{S}$.

Let α be an abstraction function, with γ and $\mathcal{O}$ as in Definition 4.1.5, let $\widetilde{\mathfrak{S}}$ be the abstract system that results from applying α to the formula representation $\varphi(\mathfrak{S})$ and the k-unfolding $\varphi(\mathfrak{S})_k$, that means $\varphi(\widetilde{\mathfrak{S}}){=}\alpha(\varphi(\mathfrak{S}))$ and $\varphi(\widetilde{\mathfrak{S}})_k{=}\alpha(\varphi(\mathfrak{S})_k)$, cf. Proposition A.2.7, let $\mathcal{V}(\varphi(\widetilde{\mathfrak{S}})_k)$ be the set of models of $\varphi(\widetilde{\mathfrak{S}})_k$, and $Run_{\widetilde{\mathfrak{S}},k}$ the set of finite runs of length k of $\widetilde{\mathfrak{S}}$. Let $\xi{:}\mathcal{V}(\varphi(\mathfrak{S})_k){\rightarrow}\mathcal{V}(\varphi(\widetilde{\mathfrak{S}})_k)$ be a mapping assigning to each interpretation $\sigma{\in}\mathcal{V}(\varphi(\mathfrak{S})_k)$ the interpretation $\widetilde{\sigma}{\in}\mathcal{V}(\varphi(\widetilde{\mathfrak{S}})_k)$, which is obtained from restricting σ to the variables in $\varphi(\widetilde{\mathfrak{S}})_k$.[8]

We define a homomorphism h_R (cf. Definition A.2.2) as

$$\begin{aligned} h_R{:}Run_{\mathfrak{S},k} &{\rightarrow} Run_{\widetilde{\mathfrak{S}},k} \\ h_R(r) &= \downarrow^{\sigma}_{r}(\xi(\downarrow^{r}_{\sigma}(r))) \end{aligned}$$

(cf. Lemmas A.1.4 and A.1.7). That means, a run $r_{\widetilde{\sigma}} \in Run_{\widetilde{\mathfrak{S}},k}$ is obtained from a run $r{\in}Run_{\mathfrak{S},k}$ by mapping r to the derived interpretation $\downarrow^{r}_{\sigma}(r){=}\sigma_r{\in}\mathcal{V}(\varphi(\mathfrak{S})_k)$ (Definition A.1.6), reducing it to the interpretation $\xi(\downarrow^{r}_{\sigma}(r)){=}\widetilde{\sigma}{\in}\mathcal{V}(\varphi(\widetilde{\mathfrak{S}})_k)$ over the variables in $\varphi(\widetilde{\mathfrak{S}})_k$, and mapping it to the derived run $\downarrow^{\sigma}_{r}(\xi(\downarrow^{r}_{\sigma}(r))){=}r_{\widetilde{\sigma}} \in Run_{\widetilde{\mathfrak{S}},k}$ (Definition A.1.2).

We define $\gamma_S{:}S{\rightarrow}\widetilde{S}$, with $\gamma_S(s){=}\widetilde{s}$ iff $\gamma(s){=}\widetilde{s}$,[9], $\gamma_\Sigma{:}\Sigma{\rightarrow}\widetilde{\Sigma}$, with $\gamma_\Sigma(\mathfrak{a}){=}\widetilde{\mathfrak{a}}$ iff $\gamma(\mathfrak{a}){=}\widetilde{\mathfrak{a}}$, $\gamma_P{:}p{\rightarrow}\widetilde{p}$, with $\gamma_P(p){=}\widetilde{p}$ iff $\gamma(p){=}\widetilde{p}$, and $\gamma_{\mathcal{D}}{:}d{\rightarrow}\widetilde{d}$, with $\gamma_{\mathcal{D}}(d){=}\widetilde{d}$ iff $\gamma(d){=}\widetilde{d}$. With this, h_R is a homomorphism as defined in Definition A.2.2. Together with the results of Proposition A.2.3, Proposition A.2.7, and Proposition A.2.8, Figure A.2 is a commuting diagram, that means every run of the original system $\mathfrak{S}$ is reproducible in the abstract system $\widetilde{\mathfrak{S}}$, and therefore the abstraction by omission is correct. □

[8] $\widetilde{\sigma}$ is well-defined, as by definition of α: $Vars(\alpha(\varphi(\mathfrak{S})_k)){\subseteq}Vars(\varphi(\mathfrak{T})_k)$, cf. Proposition A.2.7.

[9] Remember that we lifted α to constituents of TCA, cf. Remark A.2.6.

Abstract

The increasing size of present-day embedded software systems makes verification of these systems an increasingly difficult task. Even more, not only the size of systems increases, but to faithfully model and analyse real-life applications, an increasing number of features and formalisms need to be developed and supported. The two most important features demanded from embedded software systems are that they need to involve handling of dense real-time, and that they can be developed in a modular, component-based way. The last point amounts to specifying the system by a set of components (implementing the behaviour), together with a set of component connectors (implementing the coordination patterns for inter-component communication protocols).

To ensure that the behaviour of the final system is correct (that means, behaves as expected) and safe (that means, nothing bad can ever happen), it needs to be verified before it is being put into operation. To this end, two things are needed: first, formal models that are powerful enough to faithfully describe all aspects of the system, and in particular support handling of dense real-time as well as constructs to combine components and connectors. Second, formal methods to analyse the formal model of the system and verify that it satisfies certain properties, in particular including correctness of the coordination pattern.

In this thesis, we propose both formal models and formal methods to model and analyse component-based real-time systems and their coordination patterns. We present three formal models of real-time systems: Timed Automata, Timed Constraint Automata, and Timed Network Automata, which have different modelling power with respect to communication, communication primitives, and expressible constraints on the communication. We then present a translation for each of these formal models into a representation in propositional logic with linear arithmetic, which allows to use well-established SAT and SMT solver tools to analyse real-time properties of the underlying system. We give a correctness proof for the representation, which shows that results established for the representation carry over to the respective formal model.

We then present an abstraction technique that works on the representation, and reduces the size of the system by removing parts that are considered irrelevant to the verification of a particular property. This allows to further increase the manageable system size. We prove the abstract system to be an over-approximation of the original system, such that infeasibility of some erroneous behaviour (up to a certain execution bound) in the abstract system entails infeasibility of the erroneous behaviour (up to a certain execution bound) in the original system. Finally, we prove the applicability and usability of our framework with a tool implementation, that supports the design and analysis process of component-based real-time systems.

Samenvatting

De toenemende omvang van huidige ingebedde software systemen maakt de verificatie van deze systemen een steeds moeilijker opgave. Sterker nog, niet alleen de omvang van de systemen neemt toe, maar om toepassingen uit de praktijk zo waarheidsgetrouw mogelijk te modelleren en te analyseren moeten ook een toenemend aantal nieuwe mechanismen en formalismen ontworpen en ondersteund worden. De twee belangrijkste functies van ingebedde software systemen zijn dat ze in real-time moeten kunnen reageren op invoer uit de omgeving, en dat ze in een modulair, op componenten gebaseerde manier ontwikkeld kunnen worden. Het laatste komt neer op het specificeren van het systeem door middel van afzonderlijke componenten (elk waarvan een bepaald aspect van het gedrag implementeert), samen met zogeheten connectoren die de coördinatiepatronen voor de communicatie tussen de componenten implementeren.

Om ervoor te zorgen dat het gedrag van het uiteindelijke systeem correct is (dat betekent, zich gedraagt zoals verwacht) en veilig (dat betekent, dat er geen fouten optreden), moet het systeem gecontroleerd worden voordat het in werking kan worden gesteld. Hiervoor zijn twee dingen nodig: ten eerste, formele modellen die expressief genoeg zijn om alle aspecten van het systeem zo waarheidsgetrouw mogelijk te beschrijven, en die in het bijzonder zowel real-time mechanismen bevatten als constructies om componenten en connectoren te combineren. Ten tweede, formele methoden om het formele model van het systeem te analyseren en te controleren of deze voldoet aan bepaalde eigenschappen, zoals met name juistheid van de coördinatiepatronen.

In dit proefschrift introduceren wij verschillende technieken met verschillende uitdrukkingsmogelijkheden voor het modelleren en analyseren van de coördinatiepatronen van op componenten gebaseerde real-time systemen. We presenteren de volgende drie operationele modellen van real-time systemen: Timed Automata, Timed Constraint Automata, en Timed Network Automata. Vervolgens geven we een vertaling voor elk van deze formele modellen naar een representatie in propositielogica met lineaire wiskunde, die het mogelijk maakt om beproefde op logica gebaseerde technieken, zogeheten SAT en SMT solvers, te gebruiken voor de analyse van de real-time eigenschappen van de betreffende operationele modellen. We geven een bewijs van juistheid voor de logische representatie, waaruit blijkt dat de resultaten verkregen door middel van deze technieken eveneens gelden voor de verschillende

operationele modellen.

Vervolgens presenteren wij een techniek voor verdere abstractie van de logische representatie die ons in staat stelt de omvang en complexiteit van deze te verminderen, door het verwijderen van die onderdelen die worden beschouwd als niet relevant voor de verificatie van een bepaalde eigenschap. We bewijzen dat deze techniek inderdaad leidt tot een over-abstractie die ons in staat stelt te concluderen dat een fout die in het abstracte systeem niet kan optreden ook in het oorspronkelijke system niet kan optreden. Tenslotte betogen we de toepasbaarheid en bruikbaarheid van onze theorie met een implementatie die het ontwerp en de analyse van op componenten gebaseerde real-time systemen ondersteunt.

Curriculum Vitae

1979 Born on 20 July in Bremerhaven, Germany

1992-1999 High School (Gymnasium), Nordenham, Germany

1999-2006 Diplom (equivalent to master's degree) in Computer Science, Carl-von-Ossietzky Universität, Oldenburg, Germany

Major in Theoretical Computer Science

Thesis title: *SAT-based Verification for Abstraction Refinement*

2006-2011 PhD student at Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands, supervised by Prof. Dr. Frank S. de Boer

2011- Scientific Staff Member, Carl-von-Ossietzky Universität Oldenburg, Germany

Index

Bibliography

[ABdBR04] Farhad Arbab, Christel Baier, Frank S. de Boer, and J.J.M.M. Rutten. Models and temporal logics for timed component connectors. In *SEFM*, pages 198–207. IEEE Computer Society, 2004. 41

[ABdBR07] Farhad Arbab, Christel Baier, Frank S. de Boer, and J.J.M.M. Rutten. Models and temporal logical specifications for timed component connectors. *Software and System Modeling*, 6(1):59–82, 2007. 3, 4, 18, 27, 41, 65, 104, 123, 125

[ABRS04] Farhad Arbab, Christel Baier, J.J.M.M. Rutten, and M. Sirjani. Modeling component connectors in $\mathcal{R}$eo by constraint automata (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 97:25–46, 2004. 4, 18, 27, 88

[ABSS96] Ahmet F. Ates, Murat Bilgic, Senro Saito, and Behçet Sarikaya. Using timed csp for specification verification and simulation of multimedia synchronization. *IEEE Journal on Selected Areas in Communications*, 14(1):126–137, 1996. 108

[ACKS02] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded model checking for timed systems. In D. Peled and M.Y. Vardi, editors, *FORTE*, volume 2529 of *LNCS*, pages 243–259. Springer, November 2002. 43, 59, 62

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 3, 4, 7, 8, 11, 12, 16, 18, 41, 123

[Alu99] Rajeev Alur. Timed automata. In N. Halbwachs and D. Peled, editors, *CAV*, volume 1633 of *LNCS*, pages 8–22. Springer, 1999. 3, 4, 9, 14, 15, 16, 18, 41, 123

[AM04] Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In Bernardo and Corradini [BC04], pages 1–24. 9, 12, 17, 18

[AMM$^+$09] Farhad Arbab, Sun Meng, Young-Joo Moon, Marta Z. Kwiatkowska, and Hongyang Qu. Reo2mc: a tool chain for performance analysis of coordination models. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 287–288. ACM, 2009. 88

[ANT] Antlr parser generator. release 3.3. `http://www.antlr.org`. 92, 97

[Arb98] Farhad Arbab. What do you mean, coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI*, pages 11–22, 1998. 2

[Arb04] Farhad Arbab. $\mathcal{R}$eo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004. 18, 34, 88

[BBBC94] Howard Bowman, Lynne Blair, Gordon S. Blair, and Amanda G. Chetwynd. A formal description technique supporting expression of quality of service and media synchronisation. In David Hutchison, André A. S. Danthine, Helmut Leopold, and Geoff Coulson, editors, *COST 237 Workshop*, volume 882 of *Lecture Notes in Computer Science*, pages 145–167. Springer, 1994. 108

[BBBC97] G.S. Blair, L. Blair, H. Bowman, and A. Chetwynd. *Formal Specification of Distributed Multimedia Systems.* University College London Press, September 1997. 108

[BBKK09] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. A uniform framework for modeling and verifying components and connectors. In J. Field and V.T. Vasconcelos, editors, *COORDINATION*, volume 5521 of *LNCS*, pages 247–267. Springer, 2009. 125

[BC04] Marco Bernardo and Flavio Corradini, editors. *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*. Springer, 2004. 159, 161

[BCC$^+$03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003. 3, 43, 57

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In R. Cleaveland, editor, *TACAS*, volume 1579 of *LNCS*, pages 193–207, London, UK, 1999. Springer. 43, 57, 60

[BDL04] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Bernardo and Corradini [BC04], pages 200–236. 18

[Bea03] Danièle Beauquier. On probabilistic timed automata. *Theor. Comput. Sci.*, 292(1):65–84, 2003. 16

[BFK$^+$98] Howard Bowman, Giorgio P. Faconti, Joost-Pieter Katoen, Diego Latella, and Mieke Massink. Automatic verification of a lip-synchronisation protocol using uppaal. *Formal Asp. Comput.*, 10(5-6):550–575, 1998. 108, 109, 110, 116, 120, 121

[Bie09] Armin Biere. Bounded model checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009. 57

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008. 12, 13, 43

[BPM] BPMN Eclipse plugin. `http://www.eclipse.org/bpmn/`. 88

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. 57

[BS00] Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000. 16

[BSAR06] Christel Baier, M. Sirjani, Farhad Arbab, and J.J.M.M. Rutten. Modeling component connectors in $\mathcal{R}$eo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006. 34

[BZM08] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. Csisat: Interpolation for la+euf. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 304–308. Springer, 2008. 77

[CBRZ01] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001. 3, 43, 57

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. 126

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992. 126

[CCA07] Dave Clarke, David Costa, and Farhad Arbab. Connector colouring I: Synchronisation and context dependency. *Sci. Comput. Program.*, 66(3):205–225, 2007. 4, 28, 29, 34, 35

[CCK$^+$02] Pankaj Chauhan, Edmund M. Clarke, James H. Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated Abstraction Refinement for Model Checking Large State Spaces Using SAT Based Conflict Analysis. In Mark Aagaard and John W. O'Leary, editors, *FMCAD*, volume 2517 of *Lecture Notes in Computer Science*, pages 33–51. Springer, 2002. 84, 126

[CGJ$^+$03] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003. 3, 67, 68, 72, 81, 84

[CGKS02] Edmund M. Clarke, Anubhav Gupta, James H. Kukula, and Ofer Strichman. SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2002. 126

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. 43, 68

[CKA10] Behnaz Changizi, Natallia Kokash, and Farhad Arbab. A unified toolset for business process model formalization. Tool Paper, 2010. 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA). 88

[Cos10] David Costa. *Formal Models for Component Connectors*. PhD thesis, Vrije Universiteit Amsterdam, 2010. 28

[CPLA09] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Deconstructing reo. *Electr. Notes Theor. Comput. Sci.*, 229(2):43–58, 2009. 34

[Cra57] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957. 3, 5, 68, 75, 76

[csi] CSIsat: A Tool for LA+EUF Interpolation. `http://www.sosy-lab.org/~dbeyer/CSIsat/`. 77, 83

[Ecl] Eclipse platform. `http://www.eclipse.org`. 88, 124

[ECT] Extensible Coordination Tools. `http://reo.project.cwi.nl/`. 6, 88, 124

[EKS06] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2006. 125

[FKPY07] Elena Fersman, Pavel Krcál, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Inf. Comput.*, 205(8):1149–1172, 2007. 16

[FOC] FOCI: an interpolating prover. `http://www.kenmcmil.com/foci.html`. 43, 77, 83

[Fok00] Wan Fokkink. *Introduction to Process Algebra.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000. 104, 120

[GJSB05] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification.* The Java Series. Addison-Wesley, Massachusetts, third edition, 2005. 88

[GN07] Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007. 62

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997. 67

[GS05] Gregor Gößler and Joseph Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005. 16

[Häh93] R. Hähnle. Short CNF in finitely-valued logics. In H.J. Komorowski and Z.W. Ras, editors, *ISMIS*, volume 689 of *LNCS*, pages 49–58. Springer, 1993. 62

[HJMM04] T.A. Henzinger, R. Jhala, R. Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In N.D. Jones and X. Leroy, editors, *POPL*, pages 232–244. ACM, 2004. 3, 67

[Kem09] Stephanie Kemper. SAT-based verification for timed component connectors. *Electr. Notes Theor. Comput. Sci.*, 255:103–118, 2009. 65, 85, 163

[Kem10] Stephanie Kemper. Compositional construction of real-time dataflow networks. In Dave Clarke and Gul A. Agha, editors, *COORDINATION*, volume 6116 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2010. 3, 4, 5, 28, 40, 41, 65, 123

[Kem11] Stephanie Kemper. SAT-based Verification for Timed Component Connectors. *Science of Computer Programming*, 2011. This is an extended version [Kem09]. 3, 4, 5, 6, 18, 19, 27, 65, 68, 85, 87, 123

[KKdV10] Natallia Kokash, Christian Krause, and Erik P. de Vink. Data-aware design and verification of service compositions with reo and mcrl2. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 2406–2413. ACM, 2010. 88

[KLP10] Piotr Kordy, Rom Langerak, and Jan Willem Polderman. Re-verification of a lip synchronization protocol using robust reachability. *CoRR*, abs/1003.0431, 2010. 108

[KLSV03a] Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. The theory of timed I/O automata. Technical Report MIT-LCS-TR-917, MIT Laboratory for Computer Science, 2003. 19

[KLSV03b] Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In *RTSS*, pages 166–177. IEEE Computer Society, 2003. 19

[KNSS02] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theor. Comput. Sci.*, 282(1):101–150, 2002. 16

[KP07] Stephanie Kemper and A. Platzer. SAT-based abstraction refinement for real-time systems. *Electr. Notes Theor. Comput. Sci.*, 182:107–122, 2007. 5, 12, 65, 68, 85

[Kra11] Christian Krause. *Reconfigurable Component Connectors.* PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), 2011. 88, 90, 95

[LM87] Kim Guldstrand Larsen and Robin Milner. Verifying a protocol using relativized bisimulation. In Thomas Ottmann, editor, *ICALP*, volume 267 of *Lecture Notes in Computer Science*, pages 126–135. Springer, 1987. 104

[MA03] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003. 126

[mat] The MathSAT 4 SMT solver. `http://mathsat4.disi.unitn.it`. 43, 77, 83, 93, 102

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, Pittsburgh, USA, Norwell, MA, USA, 1993. 57

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003. 75, 76

[McM04] Kenneth L. McMillan. An interpolating theorem prover. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *LNCS*, pages 16–30. Springer, 2004. 76

[McM05a] Kenneth L. McMillan. Applications of craig interpolants in model checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005. 76

[McM05b] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005. 75, 76

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982. 120

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. 104, 105, 120

[MLWZ01] Huadong Ma, Liang Li, Jianzhong Wang, and Naijun Zhan. Automatic synthesis of the dc specifications of lip synchronisation protocol. In *APSEC*, pages 371–. IEEE Computer Society, 2001. 108

[MMZ$^+$01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001. 62

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005. 62

[PSHA09] Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat, and Farhad Arbab. Automated analysis of $\mathcal{R}$eo circuits using symbolic execution. *Electr. Notes Theor. Comput. Sci.*, 255:137–158, 2009. 4, 19, 41

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997. 76

[Reg93] Tim Regan. Multimedia in temporal LOTOS: A lip-synchronization algorithm. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *PSTV*, volume C-16 of *IFIP Transactions*, pages 127–142. North-Holland, 1993. 108, 109

[SHH92] Jean-Bernard Stefani, Laurent Hazard, and François Horn. Computational model for distributed multimedia applications based on a synchronous programming language. *Computer Communications*, 15(2):114–128, 1992. 108, 109

[Tri99] Stavros Tripakis. Verifying progress in timed systems. In Joost-Pieter Katoen, editor, *ARTS*, volume 1601 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 1999. 14

[upp] Uppaal: modeling, simulation and verification of real-time system. `http://www.uppaal.com/`. 43, 120

Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty