Synthesis of Reconfiguration Charts[1][2]

Technical report

Tobias Eckardt and Stefan Henkler
Heinz Nixdorf Institute
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
tobie, shenkler@upb.de

Paderborn, 2nd January 2010

[1]This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

[2]This work was developed in the project "ENTIME: Entwurfstechnik Intelligente Mechatronik" (Design Methods for Intelligent Mechatronic Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, "Investing in your future".

Abstract

Component-based architectures are widely used in embedded systems. For managing complexity and improving quality separation of concerns is one of the most important principles. For one component, separation of concerns is realized by defining the overall component functionality by separated protocol behaviors. One of the main challenges of applying separation of concerns is the later automatic composition of the separated, maybe interdependent concerns which is not supported by current component-based approaches. Moreover, the complexity of real-time distributed embedded systems requires to consider safety requirements for the composition of the separated concerns. We present an approach which addresses these problems by a well-defined automatic composition of protocol behaviors with respect to interdependent concerns. The composition is performed by taking a proper refinement relation into account so that analysis results of the separated concerns are preserved which is essential for safety critical systems.

Contents

Chapter 1

Introduction

When developing software for mechatronic real-time systems, such as computer controlled train systems, cars or air planes, the following three aspects have to be taken into account.

Mechatronic systems often act in safety critical areas where high financial damage arises or even life is threatened if the system fails. Consequently, a *high level of quality* has to be established. In order to achieve this, correct functionality also has to be guaranteed where hard deadlines have to be met by the system. Therefore, *timing aspects* for the behavior of the system have to be considered during the development. If an airbag control unit in a car, for example, triggers the airbag ignition milliseconds too late, the airbag system is worthless. Furthermore, the integration of self-optimization in mechatronic systems requires to change the system's structure at runtime. As a result, *runtime reconfigurations* of controllers also have to be regarded in the scope of the behavioral specification of such systems.

Dealing with these aspects generally results in a complex development process which compromises the required quality of the system. This quality is even more endangered by the fact that the development process has to include all three disciplines of mechatronics, mechanical, electrical and software engineering [SW07].

To handle this complexity and thus improve quality, while still considering the aspects mentioned above, the following techniques have evolved in the field of software engineering.

To deal with safety concerns, one possibility is to formalize both requirements and the system's behavioral model, in order to formally proof the fulfillment of those requirements for the system specification. This technique, commonly known as *model checking* [GV08], has become the technique of choice in the field of formal verification and has also successfully been transferred to real-time systems [ACD90, HNSY92] including tool support [Yov97, BDL04].

Regarding the development process, the principle of *separation of concerns* is one of the first explicitly named techniques to encounter complexity when developing computer programs [Dij76]. The basic idea behind it is to set the focus on one aspect at a time, and examine the dependencies between different aspects later. This idea of separation goes along with the practice of *component-based software development* [Szy98]. Here, a software component is used to encapsulate independent related functionality of the system, while providing dependent functionality to other components through well defined interfaces. The basic idea of both techniques is to decompose the system into manageable parts first, and later recompose these parts into one coherent system.

To make use of these techniques during the development of reconfigurable, mechatronic systems, the *Collaborative Research Centre 614 – "Self-optimizing Concepts and Structures in Mechanical Engineering" (CRC 614)*[1] is developing a modeling language named MECHATRONIC UML [BGT05] based on the *Unified Modeling Language (UML)* [OMG09]. Apart from applying model-based software development techniques where possible, MECHATRONIC UML strictly follows the separation of concerns paradigm. This is realized by separating the system specification into components with explicitly defined communication interfaces and using *real-time coordination patterns* to specify the inter-component communication. This way system developers are able to design separate parts of the system independently from each other by having different views on the system specification [NKF94, GV06]. Furthermore, partitioning the complex system specification into several small parts makes it possible to use *compositional model checking* techniques [GTB+03]. These techniques are unavoidable as traditional model checking approaches are not feasible for complex systems due to the state explosion problem leading to scalability problems [ABB+98].

The price of being able to specify separate views on the system independently from each other is the additional effort which has to be spent when combining these views to an overall system view. The composition has to include both the architectural and the behavioral part of the system specification. While the architectural composition is most often straight forward, the behavioral composition, commonly referred to by the term behavioral synthesis, is a complex task on which a lot of research effort has been spent in different fields of software engineering (see chapter 5 for related work). During this process of combining separate behavioral specifications, dependencies between those have to be integrated into the system specification while model checking results of the separate parts have to be preserved. For self-optimizing mechatronic systems, additionally, the system reconfigurations have to be included in the synthesis process, as those are also part of the behavioral specification of a component.

[1] http://www.sfb614.de

In this paper we present a synthesis approach for independent real-time behavioral specifications with overlapping requirements. This means, that dependencies between the individual behavioral specifications are modeled explicitly on top of those specifications, in a way that preserves the independent models. As the approach is expressed in the context of MECHATRONIC UML, we are also able to include reconfiguration behavior. Furthermore, the approach guarantees that model checking results of the separate behavioral specifications are preserved as long as there are no contradictions amongst the specifications. This means that the externally visible behavior of each behavioral model is not influenced by the synthesis procedure.

We give a more detailed overview of the synthesis approach in section 1.2. The type of systems dealt with in this paper will be exemplified by a small case study in the next section. The described system will serve as a running example throughout the whole paper.

1.1 Case Study

The research initiative *Neue Bahntechnik Paderborn (NBP – New Rail Technology Paderborn)*[2] is developing a rail-based transportation system at the University of Paderborn since 1997. The system requires most modern techniques of mechatronics – mechanical engineering, electrical engineering and software engineering. The basis of the system are driverless vehicles, called *RailCabs* (Figure 1.1), which travel on demand without a fixed schedule and are able to carry either passengers or goods. The system and especially the techniques for developing mechatronic systems are studied with intense effort by the *Collaborative Research Centre 614 – "Self-optimizing Concepts and Structures in Mechanical Engineering" (CRC 614)*.

Compared to conventional railway systems, RailCabs have two main advantages, which will be of special interest throughout this paper. The first one is the ability to travel in *automatic convoy operation* mode. In this mode, RailCabs are able to build convoys dynamically, without physical contact between each other. Operating this way increases traffic flow and minimizes energy consumption because of slipstream traveling, where the air resistance is reduced to a minimum. The second advantage is the *active track guidance* system. With this, each RailCab is able to steer its wheels to adapt to the actual track trace and condition. This increases driving comfort, decreases wear out and makes passive switches possible because RailCabs can then decide to change their direction on a track, even in convoy operation.

[2] http://www.railcab.de

Figure 1.1: Two RailCabs Driving in Convoy

For the active track guidance system, the RailCab needs track information in advance, in order to be able to steer its wheels correctly. To receive this track information, the RailCab can either communicate with an external entity or measure the track itself using its internal track sensors. In this example, the external entity is represented by so-called *base stations*. When receiving the track information in advance via a base station, the RailCab can achieve the best comfort but also has to establish and uphold a connection to a base station which might not always be possible. Using its internal track sensors instead, the RailCab is independent, but can only adapt to those track characteristics it is able to process in the short period of time between receiving the signal input of the sensor and the wheel actually being on that piece of track. In both cases the RailCab must provide the ability of *reconfiguration*, as it possibly has to reconfigure its suspension mode, activity of track sensors and radio activity for the base station communication, corresponding to whether a base station is available or not. A simple example of four RailCabs RC1 to RC4 and four base stations BS1 to BS4 is shown in figure 1.2 where each RailCab is connected to the nearest base station.

For traveling with automatic convoy operation, a RailCab needs information about other RailCabs driving nearby, which constitute potential convoy partners. In this example, this information is obtained from the base station the RailCab is connected to. Once a convoy partner is found, the (in driving direction) rear RailCab contacts the front RailCab and proposes to initiate a convoy. The front RailCab is able to accept or reject the convoy, depending on its schedule or other factors. If the front RailCab accepts the convoy, it has to do so within a certain

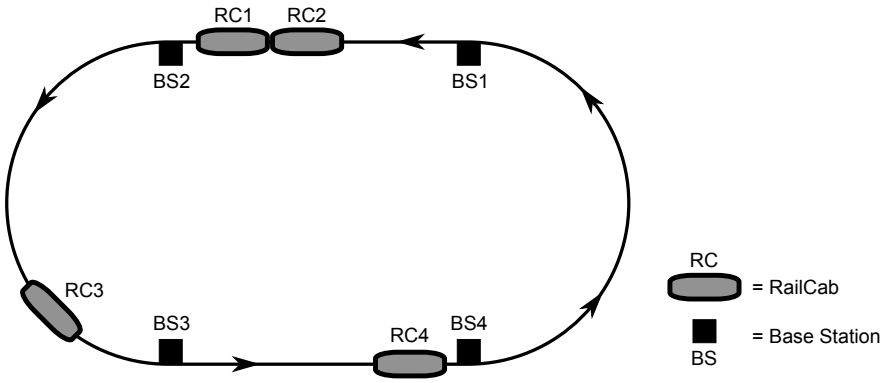

Figure 1.2: RailCabs and Base Stations on a Track

time interval and communicate this to the rear RailCab. This can then again start the convoy operation by driving into the slipstream of the front RailCab and by reconfiguring its speed control unit from a velocity based mode to a distance based mode. This reconfiguration is necessary because driving at the exact same velocity is very difficult and only slight differences in the velocity of two RailCabs driving in a convoy may cause collisions within seconds. Therefore, the rear RailCab has to measure the distance to the front RailCab and adjust its velocity according to that value. As a collision can lead to high financial damage or even life threatening situations, the convoy operation contains safety critical aspects with hard real-time requirements. In figure 1.2 RailCabs RC1 and RC2 operate in convoy, where RC1 is the front and RC2 the rear RailCab.

This example is further refined in Chapter 2 where the basic concepts of MECHATRONIC UML are explained. In the next section we introduce the objectives of this paper.

1.2 Objectives and Basic Approach

In MECHATRONIC UML separation of concerns is realized by applying *component based development* [Szy98] and in accordance with that by rigorously separating inter-component from intra-component behavior. Following this concept, the system is decomposed into participating components and *real-time coordination patterns* [GTB+03], which define how components interact with each other.

Corresponding to the case study described in the preceding section, we specify two components BaseStation and RailCab (Figure 1.3) and two coordination patterns Registration and Convoy, which define the before described communication behavior between RailCabs and base stations.

In real-time coordination patterns, *roles* are used to abstract from the actual components participating in one coordination pattern. This way, it is possible to specify and verify coordination patterns independently from other coordination patterns and component definitions and therefore to reduce complexity. In figure 1.3 the participating roles of the Registration pattern are registrar and registree; the roles of the Convoy pattern are front and rear.

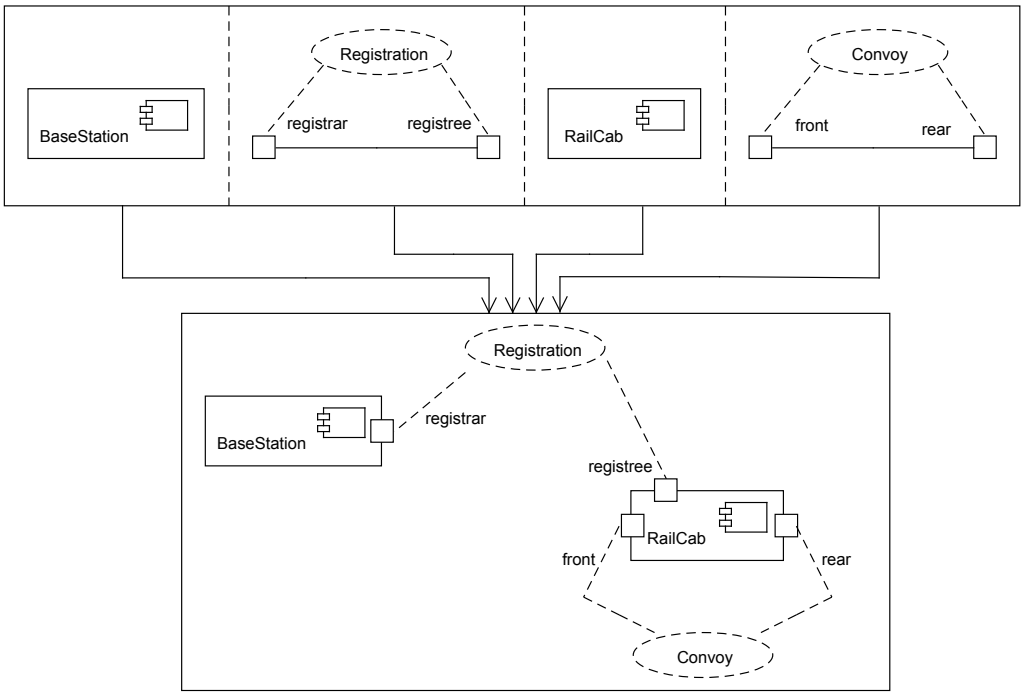

Figure 1.3: Combining Separate Specifications in MechantronicUML

To obtain an overall system specification later in the development process, the separated components and coordination patterns have to be combined again (Figure 1.3). The problem which inherently arises at this point is that separate parts of the system were specified as independent from each other when they are in fact not. This means that during the process of combining the separate parts of the system, additional dependencies between the particular specifications have to be integrated. At the same time, the externally visible behavior of the particular behavioral specifications may not be changed in order to preserve verification results [GTB+03].

In the overall system view of the RailCab example (Figure 1.3), the RailCab component takes part in both, the Registration and the Convoy pattern. While those patterns have been specified independently from each other, a system re-

quirement states that in convoy operation mode, each participating RailCab has to be registered to a base station. Accordingly, a dependency between both patterns exists, when applied by the RailCab component. As a result, the behavior of the registree role and the behavior of the rear role have to be refined and synchronized with each other when applied by the RailCab component in order to fulfill the system requirements. Still, it has to be regarded that the externally visible behavior of the RailCab component does not change. If this process of refinement and synchronization is performed manually, it is a time consuming and error-prone process. Consequently, this implies the necessity for automation in order to guarantee the required quality of the developed systems.

In the approach proposed in this paper, we separate the specification of dependencies and the specification of the pattern role behaviors in order to perform an automatic synthesis for the overall component behavior. After the synthesis is performed, it is further checked if the synthesized component behavior refines each of the particular pattern role behaviors properly.

The idea which forms the foundation of our approach was introduced by Giese and Vilbig in [GV06] for untimed behavioral models. In their approach the synthesis basically consists of three steps: (1) Construct an explicit behavioral model for the parallel execution of the participating behavioral models, (2) restrict the behavior described by this model in terms of forbidden state combinations and (3) verify that the resulting behavioral model still refines each of the input models properly. Seibel extends this approach in [Sei07] in two ways: (1) He employs real-time behavioral models as input models and (2) further allows to restrict the behavior in terms of (timed) event sequences.

In the developed solution, we adapt these ideas to the context of MECHATRONIC UML. This includes (1) the definition of suitable refinement relation, (2) the employment of a suitable and also more efficient abstraction of the timed behavioral models which is needed to perform the refinement check and (3) the integration of reconfiguration behavior. The result is a fully automatic synthesis algorithm where dependencies between separate behavioral specifications are specified explicitly by so-called *composition rules* (cf. [TOHS99]). Accordingly, the input for the algorithm are composition rules and separate behavioral specifications (Figure 1.4). If the synthesis is possible without violating the externally visible behavior of any of the input specifications, the output is one parallelly composed behavioral model which combines all of the input specifications as well as the composition rules. If the synthesis is not possible, the algorithm returns a conflict description indicating the reason for the impossibility.

With this algorithm system developers are able to attach composition rules to components of their views to specify dependencies to other views. These composition rules are later factored in automatically during the composition of the full system specification by the synthesis algorithm if none of the behavioral specifica-

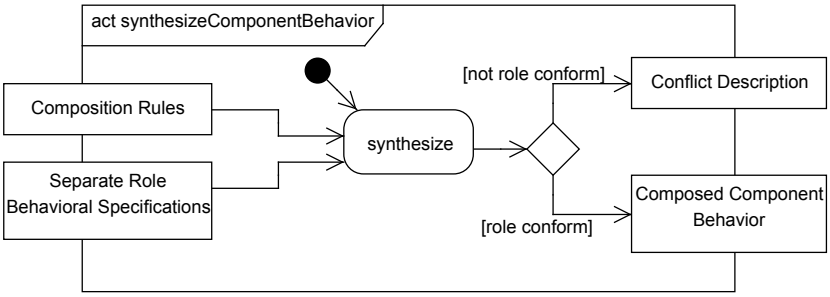

Figure 1.4: Activity Diagram Illustrating the Basic Synthesis Approach

tions is violated. If a behavioral specification is violated, the developer can use the conflict description to find the part of the specification which causes the violation in order to resolve the conflict. By giving the dependencies explicitly by means of composition rules, a detailed conflict description can be obtained referring to the conflicting composition rules. This way, the manual interaction during the composition of the views is reduced to the essential parts: (1) specifying dependencies as composition rules, (2) analyzing conflict descriptions and (3) resolving these conflicts. The second and the third action however, are only necessary if a conflict actually exists between the input specifications.

1.3 Structure

this paper is structured as follows. In the next chapter we introduce those concepts of MECHATRONIC UML which are relevant for this paper. In addition to that, we also introduce the underlying timed automaton model which forms the basis of all the real-time behavioral specifications of MECHATRONIC UML used in this paper. As the proposed synthesis procedure preserves the relevant behavior of the coordination roles by means of verification results, we also describe the specification language which is used for real-time model checking in MECHATRONIC UML.

In chapter 3 we describe the synthesis approach. For this, we start with the definition of the composition rule formalism. After that we show how an explicit model for the parallel execution of the individual pattern behaviors is constructed and how the composition rules are applied to this model. Finally, we define which properties of the role behaviors have to be preserved in order to preserve the relevant behavior of the coordination roles. In this context we also describe a semi-automatic procedure which guarantees that the relevant behavior is preserved by the synthesized behavioral model.

In chapter 4 we present results of the evaluation activities which were performed during the work on this paper. This includes the evaluation of the proposed refinement relation as well as the evaluation of the model applied for formal reasoning on the model for the parallel execution of the role behaviors.

In the last two chapters we describe related approaches (chapter 5) and finally conclude the proposed concepts also referring to future work (chapter 6).

Chapter 2

Fundamental Concepts

As already presented in the introduction, MECHATRONIC UML forms an appropriate approach for the model-based development of reconfigurable, mechatronic systems. In this chapter, we present the basic concepts of MECHATRONIC UML, while focusing on those parts of the language which are relevant for this paper.

The modeling techniques of MECHATRONIC UML can be divided into modeling of structure (section 2.2.2) and modeling of behavior (section 2.2.1), of which the latter also includes the modeling of system reconfigurations (section 2.2.3).

The synthesis procedure proposed in this paper computes the overall behavior of a component participating in two or more real-time coordination patterns with respect to specified composition rules (see section 1.2). In accordance with this, we focus on the behavioral models of MECHATRONIC UML and their underlying concepts in the following.

In MECHATRONIC UML, real-time behavior is specified using realtime statecharts [GB03] whose semantics is based on timed automata [AD90, HNSY92]. Derived from this, we define the core of the synthesis procedure on timed automata (see chapter 3) and therefore begin this chapter by defining the syntax and semantics of timed automata.

2.1 Timed Automata

Timed automata, first introduced by Alur and Dill in [AD90] and later varied by Henzinger et al. in [HNSY92], have evolved to the behavioral model of choice for formal reasoning of real-time systems. They extend the common automaton formalism by the notion of time, such that it is possible to specify system behaviors with respect to timing constraints. With these timing constraints we can express, for example, how long a system is allowed to rest in a certain state or in which time intervals it is allowed to switch between two system states.

During the last two decades several variants of timed automata have been proposed (see [AM04] for an overview). The two variants most discussed in literature are the originally proposed *timed büchi automata* [AD90] and the later proposed *timed safety automata* [HNSY92]. Derived from basic automata theory, both variants use nodes, called locations, and edges, called transitions, to reason about system states and transitions between those states. In order to relate the behavior of the automaton to the notion of time, each automaton can define one or more clocks. These clocks are referred to at transitions and in locations in order to make the execution of those transitions and the resting in locations dependent on certain values of those clocks.

The main difference between timed büchi automata and timed safety automata is the way progress conditions are expressed.

As one may assume by the name, timed büchi automata use *accepting locations* derived from büchi automata [Bü62]. If a location of the automaton is an accepting location it has to be visited continually over and over again. Transferring this to the notion of time, this also means that a system can rest in such a location forever. In the timed büchi automaton for the simple convoy coordination behavior (Figure 2.1) accepting locations are noConvoy and convoy, but not processing. Accordingly, the system can rest in locations noConvoy and convoy forever or visit them infinitely often over and over again. The location processing on the other hand can only be left as long as the value of clock c is smaller than 1000 time units. This is indirectly specified using so-called time guards at transitions rejectConvoy and startConvoy.

Timed safety automata, however, do not have accepting locations. Instead, *location invariants* are attached to locations to specify the allowed time interval the automaton is allowed to rest in that location. The system therefore has to leave a location before its invariant is violated. The timed safety automaton for the simple convoy coordination behavior (Figure 2.2) has exactly the same semantic as the timed büchi automaton described before. But this time, instead of indirectly specifying at outgoing transitions that location processing has to be left within 1000 time units, a time invariant is directly attached to the concerned location.

Summarizing this, in timed büchi automata progress conditions are specified globally over the whole automaton and in timed safety automata they are specified locally from each location's perspective. As the latter approach is argued to be more suitable for real-time specifications [Hen92] (although less expressive in general [HKWT95]) and is also said to be more intuitive from the modeling perspective [Pet99, pp. 6–7], timed safety automata are the ones which are applied in several real-time verification tools like UPPAAL [BDL04] and KRONOS [Yov97].

As mechatronic real-time systems often act in safety critical areas, formal verification of the behavioral specification of those systems is unavoid-

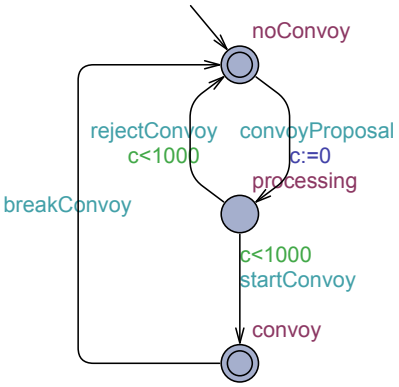

Figure 2.1: Timed Büchi Automaton

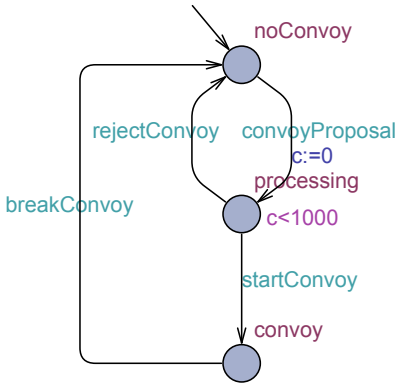

Figure 2.2: Timed Safety Automaton

able to guarantee correct functionality of the software running on those systems (see chapter 1). Accordingly, the ability of real-time model checking has also been integrated in MECHATRONIC UML's model for behavioral specifications [GTB+03, BGHS04], which are realtime statecharts. In order to use existing tools for this and with the same argument that timed safety automata have been applied in those tools, realtime statecharts are also based on timed safety automata. Consequently, the model we apply in this paper is also the one of timed safety automata and we refer to them by timed automata from here on.

2.1.1 Syntax and Semantics

Informally, a timed automaton consists of finite sets of *locations*, *transitions* and real-valued *clocks*. Starting in the *initial location*, it may either rest in a location or switch between locations using transitions and corresponding event occurrences. *Events* are modeled using a synchronous channel concept, where events can either be thrown using the special symbol "!" or received using the special symbol "?". The example automaton in figure 2.3 describes the behavior of the front role of a convoy coordination (see section 1.1). It has three locations noConvoy, processing and convoy, where noConvoy is the initial location. If the front role receives a convoyProposal, it switches to location processing. From there on, it can either accept the convoy proposal by sending a startConvoy and switching to convoy or it can reject the convoy switching through the *silent transition* back to noConvoy. A silent transition is a transition which is not connected to an event occurrence. Once the automaton is in convoy, it may arbitrarily decide to break the convoy by sending a breakConvoy event.

Time is modeled using *time guards*, *clock resets* and *location invariants*. Initially, all clocks' values are set to zero. From then on, time can only pass, i.e. all

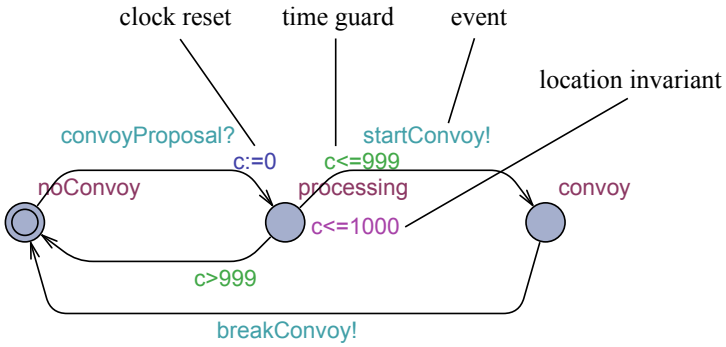

Figure 2.3: Example of a Timed Automaton

clocks' values increase by the same value, while the automaton rests in a location. This means that time does not pass, while the automaton is executing a transition. During the run of an automaton, a clock can be reset to zero again, using a *clock reset* connected to a transition. The execution of a transition can be constrained with respect to one or more clocks using *time guards*. A time guard defines a time interval for each clock using equalities and inequalities. If, for a certain clock valuation, the time guard of a transition evaluates to true, that transition becomes *enabled* and the automaton may decide to switch to the target location. It must not do so immediately, though. Finally, *location invariants* may be used to describe progress conditions. A location invariant describes an upper bound for the clock values in a certain location. This means that the automaton is only allowed to rest in the constrained location while its invariant evaluates to true or with other words, the automaton has to leave the location before its invariant is violated.

In the convoy example (Figure 2.3) the one clock c is reset to zero when entering the processing location. As described by the time guard c<=999, the automaton is able to send a startConvoy only as long as the value of c is smaller or equal to 999. If the value is greater than 999 (c>999), the automaton is only able to switch back to noConvoy, taking the corresponding silent transition. It also has to do so within one time unit, as otherwise the location invariant c<=1000 of the processing location would be violated.

To define the syntax of a timed automaton formally, we first need to define the syntax of location invariants and time guards by the notion of *clock constraints*. Similar to [BLR05, BDFP00, LMST03], we explicitly distinguish between general, diagonal-free and downwards closed clock constraints as defined below.

Definition 2.1.1 (General Clock Constraint)
For a set C of clocks, the set $\Phi(C)$ of general clock constraints *is inductively defined by the grammar*

$$\varphi ::= x \sim n \mid x - y \sim n \mid \varphi \wedge \varphi \mid true \mid false,$$

where $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$, $n \in \mathbb{N}$.

Accordingly, the most general form of clock constraints does not only allow to specify time intervals for clocks in terms of equalities and inequalities with respect to one clock. It also allows to specify constraints on the difference of two clocks in the form of $x - y \sim n$. Furthermore, any conjunction of two clock constraints is again a clock constraint. The special symbols $true$ and $false$ describe the clock constraints which are always $true$ or always $false$ without regarding any of the clocks' values. General clock constraints will be used as time guards attached to transitions in timed automata.

Definition 2.1.2 (Diagonal-free Clock Constraint)
For a set C of clocks, the set $\Phi_{df}(C) \subset \Phi(C)$ of diagonal-free clock constraints *is inductively defined by the grammar*

$$\varphi ::= x \sim n \mid \varphi \wedge \varphi \mid true \mid false,$$

where $x \in C$, $\sim \in \{\leq, <, =, >, \geq\}$, $n \in \mathbb{N}$.

The set of *diagonal-free clock constraints* is a proper subset of the set of general clock constraints, where equalities and inequalities of the form $x - y \sim n$ are not contained. Diagonal-free clock constraints will be of interest in the next section, where we introduce abstraction methods for timed automata.

Definition 2.1.3 (Downwards Closed Clock Constraint)
For a set C of clocks, the set $\Phi_{dc}(C) \subset \Phi(C)$ of downwards closed clock constraints *is inductively defined by the grammar*

$$\varphi ::= x \sim n \mid x - y \sim n \mid \varphi \wedge \varphi \mid true,$$

where $x, y \in C$, $\sim \in \{\leq, <\}$, $n \in \mathbb{N}$.

The definition of *downwards closed clock constraints* allows the diagonal form of clock constraints but does not allow to define lower bounds for clock values of the form $x \sim n$ or $x - y \sim n$, where $\sim \in \{=, >, \geq\}$. In other words, only upper bounds for clocks or differences between clocks can be specified. This form of clock constraints is used for location invariants, as it simplifies the semantic when location invariants are combined with clock resets. This way, clock resets cannot

lead to the impossibility of executing a transition, which would be possible if the location invariant of the target location did not allow the concerned clock to be zero. Furthermore, the special value $false$ is not allowed, as a location whose invariant is always false would be unreachable by definition.

As we now have proper formal definitions for the different forms of clock constraints we can proceed with the formal definition of the syntax of a timed automaton.

Definition 2.1.4 (Timed Automaton)
A Timed Automaton A *is a tuple* $(L, l^0, \Sigma, C, I, T)$ *where*

- L *is the set of locations*
- $l^0 \in L$ *is the initial location,*
- Σ *is the finite set of events where the symbol* τ *is used for internal events (silent transitions),*
- $I : L \rightarrow \Phi_{dc}(C)$ *assigns each location a location invariant as a downwards closed clock constraint,*
- C *is the finite set of clocks, and*
- $T \subseteq L \times \Sigma \times \Phi(C) \times 2^C \times L$ *is the finite set of transitions* $t = (l, e, g, r, l') \in T$ *with*
  - $l \in L$ *the source location,*
  - $e \in \Sigma$ *the related event,*
  - $g \in \Phi(C)$ *the time guard as a general clock constraint,*
  - $r \subseteq C$ *a set of clocks to be reset, and*
  - $l' \in L$ *the target location.*

In accordance with [HNSY92, BY03], we do only allow to model upper bounds for location invariants. Lower bounds can be expressed by using time guards at the incoming transitions of a location. Consequently, restricting location invariants to downwards closed clock constraints does not change the expressiveness of timed automata, but simplifies modeling and semantical analysis, as this way, a clock reset cannot influence the executability of a transition.

Furthermore, observe that, although disjunctions are not allowed in general clock constraints, they can be modeled implicitly for time guards of transitions. This can be achieved by modeling two separate transitions with the same event occurrence, clock resets and source and target location, but with different time guards. This way, only one of the time guards has to be fulfilled in order to enable

the corresponding transition, which is semantically equivalent to one transition with a disjunction in its time guard.

In order to formally define the semantics of timed automata, we first have to define the semantics of clock valuations, clock resets and the necessary operations on clocks.

Definition 2.1.5 (Clock Valuation)
For a set C of clocks, a clock valuation $\nu \in \Psi(C)$ *is a function*

$$\nu : C \rightarrow \mathbb{R}_+$$

assigning a non-negative real-value to each clock in C.

Informally, a *clock valuation* describes a point in time of the automaton by defining a certain value for each clock of the automaton. This point in time must not necessarily be unique, though, as clock resets can lead to the same clock assignment for different points in time of the automaton (see Definition 2.1.7, *clock reset evaluation*). Figure 2.4 illustrates three possible clock valuations ν_0, ν_1 and ν_2 for $C = \{x, y\}$, where $\nu_0(x) = \nu_0(y) = 0$, $\nu_1(x) = \nu_1(y) = 1$, $\nu_2(x) = 2.5$ and $\nu_2(y) = 1$.

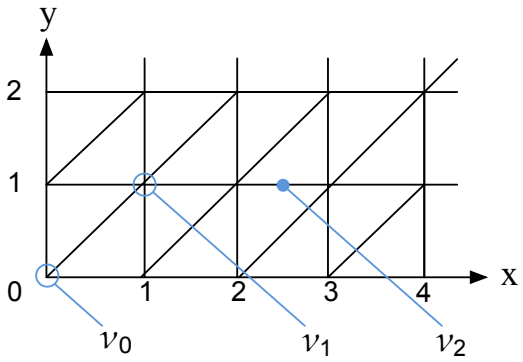

Figure 2.4: Illustration of Clock Valuations for Two Clocks x and y

On the basis of clock valuations, we can now also define operations on clock valuations in terms of addition and appliance of clock resets.

Definition 2.1.6 (Addition on Clock Valuations)
For a clock valuation $\nu \in \Psi(C)$ *and a non-negative real-value $\delta \in \mathbb{R}_+$, we define the addition $\nu + \delta$ by*

$$\forall c \in C : (\nu + \delta)(c) = \nu(c) + \delta.$$

Addition on clock valuation is used to describe the flow of time, where each clocks runs with the same speed, i.e. each clock's valuation is increased by the same value. For the clock valuations ν_0 and ν_1 of figure 2.4, ν_1 can be interpreted as the the clock valuation of ν_0 after one time unit, i.e. $\nu_1 = \nu_0 + 1$.

Definition 2.1.7 (Clock Reset Evaluation)
For a clock reset defined as a subset $Y \subseteq C$, the corresponding clock reset evaluation*, denoted $\nu[Y := 0]$, is defined by*

$$\forall c \in C : \nu[Y := 0](c) = \begin{cases} 0, & c \in Y, \\ \nu(c), & c \notin Y. \end{cases}$$

According to the above definition, a *clock reset evaluation* sets all values of the clocks defined by the clock reset to zero and agrees with ν for the rest of the clocks. This way, it might appear that different points in time have the same clock valuation for the clocks of the automaton.

To exemplify this using the illustration of figure 2.4, assume a global clock g which can never be reset and therefore represents the overall time passed by. This clock would form a third dimension in the illustration. The initial values of all clocks are 0.

Now, after one time unit has passed by, the clock values of the regular clocks x and y are equal to ν_1 and the value of the global clock g is equal to 1. Furthermore, assume that a clock reset $\nu_1[\{x, y\} := 0]$ will be performed at that point in time. Now the regular clocks again have the initial valuation ν_0, while the global clock g, representing the overall time passed by, has the valuation 1. Consequently, different points in time of the automaton, can have the same clock valuations.

As we have introduced the notion of clock valuations and addition on clock valuations above, we proceed with the definition of the semantics of the evaluation of a clock constraint.

Definition 2.1.8 (Clock Constraint Evaluation)
For a clock constraint $\varphi \in \Phi(C)$ we inductively define its evaluation as a function $\varphi : \Psi(C) \rightarrow \{true, false\}$ by differentiating between the following cases

$$\varphi(\nu) = \begin{cases} true, & \textit{iff } \varphi \textit{ is the literal } true, \\ false, & \textit{iff } \varphi \textit{ is the literal } false, \\ \nu(x) \sim n, & \textit{iff } \varphi \textit{ is of the form } x \sim n, \\ \nu(x) - \nu(y) \sim n, & \textit{iff } \varphi \textit{ is of the form } x - y \sim n, \\ \neg\varphi_1(\nu), & \textit{iff } \varphi \textit{ is of the form } \neg\varphi_1, \\ \varphi_1(\nu) \wedge \varphi_2(\nu), & \textit{iff } \varphi \textit{ is of the form } \varphi_1 \wedge \varphi_2. \end{cases}$$

where $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$.

Informally, a *clock constraint evaluation* $\varphi(\nu)$ defines for the corresponding clock constraint φ and an arbitrary clock valuation ν, if ν satisfies φ or not. By definition of the general clock constraint (Definition 2.1.1), this may have different forms which have to be distinguished here. While $true$, $false$, the complement and the join operation follow the conventional rules of Boolean algebra, $x \sim n$ and $x - y \sim n$ have to be evaluated using the corresponding clock valuations $\nu(x)$ and $\nu(y)$. As clock constraints are used by guards g and location invariants $I(l)$ we also use the notations $g(\nu)$ and $I(l)(\nu)$ to describe their corresponding evaluations.

We can now proceed with the definition of the semantics of a timed automaton by mapping it to an infinite discrete transition system.

Definition 2.1.9 (Timed Automaton Semantics)
The semantics of a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ *is defined by mapping it to an infinite transition system (also called timed transition system)* $S_A = (Q_A, q^0, \Sigma, T_\delta, T_\Sigma)$ *where*

- $Q_A \subseteq L \times \Psi(C)$ *is the infinite set of timed system* states $s = (l, \nu)$ *with* $l \in L$ *and* $\nu \in \Psi(C)$ *and* $s \in Q_A$ *iff* s *is reachable through* T_δ *or* T_Σ*,*
- $q^0 = (l^0, \nu_0) \in Q_A$ *is the* initial state *where* $\forall c \in C : \nu_0(c) = 0$ *and* l^0 *is the initial location of* A*,*
- Σ *the set of* events *equal to the set of events of the corresponding timed automaton,*
- $T_\delta \subseteq Q_A \times \mathbb{R}_+ \times Q_A$ *is the infinite set of* delay transitions $t_\delta = (q, \delta, q')$*, where*
  - $q = (l, \nu)$ *is the source state,*
  - δ *is representing the elapsed time as a non-negative real value,*
  - $q' = (l, \nu + \delta)$ *is the target state,*

  and $t_\delta \in T_\delta \Leftrightarrow \ \forall\, \delta' \in \mathbb{R}_+, 0 \leq \delta' \leq \delta : I(l)(\nu + \delta') = true$*,*
- $T_\Sigma \subseteq Q_A \times \Sigma \times Q_A$ *is the infinite set of* event transitions $t_\Sigma = (q, e, q')$*, where*
  - $q = (l, \nu)$ *is the source state,*
  - $e \in \Sigma$ *is the corresponding event,*
  - $q' = (l', \mu)$ *is the target state,*

  and $t_\Sigma \in T_\Sigma$ *iff* $\exists\, t = (l, e, g, r, l') \in T$*, such that*

- $I(l)(\nu) = true$, *and*
- $g(\nu) = true$, *and*
- $I(l')(\mu) = true$, *with* $\mu = \nu[r := 0]$.

In the corresponding transition system of a timed automaton A, timed system *states* have the form (l, ν), which means that a system state is the location the timed automaton rests in and a possible valuation for the clocks C of the automaton. Furthermore, two types of transitions exist, namely delay transitions and event transitions. *Delay transitions* represent the elapse of time while the automaton rests in a certain location. However, the time elapse may not violate the invariant of that location. The *event transitions* represent location changes on event occurrences corresponding to the transitions of the automaton. Accordingly, they have to take the time guards and the clock resets of the corresponding timed automaton transition into account.

To exemplify the timed automaton semantics defined above, we use again the example of the convoy coordination behavior (Figure 2.3) and its corresponding (infinite) timed transition system depicted in figure 2.5. As the set of clocks of the automaton only consists of the clock c, the possible clock valuations ν also only consist of one value defining the value of the clock c. Note the notation for clock values in the figure c=v abbreviates $\nu(c) = v$ for some $v \in \mathbb{R}_+$. Furthermore, we denote a state of the system consisting of a location loc and a value val for the clock c with (loc,val).

As the initial location of the automaton is noConvoy, the initial state of the transition system is (noConvoy,0). Following the delay transitions from that state in downwards direction, there exists an infinite number of states in the timed transition system for the following two reasons: (1) the automaton can rest in this location for an infinite time, as there is no location invariant connected to noConvoy and (2) for two states (noConvoy,v_1) and (noConvoy,v_2) there is always a third state (noConvoy,v_3) with $v_1 < v_3 < v_2$, as the clock values are real-values. In all of these noConvoy states there is an event transition leading to (processing,0), because the corresponding timed automaton transition (noConvoy,convoyProposal?,$true$,{c},processing) has no time guard and resets c. Following the delay transitions in this state, there also exist infinitely many states for the location processing, but only up the value $\nu(c) = 1000$. Observe that for the processing states where $\nu(c) = v \leq 999$ there exists an event transition ((processing,v),startConvoy!,(convoy,v)), while for the states where $\nu(c) = v > 999$ there exists an event transition ((processing,v),τ,(noConvoy,v)). Once in convoy, the automaton can again rest in that location for an infinite time and from each state it can switch back to noConvoy with breakConvoy! not changing the value of the clock c.

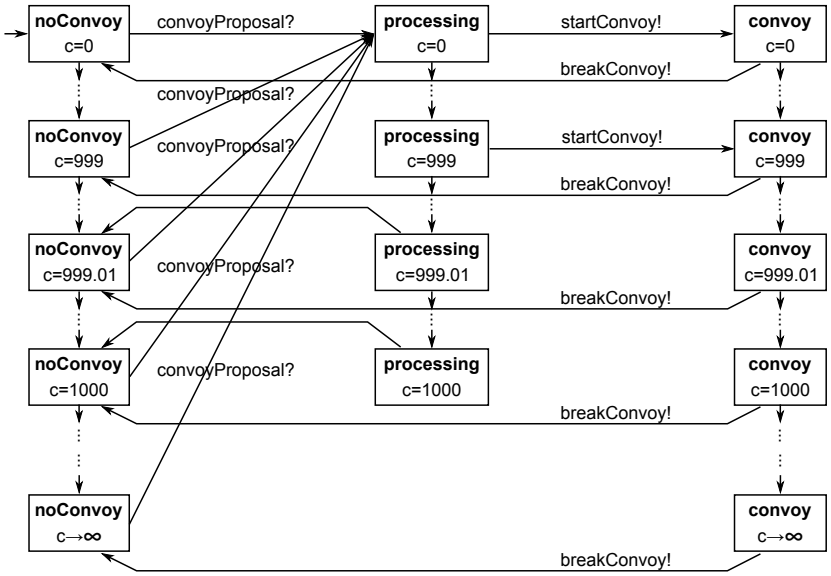

Figure 2.5: Infinite Timed Transition System of the Timed Automaton of the Convoy Coordination Behavior (Figure 2.3)

In this section we formally and informally described the syntax and semantics of timed automata. This also includes the concepts of clocks, clock constraints and clock valuations. We discovered that an infinite state space emerges from the fundamental semantics, because of the real-valued clocks. Still, real-valued clocks correspond most realistically to the notion of time. But obviously, a model with an infinite state space cannot be used for real-time reasoning or manipulation which comes along with the synthesis concept we propose in this paper (see section 1.2). Consequently, we need an interpretation of timed automata, which abstracts from the infinitely many and infinitely high clock values appearing during the run of a timed automaton.

2.1.2 Timing Abstractions

To abstract away from the infinite number of clock valuations for locations, which lead to an infinite state space for timed automata several techniques have been proposed. We give a short overview of the most often discussed techniques in the following, also stating advantages and disadvantages of each technique, and focus on the techniques relevant for the rest of this paper in the remainder of this section.

2.1.2.1 Region Automaton

The *region automaton* (or region graph) introduced by the inventors of timed (büchi) automata Alur and Dill in [AD90, ACD90, ACD93, AD94] was the first idea to handle the infinite state space problem for timed automata. They solve the infinite clock valuations problem by applying the following two techniques: (1) they define *equivalence classes* on the basis of the integral and fractional parts of clock values in order to solve the problem of infinitely many clock values between two clock values and (2) they introduce *global upper bounds* for each clock in order so solve the problem of infinitely high clock values. Once a clock value has exceeded its global upper bound, it is no longer important what the exact value is, but only that it is above its global upper bound.

A great advantage of the region automaton is that it preserves the continuous time model of the original timed automaton while obtaining a finite state space, which can be used for formal reasoning. Nonetheless, the state space still grows exponentially in the number of clocks and the highest constants appearing in the guards of the automaton. Altogether, this model is suitable for theoretical analysis of timed automata but not practical for real-life system specifications.

In the next section we describe a technique of abstraction, which has already successfully been used in the model checking domain and has also been applied in the synthesis approach proposed by Seibel in [Sei07].

2.1.2.2 Discretization of Time

Another technique widely discussed for the abstraction of the infinite state space is *discretizing the time domain* in timed automata as first proposed in [GPV94]. This technique restricts clock values to the set of natural numbers instead of using real values, which solves the problem of infinitely many clock values between two clock values by construction. The problem of having infinitely high clock values is again solved by using global upper bounds for each clock.

A discretization of time in timed automata legitimately leads to the question if this influences the semantics and therefore leads to incorrect verification results. Fortunately, it has been shown by Beyer and Noack that for the class of closed timed automata[1] the discrete time model is location equivalent to the continuous one, which means that the set of reachable locations in both models is equivalent [BN01, Bey01, Bey02].

A discrete time model is for example applied in the synthesis approach of [Sei07]. The number of discretized system states is, however, still of exponential

[1] Closed timed automata allow clock constraints φ of the form $\varphi ::= x \leq c \mid x \geq c \mid \varphi \wedge \varphi$ only, where x is a clock and $c \in \mathbb{N}$. It is said that the form of clock constraints is only restricted to simplify the discretization and enable efficient reachability analysis [BN01].

growth in the number of clocks and the highest constants appearing in the guards of the automaton. This could be solved by searching for the greatest common divisor amongst all constants appearing in clock constraints and by dividing all of the constants by this greatest common divisor. In many examples, though, like the one of the convoy coordination automaton where the constants are 999 and 1000, the greatest common divisor is 1 which would not lead to any improvement. The persistent presence of the state explosion problem is also supported by the fact that Beyer and Noack use the technique of abstraction in order to reduce the state space while performing verification algorithms [BN01, Bey01].

In addition to the state explosion problem, it is further argued that a discrete time model is suitable for synchronous but not for asynchronous systems. For *synchronous systems*, a single global clock can be assumed, which runs synchronously to all other clocks. This way, time can be discretized to time quanta, such that all events occur at an exact multiple of a time quantum. For *asynchronous systems* choosing this time quantum before running the system is difficult, as events can occur at an arbitrary point in time. This is especially the case, when regarding systems that interact with the environment such as mechatronic systems. Consequently, the time quantum would have to be chosen arbitrarily small in order to achieve the least loss of accuracy. Unfortunately, this would again blow up the state space, making formal reasoning no longer feasible. Furthermore, it is also cited a proof in this context, where it has been shown that the reachability problem cannot be solved correctly using a discrete time model [ACD93, p. 3], [CGP99, p. 265].

Summarizing this, a discrete time model is only applicable for analysis of synchronous systems, where abstraction can additionally be used to reduce the state space of the system. It is not applicable for real-time reasoning of asynchronous systems, where the complete interval of possible clock values has to be taken into account in order to prove system properties.

Accordingly, we describe another model of abstraction in the next section which preserves the continuous time domain, like the region automaton, but merges clock values efficiently to intervals in order to reduce the number of states and transitions.

2.1.2.3 Zone Automaton

To overcome the state explosion problem still present in the region automaton, Alur et al. propose the construction of a minimal region graph for timed büchi automata based on convex unions of regions, called clock zones, in [ACH+92]. Later this construction was called the *zone automaton* [AD96]. It was adapted to timed safety automata [Alu98, Alu99, BY03] and was successfully implemented in the real-time model checkers UPPAAL and KRONOS [BDL04, Yov97]. The zone

automaton is currently the most efficient continuous representation of a timed automaton, as it differentiates only between those time intervals in locations, which are necessary for a correct timing analysis.

After giving a short introduction on zone automata in the following, we formally and informally describe the syntax and semantics of zone automata in the remainder of this section. This includes the notion of clock zones and operations on clock zones which are necessary to construct a zone automaton. We further distinguish between the zone automaton defined by Alur in [Alu98, Alu99] and the zone automaton defined by Bengtsson and Yi in [BY03] and give advantages and disadvantages of both definitions.

Informally, a *clock zone* describes an infinite set of clock valuations defined as an interval for the values of each clock. This way, the problem of infinitely many clock values between two clock values is solved by defining the bounds for these clock values only. The zone $x \leq 1 \wedge y > 2$ for clocks x, y for example, describes the infinite number of clock values, where $0 \leq \nu(x) \leq 1$ and $2 < \nu(y) <= \infty$. A set of system states, called *zone location* in the following, can then be constructed by combining a timed automaton location with a zone. This way, the problem of infinitely high clock values is also solved, as a clock value does not have to be restricted by an upper bound, like the clock y in the zone given above, for example.

An example of a zone automaton corresponding to the timed automaton of the convoy coordination behavior (Figure 2.3) is depicted in figure 2.6. This zone automaton was constructed using the definition of Bengtsson and Yi given in [BY03]. As this definition also differentiates between delay and event transition (dashed and solid lines, respectively), it seems more intuitive to us as a preliminary example coming from the fundamental infinite dense-time semantics (see also Figure 2.5).

Again the initial state of the automaton is noConvoy with $c = 0$ as the initial zone. From there on, the delay transition points to the zone location of noConvoy with $true$ as the zone, which means that there is neither a lower nor an upper bound for this zone location. Observe at this point, that the infinite number of system states depicted in figure 2.5 is represented by two zone locations and one delay transition. From both of these zone locations, the automaton can switch to processing with $c = 0$, as the corresponding timed automaton transition resets the clock c. The time elapse in processing is again represented using a delay transition leading to processing with c<=1000, corresponding to the location invariant of processing. Note that this zone location represents all system states, where the automaton rests in processing and the value of the clock c is $0 \leq c \leq 1000$. Accordingly, for the clock values $0 \leq c \leq 999$ the automaton is able to switch to convoy, which is represented by the outgoing transition labeled with startConvoy!. As time cannot elapse during the execution of transitions, the target zone location

is convoy with c<=999 corresponding to the time guard of the timed automaton transition. For the clock values $999 < c \leq 1000$ the automaton is able to switch from processing to noConvoy again, which results in the zone location noConvoy with 999<c & c<=1000.

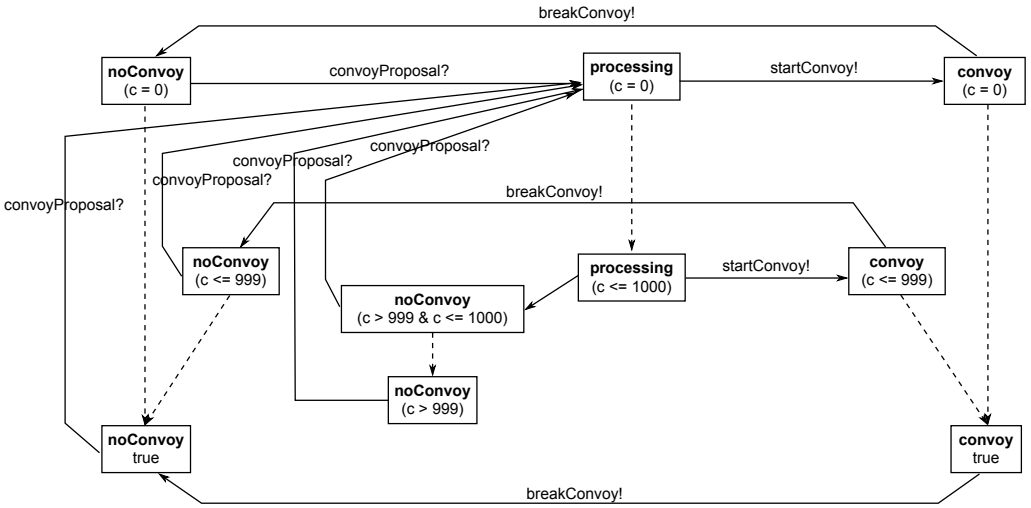

Figure 2.6: Zone Automaton of the Timed Automaton of the Convoy Coordination Behavior (Figure 2.3) according to [BY03]

We proceed with the formal definitions of clock zones, zone locations and operations on clock zones, which are necessary to construct a zone automaton from a given timed automaton.

Definition 2.1.10 (Clock Zone)
For a set C of clocks, the set $\Theta(C)$ of clock zones *ϑ is inductively defined by the grammar*

$$\vartheta ::= x \sim n \mid x - y \sim n \mid \vartheta \wedge \vartheta \mid true \mid false,$$

where $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$, $n \in \mathbb{N}$.

Although the syntax of a clock zone is equal to the conjunctive clock constraint, their semantics differ. For k clocks, both formalisms define a convex set of clock values in the k-dimensional euclidean space, possibly putting a lower and an upper bound on a clock or on the difference of two clocks. But while a clock constraint defines permitted clock valuations, the clock zone defines the solution set of a clock constraint, which is the maximal set of clock valuations for which the corresponding clock constraint evaluates to true. Speaking in context of the timed automaton, a clock constraint is used to describe the permitted clock valuations for a location or the execution of a transition. A clock zone, on the other

hand, is used to describe a set of clock valuations of a possible execution trace of a timed automaton. This is done in combination with an automaton location as defined in the following by the notion of *zone locations*.

Definition 2.1.11 (Zone Location)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$*, a location* $l \in L$ *and a clock zone* $\vartheta \subseteq \Theta(C)$ *we define a* zone location $s = (l, \vartheta) \in S_\Theta$ *with*

$$(l, \vartheta) \in L \times \Theta(C) = S_\Theta$$

As already seen in the example (Figure 2.6), a zone location is the combination of a timed automaton location and a clock zone defining a set of clock valuations for all clocks of the automaton.

In the following, we describe the *intersection*, the *time elapse* and the *clock reset* operation for clock zones. These operations are used to compute the successor of an initial zone location, which is also described below. Originally, the successor computation is used for reachability analysis on the zone automaton without creating the complete automaton [Alu99, BY03]. However, as reachability analysis in the worst case has to search the whole state space of a zone automaton, the reachability algorithm can also be applied to construct the complete zone automaton of an input timed automaton. We use this zone automaton in order to perform the refinement analysis after composition rules have been applied (see section 1.2 and section 3.3). We proceed with the definitions of the needed operations on clock zones starting with intersection.

Definition 2.1.12 (Intersection of Clock Zones)
For two clock zones ϑ_1 *and* ϑ_2*, the* intersection *of* ϑ_1 *and* ϑ_2 *denoted* $\vartheta_1 \wedge \vartheta_2$ *is defined with*

$$\vartheta_1 \wedge \vartheta_2 = \{\nu \mid \nu \in \vartheta_1 \wedge \nu \in \vartheta_2\}$$

Accordingly, the intersection of two clock zones is the set of clock valuations which are element of both clock zones. For a clock x and two clock zones $\vartheta_1 = x \geq 2 \wedge x < 6$ and $\vartheta_2 = x \geq 4 \wedge x \leq 8$ their intersection is $\vartheta_1 \wedge \vartheta_2 = x \geq 4 \wedge x < 6$. In the following we define the time elapse operation.

Definition 2.1.13 (Time Elapse of Clock Zones)
For a clock zone ϑ *the* time elapse *operation (also referred to with* up *operation) denoted* $\vartheta^{\Uparrow}$ *is defined with*

$$\vartheta^{\Uparrow} = \{\nu + \delta \mid \forall\, \nu \in \vartheta, \delta \in \mathbb{R}_+\}$$

The time elapse operation semantically describes the elapse of an arbitrary amount of time for a clock zone. Technically, this means that all upper bounds

are removed from a clock zone. Given two clocks x, y and a clock zone $\vartheta_1 = (x = 0 \wedge y = 0)$, which is actually the single clock valuation ν_0 of figure 2.4 with $\nu(x) = \nu(y) = 0$. The time elapse operation on ϑ_1 results in $\vartheta_2 = \vartheta^{\Uparrow} = (x - y = 0)$, which corresponds to the diagonal ray starting in ν_0 and going through ν_1.

In the following, we proceed with the definition of clock resets on zones, which is the last operation needed to construct a zone automaton from a given timed automaton.

Definition 2.1.14 (Clock Resets on Clock Zones)
For a set of clocks C, a clock zone $\vartheta \in \Theta(C)$ and a clock reset $Y \subseteq C$, the clock reset *operation on a clock zone denoted $\vartheta[Y := 0]$ is defined with*

$$\vartheta[Y := 0] = \{\nu[Y := 0] \mid \forall\, \nu \in \vartheta\}$$

The clock reset operation on clock zones, accordingly, performs a clock reset for the given clocks on all clock valuations of the given clock zone. As an example, assume the clock zone $\vartheta_2 = (x - y = 0)$ created in the example above. Performing a clock reset $r = \{x\}$ of the clock x on this zone results in the zone $\vartheta_3 = \vartheta_2[r := 0] = (x = 0)$. In the example of figure 2.4 this set of clock valuations is described by the ray starting in ν_0 and going along the x-axis.

With the above defined operations on clock zones, we are now able to define the *successor functions* which form the core of the zone automaton construction algorithm. As already mentioned above, Bengtsson and Yi [BY03] distinguish between a *delay successor* and an *event successor* of a zone location as defined below.

Definition 2.1.15 (Delay Successor of a Zone Location)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$, a location $l \in L$ and its invariant $I(l)$ and a corresponding zone location $s = (l, \vartheta) \in S_\Theta$, the delay successor *$succ_\delta(s)$ is computed by the function $succ_\delta : S_\Theta \rightarrow S_\Theta$ with*

$$succ_\delta((l, \vartheta)) = (l, \vartheta^{\Uparrow} \wedge I(l))$$

The delay successor function returns the maximal set of clock valuations for a location l and a given clock zone ϑ for which there exists a zone location $s = (l, \vartheta)$. This is done by letting time elapse on ϑ and intersecting it with the invariant $I(l)$. This way, it is guaranteed that the time elapse cannot exceed the invariant of the corresponding location.

This can be best observed at the zone location (processing,c=0) in the example of the zone automaton for the convoy timed automaton (Figure 2.6). The target zone location of the delay transition is computed by the delay successor function. While the corresponding timed automaton location is the same as in the source

zone location, the clock zone of the target zone location is $c \leq 1000$. The calculation of the successor zone location is exemplified in the following equation:

$$\begin{aligned} & succ_\delta((processing, c = 0)) \\ = \; & (processing, (c = 0)^{\Uparrow} \wedge I(l)) \\ = \; & (processing, true \wedge c \leq 1000) \\ = \; & (processing, c \leq 1000) \end{aligned}$$

Note that for one timed automaton location, there can at most exist one delay successor. Repeating the delay successor computation on an already computed successor can only result in the same zone location, as the function computes the maximal set of clock valuations by definition.

For calculating the successor zone locations of event transitions of the timed automaton, we use the *event successor* function as defined in the following.

Definition 2.1.16 (Event Successor of a Zone Location)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$, a transition $t = (l, e, g, r, l') \in T$ and a corresponding zone location $s = (l, \vartheta) \in S_\Theta$, the event successor $succ_\Sigma(s, t)$ *is computed by the function $succ_\Sigma : S_\Theta \times T \rightarrow S_\Theta$ with*

$$succ_\Sigma((l, \vartheta), (l, e, g, r, l')) = (l', (\vartheta \wedge g)[r := 0] \wedge I(l'))$$

The event successor function computes for a given zone location (l, ϑ) and a corresponding timed automaton transition $t = (l, e, g, r, l')$ the succeeding target zone location (l', ϑ'). This is done by (1) intersecting the clock zone ϑ with the guard g of the transition, (2) applying the clock resets r of t to the resulting zone and (3) intersecting the resulting zone with the invariant $I(l')$ of the target location. This way it is guaranteed, that (1) the transition is actually enabled in some of the clock valuations of the input zone ϑ and (2) that the invariant of the target location is not violated. If one of these conditions did not hold, the calculation of the event successor would lead to $(l', false)$, which means that from the input zone location the corresponding timed automaton transition is never enabled.

To give an example using the zone automaton of the convoy coordination behavior (Figure 2.6) again, we examine the zone location (processing,c<=1000) and the silent event transition (processing,τ,c>999,{},noConvoy) (see also Figure 2.3). The computation of the target zone is exemplified by the following equation, where $proc$ and $noCon$ abbreviate the automaton locations processing and

noConvoy:

$$
\begin{aligned}
& succ_{\Sigma}((proc, c \leq 1000), (proc, \tau, c > 999, \varnothing, noCon)) \\
&= (noCon, ((c \leq 1000) \wedge (c > 999))[\varnothing := 0] \wedge true) \\
&= (noCon, ((c \leq 1000 \wedge c > 999))[\varnothing := 0] \wedge true) \\
&= (noCon, (c \leq 1000 \wedge c > 999) \wedge true) \\
&= (noCon, c \leq 1000 \wedge c > 999).
\end{aligned}
$$

Using these definitions of the delay and the event successor of a zone location, we are now able to formalize the construction of the zone automaton by the following definition according to Bengtsson and Yi [BY03].

Definition 2.1.17 (Zone Automaton (Bengtsson & Yi))
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ *the corresponding* zone automaton $Z_A = (S_\Theta, s^0, \Sigma, C, T_\delta, T_\Sigma)$ *is defined with*

- $S_\Theta \subseteq L \times \Theta(C)$ *is the finite set of zone locations* $s = (l, \vartheta)$ *with* $l \in L$ *and* $\vartheta \in \Theta(C)$*, where* $s \in S_\Theta$ *iff* s *is reachable through* T_δ *or* T_Σ*,*
- $s^0 = (l^0, \vartheta_0) \in S_\Theta$ *is the* initial zone location *where* $\vartheta_0 = \{\nu_0\} \wedge I(l^0)$*,* $\forall c \in C : \nu_0(c) = 0$ *and* l^0 *is the initial location of* A*,*
- Σ *the set of* events *equal to the set of events of the corresponding timed automaton,*
- C *the set of* clock *equal to the set of clock of the corresponding timed automaton,*
- $T_\delta \subseteq S_\Theta \times S_\Theta$ *is the set of* delay transitions $t_\delta = (s, s')$*, where*
  - $s = (l, \vartheta)$ *is the source zone location,*
  - $s' = succ_\delta(s) = (l, \vartheta')$ *is the target zone location,*

  and $t_\delta \in T_\delta \Leftrightarrow$
  - s *is reachable through* T_δ *or* T_Σ*, and*
  - $\vartheta' \neq \vartheta$*, and*
  - $\vartheta' \neq false$*,*
- $T_\Sigma \subseteq S_\Theta \times \Sigma \times S_\Theta$ *is the infinite set of* event transitions $t_\Sigma = (s, e, s')$*, where*
  - $s = (l, \vartheta)$ *is the source zone location,*

- $e \in \Sigma$ *is the corresponding event,*
- $s' = succ_\Sigma(s,t) = (l', \vartheta')$ *with* $t \in T$ *is the target zone location,*

and $t_\Sigma \in T_\Sigma \Leftrightarrow$

- s *is reachable through* T_δ *or* T_Σ*, and*
- $\vartheta' \neq false$*.*

The formal definition of the zone automaton is obviously very similar to the one of the timed automaton semantics (Definition 2.1.9), as the zone automaton is an abstraction of the timed automaton semantics. Locations in the zone automaton are zone locations (Definition 2.1.11) and the initial zone location s^0 is the initial timed automaton location l^0 combined with the initial clock zone ϑ_0. Note that ϑ_0 already takes the invariant $I(l^0)$ into account to guarantee that the initial zone location is in fact reachable. Furthermore, the definition distinguishes between delay and event transitions as already described above. Observe that the source zone location s is defined to be reachable through the transition relations T_δ or T_Σ and that the clock zone ϑ' of the target zone location s' may not be $false$. This means, zone locations which are in fact not reachable due to timing constraints are excluded from the zone automaton. An example of a zone automaton according to Bengtsson and Yi is described in the introduction of this section (Figure 2.6).

The zone automaton, as defined by Bengtsson and Yi, includes delay transitions to explicitly describe the flow of time. In fact, for a simple reachability analysis where only the potential occurrences of events are of concern, delay transitions do not have to be considered explicitly. Alur proposes a different notion of the successor function in [Alu98, Alu99], in the following called *entrance successor*, which takes the time elapse of a zone location only implicitly into account. This way, the zone automaton can be reduced both in the size of zone locations and in the size of transitions.

Before we formally define the entrance successor and the zone automaton according to Alur, we informally describe the resulting zone automaton using the example of the convoy coordination behavior again (Figure 2.7).

The initial zone location is again (noConvoy,(c=0)). But in contrast to the zone automaton depicted in figure 2.6, there is no explicit delay transition in this automaton. The amount of time, where the automaton may rest in location noConvoy, is simply omitted. Instead, the computation of the outgoing (event) transitions takes this time interval, given as the location invariant, into account. For each outgoing transition it is calculated if there is some clock valuation reachable from the source zone location, such that the corresponding transition is enabled. The time guard of the transition is also regarded in this calculation. As this is true for the convoyProposal? transition, the clock reset is applied, which results in the

clock zone (c=0) for the location processing. From there on, following the silent transition back to noConvoy again, the clock zone of that zone location exemplifies the notion *entrance successor* perfectly. When entering noConvoy through the transition (processing,τ,c>999,{},noConvoy), the clock value of c must be $999 < c \leq 1000$, regarding the transition's time guard (c>999) and the location invariant of processing (c<=1000). This is exactly reflected in the clock zone of the zone location (noConvoy,(c>999 & c<=1000)). The rest of the automaton is constructed analogously.

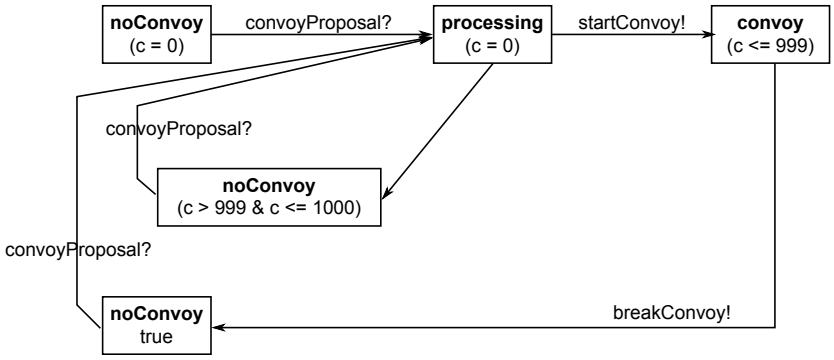

Figure 2.7: Zone Automaton of the Timed Automaton of the Convoy Coordination Behavior (Figure 2.3) According to [Alu98, Alu99]

In the remainder of this section we give formal definitions of both the *entrance successor* and of the zone automaton in accordance with Alur [Alu98, Alu99].

Definition 2.1.18 (Entrance Successor of a Zone Location)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$*, a transition* $t = (l, e, g, r, l') \in T$ *and a corresponding zone location* $s = (l, \vartheta) \in S_\Theta$*, the* entrance successor $succ_\alpha(s, t)$ *is computed by the function* $succ_\alpha : S_\Theta \times T \to S_\Theta$ *defined with*[2]

$$succ_\alpha((l, \vartheta), (l, e, g, r, l')) = (l', (\vartheta^{\Uparrow} \wedge I(l) \wedge g)[r := 0] \wedge I(l')).$$

For a given zone location $s = (l, \vartheta)$ and a corresponding outgoing timed automaton transition $t = (l, e, g, r, l')$, the entrance successor function computes the succeeding zone location of s. Semantically, this is the zone location describing the target timed automaton location with the clock zone right after the execution

[2]Note that in [Alu98, Alu99] the successor function is actually defined with $succ((l, \vartheta)) = (l', ((\vartheta \wedge I(l))^{\Uparrow} \wedge I(l) \wedge g)[r := 0])$ which does not take the invariant of the target location into account. Obviously, this would lead to zone locations which are in fact not reachable, as the target location invariant might restrict the execution of the corresponding transition.

of the transition t. In other words, the target clock zone is the set of clock valuations, where each of these clock valuations is a possible valuation, when the target location is entered by the transition t.

The timed automaton location of the target zone location is exactly the target location of t which is l'. The clock zone of the target zone location is computed in the following steps: (1) a time elapse is applied on ϑ, (2) the resulting zone is intersected with the invariant $I(l)$, (3) the result of this is intersected with the guard of the transition t, (4) the clock resets of t are applied and (5) the resulting clock zone is intersected with the invariant of the target location $I(l')$. After (1) and (2), the resulting zone describes the maximal set of clock valuations being valid for the location l, starting in the zone ϑ. The intersection with the guard g of the transition t in step (3) guarantees that the transition is actually enabled in at least one of the clock valuations of $(\vartheta^{\Uparrow} \wedge I(l))$. Step (4) applies the clock resets and step (5) guarantees, that the resulting clock zone does not violate the invariant of the target location $I(l')$. To illustrate the computation of the entrance successor, we take the zone location (processing,c=0) and the (silent) timed automaton transition (processing,τ,c>999,{},noConvoy) as the input (Figure 2.7 and Figure 2.3). The computation is exemplified in the following equation, where $proc$ and $noCon$ again abbreviate the locations processing and noConvoy:

$$
\begin{aligned}
& succ_{\alpha}((proc, c = 0), (proc, \tau, c > 999, \varnothing, noCon)) \\
&= (noCon, ((c = 0)^{\Uparrow} \wedge I(proc) \wedge (c > 999))[\varnothing := 0] \wedge I(noCon)) \\
&= (noCon, (true \wedge c \leq 1000 \wedge c > 999)[\varnothing := 0] \wedge true) \\
&= (noCon, (c \leq 1000 \wedge c > 999)[\varnothing := 0] \wedge true) \\
&= (noCon, c \leq 1000 \wedge c > 999 \wedge true) \\
&= (noCon, c \leq 1000 \wedge c > 999)
\end{aligned}
$$

On the basis of this definition of the entrance successor, we formalize the definition of the zone automaton according to Alur in the following.

Definition 2.1.19 (Zone Automaton (Alur))
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ *the corresponding* zone automaton $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ *is defined with*

- $S_\Theta \subseteq L \times \Theta(C)$ *is the finite set of zone locations* $s = (l, \vartheta)$ *with* $l \in L$ *and* $\vartheta \in \Theta(C)$*, where* $s \in S_\Theta$ *iff* s *is reachable through* T_Θ*,*
- $s^0 = (l^0, \vartheta_0) \in S_\Theta$ *is the* initial zone location *where* $\vartheta_0 = \{\nu_0\} \wedge I(l^0)$*,* $\forall c \in C : \nu_0(c) = 0$ *and* l^0 *is the initial location of* A*,*
- Σ *the set of* events *equal to the set of events of the corresponding timed automaton,*

- C *the set of* clock *equal to the set of clock of the corresponding timed automaton,*
- $T_\Theta \subseteq S_\Theta \times \Sigma \times S_\Theta$ *is the infinite set of* event transitions $t_\Theta = (s, e, s')$ *where*
  - $s = (l, \vartheta)$ *is the source zone location,*
  - $e \in \Sigma$ *is the corresponding event,*
  - $s' = succ_\alpha(s, t) = (l', \vartheta')$ *with* $t \in T$ *is the target zone location,*

  and $t_\alpha \in T_\alpha \Leftrightarrow$
  - s *is reachable through* T_Θ*, and*
  - $\vartheta' \neq false$

The definition of the zone automaton on the basis of Alur's successor function is in most parts equal to the definition of Bengtsson and Yi. The main difference is the transition relation, as there is only one type of transition in this automaton. This transition relation is based on the entrance successor. Note that also in this definition every source zone location (l, ϑ) of an event transition must be reachable in the automaton and the target clock zone may not be equal to false, as this would again mean, that the target location is not reachable starting in the source zone location of that transition.

As a consequence of this transition relation, there is at most one successor of a zone location per outgoing transition of the corresponding timed automaton location. This leads to a much smaller state space compared to the zone automaton of Bengtsson and Yi, which defines additionally a delay successor for each reachable zone location (see Definition 2.1.17). Consequently, the zone automaton based on Alur's entrance successor function can be estimated to be only half of the size of the zone automaton of Bengtsson and Yi.

Observe that the algorithm to create a zone automaton further applies a procedure called *normalization*. This procedure ensures that the algorithm which creates the successors of a zone location actually terminates for all given timed automata. Although we employ the procedure in the creation of the zone automata, it is not of direct relevance for this paper, as it is not further used by the proposed synthesis algorithm. Therefore, we omit the description of the normalization procedure and refer to [BY03, pp. 95–98] instead.

In this section we described the foundations for the core of the synthesis procedure which is the timed automaton formalism. In the first part, we described the most discussed variants, timed büchi and timed safety automata, and justified our choice of timed safety automata. After that we gave the necessary formal definitions for later reasoning on timed automata. As the fundamental semantics of

timed automata leads to an infinite state space, we described possible abstractions in the second part of this section. We made our choice for a continuous dense-time abstraction, namely the zone automaton. We furthermore justified this choice and described the two variants discussed in literature formally and informally. In the next section we introduce the requirement specification language *timed computation tree logic (TCTL)* as it is employed by MECHATRONIC UML for the formal verification of the behavioral models by means of model checking.

2.1.3 Timed Computation Tree Logic (TCTL)

As mentioned in the introduction of this paper, our synthesis approach preserves verification results on the given behavioral models of the pattern roles. More precisely this means that the requirements for each role are not violated by the appliance of composition rules. In this section we introduce the requirements specification language applied for model checking in MECHATRONIC UML. Referring to different parts of this language, we define in the next chapter which parts of the language describe the relevant behavior of the pattern roles. In accordance with that, we also describe which type of verification results have to be preserved by the synthesis procedure (see section 3.3).

Model Checking, as introduced by Clarke and Emerson in [CE82] and by Queille and Sifakis in [QS82], is the procedure of verifying formalized requirements on a formalized system's behavioral specification. In the preceding section we introduced timed automata as a formal model for the behavioral specification of real-time systems. In this section we introduce a formal language for real-time system requirements which is used to verify requirements on a given timed automaton. For a general overview on model checking we refer to [CGP99, GV08].

In MECHATRONIC UML, model checking is performed by transforming the corresponding models to timed automata and by using these timed automata as input for the model checker UPPAAL [BGHS04]. Accordingly, the requirements specification language of UPPAAL is also applied to specify requirements for the corresponding models in MECHATRONIC UML. This formal language is a subset of the *timed computation tree logic (TCTL)*, the timed version of the *computation tree logic (CTL)*. The TCTL variant of UPPAAL is described in the following and is referred to by *U-TCTL*. For descriptions of the more general versions TCTL and CTL we refer to [ACD90, ACD93] (TCTL) and [CGP99, pp. 30–33] (CTL).

Before we formally define the syntax and semantics of U-TCTL, we give a short introduction and give some examples of the language using the front role timed automaton of the convoy example (Figure 2.3).

Informally, a U-TCTL formula specifies a property for a timed automaton by referring to all reachable timed system states of its corresponding timed transition system (cf. Figure 2.5 and Definition 2.1.9, Timed Automaton Semantics). Dur-

ing the execution of a timed automaton, all these reachable states can be represented by an infinite tree, called the *computation tree*, where the root is the initial location in which all clocks' values are zero (Figure 2.8). An example of a property expressable in U-TCTL is that there exists a path in the computation tree of the automaton where a state with location convoy and the value of clock c being equal to 500 is reachable. In the example tree, this is the second path on the left hand side.

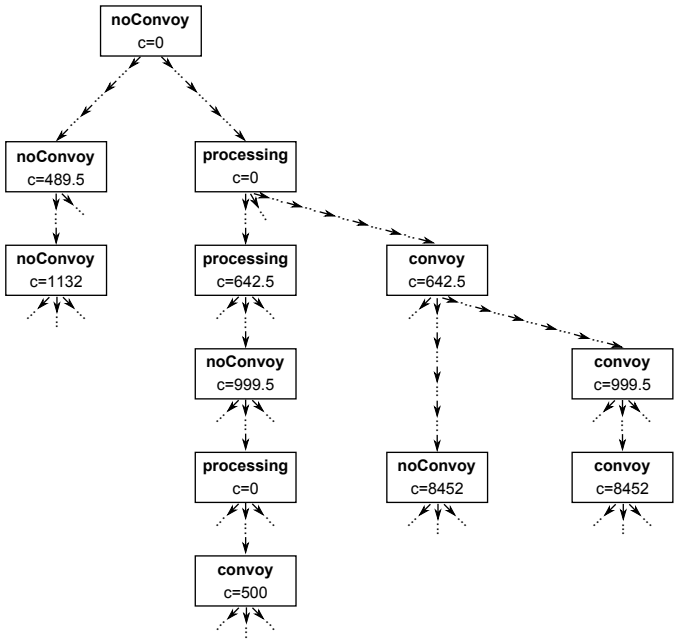

Figure 2.8: Illustration of the Computation Tree of the Timed Automaton of the Convoy Protocol (Figure 2.3)

Syntactically, a U-TCTL formula consists of a path quantifier, a temporal operator and a state property. The *path quantifier*, either $\forall$ or $\exists$, specifies if the property has to hold on all paths or only on some (but at least one) path of the computation tree. The *temporal operator*, either $\Box$ (globally) or $\Diamond$ (eventually), specifies if the property has to hold in all states of the path or if there only has to be a reachable state, where the property holds. Finally, the *state property*, consisting of location labels and clock constraints (see Definition 2.1.1, General Clock Constraint) connected by the Boolean operators $\neg, \wedge, \vee$ and $\Rightarrow$, specify the property which has to hold in the specified states of the computation tree. Using this syntax, the above state property can be formalized to λ_1 specified as:

$$\lambda_1 = \exists\Diamond\ (convoy \wedge c = 500).$$

Note that, unlike to the more general logics TCTL and CTL, nesting of path quantifiers and temporal operators is not allowed in U-TCTL. Accordingly, the formula $\forall\Box\,(processing \Rightarrow \forall\Diamond(convoy \vee noConvoy))$, which states that for all states where processing holds, there is a later state where convoy or noConvoy holds, is not a valid U-TCTL formula. However, a very important property for real-time systems can only be expressed this way, which is the *bounded response* [KC05]. In general, it states that whenever a certain property holds, another property holds within a specified time interval.

To overcome this problem, U-TCTL introduces the *leads-to operator*, denoted $\rightsquigarrow$. This operator is used to specify that the fulfillment of one state property always leads to the fulfillment of another state property later. When the state property which has to hold later in time includes a clock constraint, this can be used to express bounded response properties. Accordingly, it can be specified for the convoy example that processing always leads to convoy or noConvoy with c<=1000 through the following U-TCTL λ_2:

$$\lambda_2 = processing \rightsquigarrow (convoy \vee noConvoy) \wedge c \leq 1000.$$

As we informally introduced the syntax and semantics of U-TCTL formulas above, we proceed with the formal definition of the syntax and semantics. As the core of a U-TCTL formula is always a state property, we begin with the syntax of state properties.

Definition 2.1.20 (State Property)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ *a* state property $\upsilon \in \Upsilon(L, C)$ *is inductively defined by the grammar*

$$\upsilon ::= l \mid \varphi \mid \neg\upsilon \mid \upsilon \wedge \upsilon \mid \upsilon \vee \upsilon \mid \upsilon \Rightarrow \upsilon$$

where $l \in L$ *and* $\varphi \in \Phi(C)$.

The definition of a state property exactly reflects the above stated concepts. Each terminal symbol is either a state or a clock constraint (see Definition 2.1.1, General Clock Constraint). All terminal symbols can be arbitrarily connected by the Boolean operators $\neg, \wedge, \vee$ and $\Rightarrow$. A valid state property example is $(\neg processing \wedge c > 1000)$, which states that the examined state must be different from processing and the value of clock c must be greater than 1000.

Note that this grammar also allows to construct state properties by connecting two locations with a Boolean meet. This type of properties can never be satisfied if evaluated on a single automaton. However, when running several automata in parallel and the different locations are of different automata, these types of formulas can specify important properties for the system by referring to combinations of automaton locations. State properties are embedded in U-TCTL formulas, as defined in the following.

Definition 2.1.21 (U-TCTL Formula)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ a U-TCTL formula $\lambda \in \Lambda(L, C)$ is defined by the grammar

$$\lambda ::= \forall\Box\, \upsilon \mid \forall\Diamond\, \upsilon \mid \exists\Box\, \upsilon \mid \exists\Diamond\, \upsilon \mid \upsilon_1 \rightsquigarrow \upsilon_2$$

where $\upsilon, \upsilon_1, \upsilon_2 \in \Upsilon(L, C)$.

According to this definition and also in accordance with the above given descriptions, a U-TCTL formula consists of a path quantifier ($\forall$ of $\exists$) and a temporal operator ($\Box$ (globally) or $\Diamond$ (eventually)) and a state property υ. Furthermore, the special operator $\rightsquigarrow$ (always leads to) can be used to specify that one state property υ_1 always leads to the fulfillment of another state property υ_2. Examples of U-TCTL formulas are given above with λ_1 and λ_2.

We proceed with the formal definition of the semantics of a U-TCTL formula, which defines in which cases a timed automaton satisfies a certain U-TCTL formula. To evaluate a U-TCTL formula, it is necessary to examine all reachable timed system states as defined by the timed automaton semantics (Definition 2.1.9). Consequently, the following definitions are based on the corresponding timed transition system of a timed automaton. We begin with the evaluation of a state property, which defines for a given timed system state if the property holds or not.

Definition 2.1.22 (State Property Evaluation)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$, its corresponding timed transition system $S_A = (Q_A, q^0, \Sigma, T_\delta, T_\Sigma)$, a timed system state $s = (l, \nu) \in Q_A$ with $\nu \in \Psi(C)$ and a state property $\upsilon \in \Upsilon(L, C)$, the state property evaluation $\upsilon((l_1, \nu_1))$ *is a function $\upsilon : Q_A \to \{true, false\}$ defined with*

$$\upsilon((l,\nu)) = \begin{cases} l = l_\upsilon, & \text{iff } \upsilon \text{ is a location } l_\upsilon, \\ \varphi_\upsilon(\nu), & \text{iff } \upsilon \text{ is a clock constraint } \varphi_\upsilon, \\ \neg\upsilon_1((l,\nu)), & \text{iff } \upsilon \text{ is of the form } \neg\upsilon_1, \\ \upsilon_1((l,\nu)) \wedge \upsilon_2((l,\nu)), & \text{iff } \upsilon \text{ is of the form } \upsilon_1 \wedge \upsilon_2, \\ \upsilon_1((l,\nu)) \vee \upsilon_2((l,\nu)), & \text{iff } \upsilon \text{ is of the form } \upsilon_1 \vee \upsilon_2, \\ \upsilon_1((l,\nu)) \Rightarrow \upsilon_2((l,\nu)), & \text{iff } \upsilon \text{ is of the form } \upsilon_1 \Rightarrow \upsilon_2, \end{cases}$$

where $l \in L$ and $\varphi \in \Phi(C)$.

The central part of the evaluation of a state property, is the evaluation of the contained locations and clock constraints. The evaluation of a contained location is true, if the location l of the timed system state is equal to the state l_υ of the formula. The evaluation of a clock constraint corresponds to the clock constraint

evaluation of a timed automaton (Definition 2.1.8). Each evaluation of a state property which is constructed using the Boolean operators $\neg, \wedge, \vee$ or $\Rightarrow$ is evaluated, by evaluating the corresponding sub formulas and applying the conventional rules of Boolean logic.

The evaluation of a state property is applied in the evaluation of a U-TCTL formula as defined in the following.

Definition 2.1.23 (U-TCTL Formula Evaluation)
Given a timed automaton $A = (L, l^0, \Sigma, C, I, T)$, its corresponding timed transition system $S_A = (Q_A, q^0, \Sigma, T_\delta, T_\Sigma)$ and a U-TCTL formula $\lambda \in \Lambda(L, C)$. Furthermore, let (π, s) denote an (infinite) sequence of timed system states reachable from s through T_δ and T_Σ. The timed automaton A satisfies the formula λ, denoted $A \models \lambda$, by distinguishing between the following cases:

- $A \models \forall\Box\ \upsilon \Leftrightarrow \forall\ (\pi, s_0) \in S_A, \forall\ s \in \pi : \upsilon(s) = true$,
- $A \models \forall\Diamond\ \upsilon \Leftrightarrow \forall\ (\pi, s_0) \in S_A, \exists\ s \in \pi : \upsilon(s) = true$,
- $A \models \exists\Box\ \upsilon \Leftrightarrow \exists\ (\pi, s_0) \in S_A, \forall\ s \in \pi : \upsilon(s) = true$,
- $A \models \exists\Diamond\ \upsilon \Leftrightarrow \exists\ (\pi, s_0) \in S_A, \exists\ s \in \pi : \upsilon(s) = true$,
- $A \models \upsilon_1 \rightsquigarrow \upsilon_2 \Leftrightarrow \forall\ (\pi, s_0) \in S_A, \forall\ s \in \pi : (\upsilon_1(s) = true) \Rightarrow \forall\ (\pi', s) \in S_A, \exists\ s' \in (\pi', s) : \upsilon_2(s') = true$.

For the evaluation of a U-TCTL formula, we distinguish between the different forms of a formula, as defined in its syntax (Definition 2.1.21, U-TCTL Formula). For all formulas we have to examine all paths (π, s_0) of the timed transition system S_A, where (π, s_0) is a path starting in s_0 following transitions of T_δ or T_Σ. For the formulas starting with $\forall\Box$, all states of all paths have to satisfy the corresponding state property. For the formulas starting with $\forall\Diamond$, only one state of all paths has to satisfy this property. Analogously, the formulas starting with $\exists$ are evaluated, with the difference, that only one path (π, s_0) has to exist in S_A to satisfy the formula. The evaluation of a formula $\upsilon_1 \rightsquigarrow \upsilon_2$ is evaluated in two steps: (1) it is searched for all paths (π, s_0) where there exists a state s for which υ_1 holds and (2) it is verified for these paths, that on all paths starting in this state, there is a state s', where υ_2 holds.

In the above given descriptions, we formally and informally defined the syntax and semantics of U-TCTL, the TCTL variant applied by UPPAAL. The synthesis approach proposed in this paper preserves the relevant verification results by means of verified U-TCTL formulas. It is not possible, however, to preserve all properties, which can be specified through U-TCTL formulas (see section 3.3). Consequently, we further distinguish between several sub logics of U-TCTL as

defined in the remainder of this section. We begin by distinguishing between formulas which only refer to all paths, defined by U-ATCTL, or which only refer to the existence of paths, defined by U-ETCTL, for a timed automaton.

Definition 2.1.24 (U-ATCTL Formula)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ *a U-ATCTL formula* $\lambda \in \Lambda_A(L, C)$ *is defined by the grammar*

$$\lambda ::= \forall\Box\, \upsilon \mid \forall\Diamond\, \upsilon \mid \upsilon_1 \rightsquigarrow \upsilon_2$$

where $\upsilon, \upsilon_1, \upsilon_2 \in \Upsilon(L, C)$.

Definition 2.1.25 (U-ETCTL Formula)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ *a U-ETCTL formula* $\lambda \in \Lambda_E(L, C)$ *is defined by the grammar*

$$\lambda ::= \exists\Box\, \upsilon \mid \exists\Diamond\, \upsilon$$

where $\upsilon \in \Upsilon(L, C)$.

The definitions of U-ATCTL and U-ETCTL formulas split the definition of U-TCTL formulas into formulas only using the $\exists$ path quantifier and formulas only using the $\forall$ path quantifier. With U-ETCTL formulas only those properties can be specified which only have to hold on at least one path of the automaton. With U-ATCTL formulas only those properties can be specified which have to hold on all paths of the automaton. This includes the leads-to operator $\rightsquigarrow$ corresponding to its semantic given in the definition of the U-TCTL Formula Evaluation (Definition 2.1.23). The U-TCTL formula $\lambda_1 = \exists\Diamond\, (convoy \wedge c = 500)$, for example, is a valid U-ETCTL formula, but not a valid U-ATCTL formula. The formula $\lambda_2 = processing \rightsquigarrow (convoy \vee noConvoy) \wedge c \leq 1000$, on the other hand, is a valid U-ATCTL formula, but not a valid U-ETCTL formula.

Furthermore, note that the set of possible U-TCTL formulas $\Lambda(L, C)$ can be constructed by the union of sets of U-ATCTL formulas and U-ETCTL formulas, i.e. $\Lambda(L, C) = \Lambda_A(L, C) \cup \Lambda_E(L, C)$. The reason for this is that nesting of path quantifiers is not included in the grammar of U-TCTL. For the same reason, every U-TCTL formula is either a U-ATCTL or a U-ETCTL formula but never both, i.e. $\Lambda_A(L, C) \cap \Lambda_E(L, C) = \varnothing$.

In the following, we define U-ECTL as a sub logic of U-TCTL, which does not allow to use clock constraints in state properties. This is used to specify properties of existing paths in timed automata while not taking any timing constraints into account. We begin with the definition of an *untimed state property*.

Definition 2.1.26 (Untimed State Property)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ *an* untimed state property $\upsilon \in \Upsilon_U(L, C)$ *is inductively defined by the grammar*

$$\upsilon ::= l \mid \neg\upsilon \mid \upsilon \wedge \upsilon \mid \upsilon \vee \upsilon \mid \upsilon \Rightarrow \upsilon$$

where $l \in L$.

An *untimed state property* is simply a state property without any clock constraint. Consequently, we can only specify properties referring to the locations of one or more automata (cf. Definition 2.1.20, State Property). Accordingly, $(convoy \wedge c = 500)$ is not a valid untimed state property, but can be transformed into a valid one by removing the clock constraint $c = 500$.

Note that the set of untimed state properties is a proper subset of the set of state properties, as the clock constraint φ is a terminal symbol in the grammar of state properties, i.e. $\Upsilon_U(L, C) \subset \Upsilon(L, C)$. The untimed state property is used in the following definition of U-ECTL formulas.

Definition 2.1.27 (U-ECTL Formula)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$ *a U-ECTL formula* $\lambda \in \Lambda_{UE}(L, C)$ *is defined by the grammar*

$$\lambda ::= \exists\Box\ \upsilon \mid \exists\Diamond\ \upsilon$$

where $\upsilon \in \Upsilon_U(L, C)$.

A U-ECTL formula is defined to be a formula only referring to untimed state properties $\Upsilon_U(L, C)$. Consequently, only those properties can be specified which do not refer to any clock values of the clocks of the automaton. Accordingly, the U-TCTL formula $\lambda_1 = \exists\Diamond\ (convoy \wedge c = 500)$ is not a valid U-ECTL formula. The weaker version $\exists\Diamond\ convoy$, where the clock constraint is removed, is a valid E-TCTL formula, which states that the location convoy is reachable somewhere on a paths of the timed automaton.

Observe that the set of U-ECTL formulas is a proper subset of the set of U-ETCTL formulas, i.e. $\Lambda_{UE}(L, C) \subset \Lambda_E(L, C)$. This is due to the fact that the embedded untimed state properties are a proper subset of the set of state properties used in U-ETCTL formulas.

In this section we defined the timed automaton formalism which is the foundation of the behavioral models in MECHATRONIC UML. We further defined an efficient method to abstract from the infinite state space of timed automata, namely the zone automaton, which preserves the continuous time property for formal analysis. Finally, we described the specification language TCTL as it is employed by the real-time model checker UPPAAL to verify system requirements on a given timed automaton model. As UPPAAL is employed for real-time model checking in MECHATRONIC UML, TCTL is also the specification language used in MECHATRONIC UML. In the next section, we introduce the modeling language MECHATRONIC UML by describing those parts of the language which are relevant for this paper.

2.2 Modeling Reconfigurating Real-time Systems in MECHATRONIC UML

In this section we describe the modeling concepts of MECHATRONIC UML which are relevant for this paper. MECHATRONIC UML forms an appropriate approach for the model-driven development and verification of mechatronic real-time systems. This includes the modeling of behavior between different entities using *real-time coordination patterns*, as well as modeling the structure of the developed system using component diagrams based on the *Unified Modeling Language (UML)* [OMG09]. We furthermore focus on those parts of the software development process where continuous entities also have to be taken into account, for example feedback controllers developed by control engineers. This has to be regarded, as mechatronic systems are part of multiple domains and therefore the interleaving of different domains in the development process is unavoidable.

We begin by describing the models of MECHATRONIC UML which define the behavior between different entities which are *real-time statecharts* used in *real-time coordination patterns* (section 2.2.1). After that, we proceed with the models which define the system structure by means of discrete and continuous components (section 2.2.2). At last, reconfigurations of the system structure are defined through *reconfiguration charts*, an extension of real-time statecharts, which are described in the last part of this section (section 2.2.3).

2.2.1 Real-time Coordination Behavior

In MECHATRONIC UML, the real-time communication behavior between different entities of the system is defined by *real-time statecharts* [GB03], which extend UML state machines by the notion of time. The semantics of real-time statecharts, consequently, is based on timed automata (see section 2.1), which means that every real-time statechart can be transformed into a semantically equivalent timed automaton [BGHS04].

For the specification of real-time statecharts each participating entity is referred to by a *role*, such that one real-time statechart describes the behavior of exactly one role (cf. section 1.2). This way, no direct dependencies to the actual components of the system are created.

The real-time statecharts for the front role and the rear role behavior of two RailCabs potentially driving in convoy are depicted in figure 2.9 and figure 2.10. The defined behavior corresponds to the behavior of the timed automaton depicted in figure 2.3 (see section 2.1.1). The rear role real-time statechart (Figure 2.10) has two superstates noConvoy and convoy, where the noConvoy superstate in turn has two supstates default and wait.

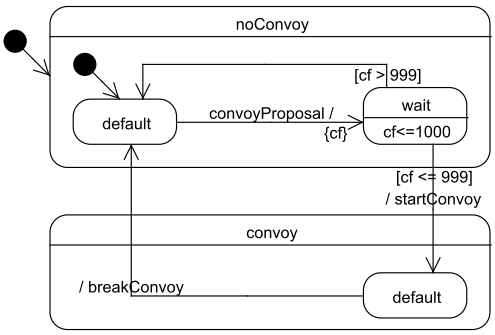

Figure 2.9: Real-time Statechart for the Front Role Behavior

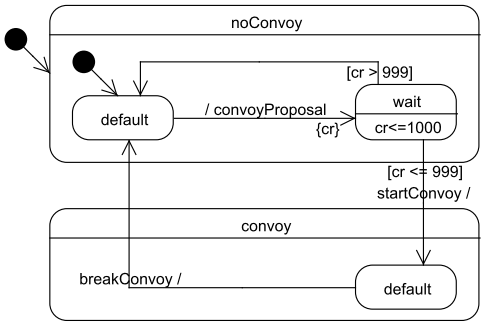

Figure 2.10: Real-time Statechart for the Rear Role Behavior

The main difference between the semantics of a timed automaton and a real-time statechart is that time can pass during the execution of a transition of a real-time statechart. For this reason, each transition of a real-time statechart may have a *triggering event* and a *raised event*. Syntactically, those are separated by a slash ("/"), where the triggering event is annotated on the left-hand side of the slash. In the real-time statechart of the rear role behavior, the transition leading from default to wait raises a convoyProposal and resets the clock cr.

In real-time statecharts, the communication between statecharts can also be of asynchronous character, which means that the sending of an event does not block any further action of the sender until the event is received. The convoyProposal event sent by the rear role is accordingly received by the front role, possibly at a later point in time, depending on the underlying communication channel.

The usage of time guards and clock resets in real-time statecharts corresponds to the usage of those in timed automata and is therefore not further described at this point.

To complete the example, we further describe the communication behavior necessary for the registration of a RailCab at a base station (cf. section 1.1). The coordination behavior is divided into the role registrar representing a base station and the role registree representing a RailCab (Figure 2.11 and Figure 2.12).

The coordination is initiated by the registree role by sending a register event to the registrar role. After that, both roles are in the state default of the superstate registered. Resting in this state is only allowed for at most 2000 ms. In this interval, the registree role sends a requestUpdate to the registrar role, which answers with a performUpdate within 500 ms. Finally, the registree role is able to unregister from the registrar, by sending an unregister event to the registrar, which is provided in any registered state of its real-time statechart.

Real-time statecharts are integrated into the system specification of the system through *real-time coordination patterns*. A real-time coordination pattern is

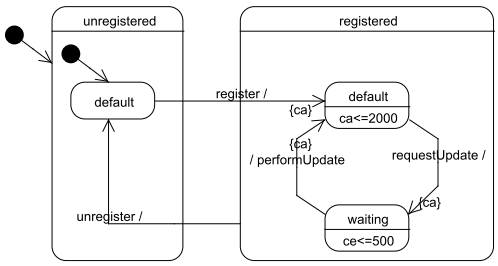

Figure 2.11: Real-time Statechart for the Registrar Role Behavior

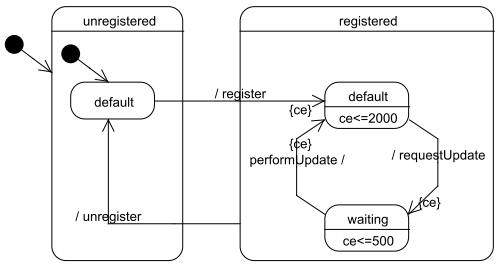

Figure 2.12: Real-time Statechart for the Registree Role Behavior

a structural model, which defines a reusable communication pattern between a number of *roles*. The behavior of each role is defined by a real-time statechart. The real-time coordination patterns for the convoy coordination and for the registration coordination are depicted in figure 2.13 and figure 2.14 respectively.

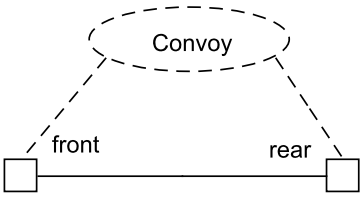

Figure 2.13: Convoy Coordination Pattern

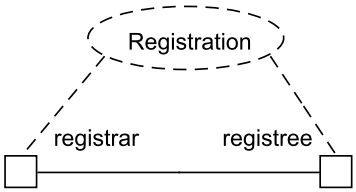

Figure 2.14: Registration Coordination Pattern

In addition to that, also the *quality of service (QoS)* characteristics of the underlying communication channel is specified in the communication pattern, such as throughput or maximal delay. In our example, the maximal delay of a message is specified with 5 ms. Additionally, the QoS channel characteristic reliable is set to true.

During the specification of a coordination pattern, real-time model checking (see section 2.1.3) can be applied in order to verify the desired requirements for a given coordination pattern. It has to be taken into consideration, however, that due to the current compositional approach of MECHATRONIC UML (see section 1.2) only the universally quantified properties (cf. Definition 2.1.24, U-ATCTL Formula) will be preserved, if a role is refined in a component specification. Once a the specification of a real-time coordination pattern is complete, it is stored to a *pattern library* for later reuse.

2.2.2 Component Specification

The structural part of the (software) system specification is defined by component diagrams derived from UML. For MECHATRONIC UML components, we distinguish between *discrete components* and *continuous components*. While the behavior of a discrete component is specified by using real-time statecharts, the behavior of a continuous component is specified through methodologies of the control engineering domain [Bur06].

A discrete component can define a number of other components which are contained in the component definition. The RailCab component depicted in figure 2.15 contains the continuous components VelocityController, DistanceController, RadioController and SteeringController, where the component references also include the required inputs and provided outputs denoted by continuous ports (▶).[3]

The VelocityController and the DistanceController are responsible for the acceleration of the RailCab. While the VelocityController computes the desired acceleration for the RailCab, the DistanceController measures the distance to a preceding RailCab and returns the corresponding desired speed.

The RadioController and the SteeringController, on the other hand, are responsible for the correct steering of the wheels. For this, the SteeringController is able to use the RadioController in order to retrieve track data of a base station.

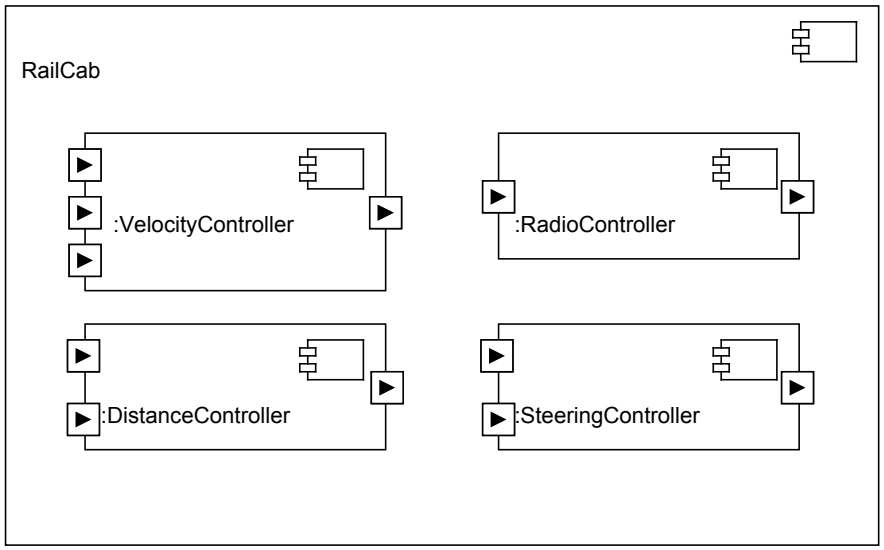

Figure 2.15: RailCab Component Embedding Four Continuous Components

[3]For reasons of simplification, we omitted the type of the inputs at continuous ports.

In order to define the behavior of a discrete component, this implements a number of roles of one or more real-time coordination patterns. In the current approach of MECHATRONIC UML, those roles have to be refined and connected through *synchronization statecharts*, in order to resolve required dependencies between the role implementations.

In order to simplify the example, the RailCab component only implements the rear role of the convoy pattern and the registree role of the registration pattern. The corresponding real-time statechart, also including the synchronization statechart, is depicted in figure 2.16.

The synchronization statechart realizes the requirement that the RailCab has to be registered at a base station in order to build a convoy. This is achieved by refining the rear role statechart by the additional events buildConvoy and notInConvoy and by refining the registree role statechart by the additional events doRegister and doUnregister.

Initially, the synchronization statechart is in state unregistered. As soon as a near base station becomes available, it triggers the registration process by raising a doRegister event, which is received by the refined registree role statechart. In state registered the synchronization statechart rests in state noConvoy for at least 2500 ms, such that the RailCab is able to receive the current track data from the base station through the update mechanism. Then, it may decide to build a convoy by sending the buildConvoy event to the rear role statechart. This in turn sends a convoyProposal to the front role statechart of another RailCab, as soon as it receives a buildConvoy event from the synchronization statechart. As long as the synchronization statechart rests in state convoy, it is not able to unregister from the base station. The reason for this is that the registree role statechart requires a doUnregister event for this, which is only raised from the synchronization statechart if this is in state noConvoy. Consequently, the RailCab can never be in state convoy and unregistered at the same time, which realizes the desired system requirement.

Observe that the specification of the synchronization statechart is a manual, error prone process, as it also has to be checked if the role statecharts are refined properly. If the role statecharts are not refined properly, the verification results produced during the specification of the real-time coordination patterns are not preserved. In this paper, we propose a procedure for specifying the requirements on top of the role statecharts and integrate them automatically (see chapter 3). This way, it can also automatically be verified that the role statecharts are properly refined for the component behavior, such that the necessary verification results are preserved. In the next section we show how reconfigurations of continuous components are considered in the behavioral specifications of MECHATRONIC UML.

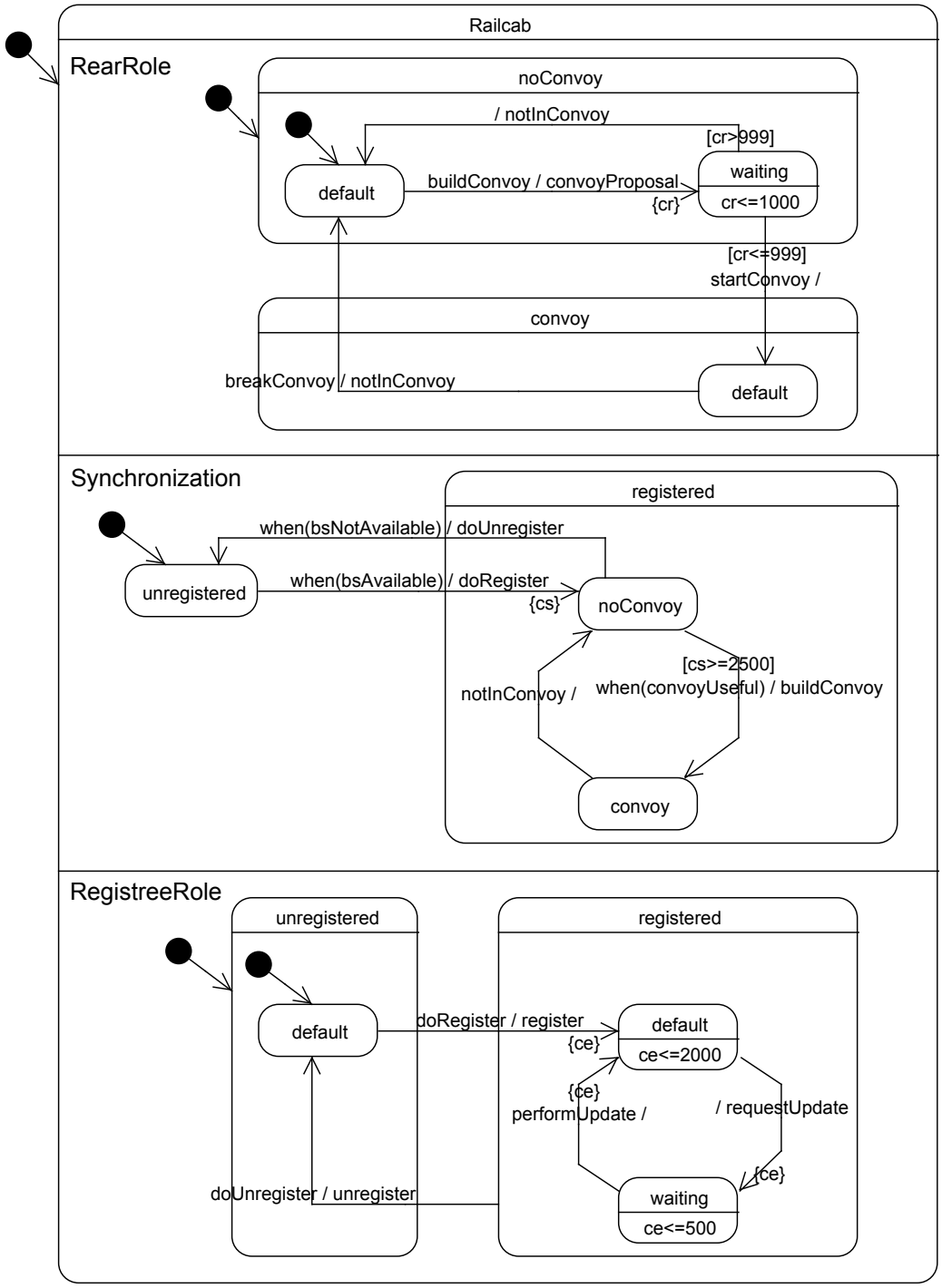

Figure 2.16: Real-Time Statechart for the RailCab Component

2.2.3 Reconfiguration Behavior

For the specification of reconfiguration behavior of a MECHATRONIC UML component, the configurations of the continuous components of a surrounding component are regarded. For the RailCab component (Figure 2.15), for example, it has to be defined in which states of the component's real-time statechart, which continuous component is active, as typically for mechatronic systems, not all components are active at the same time. In MECHATRONIC UML, this is defined by *hybrid reconfiguration charts* [GBSO04, BGO06, BGT05, Bur06] or simply reconfiguration charts, which extend real-time statecharts by component configurations.

The reconfiguration chart for RailCab's rear role behavior is depicted in figure 2.17. It corresponds to the rear role statechart of the convoy coordination pattern (Figure 2.10), but extends this by the controller configurations. While in state noConvoy only the VelocityController is active, in state convoy both the VelocityController and the DistanceController are active. This is due to the fact that in convoy operation mode the distance to the preceding RailCab has to be taken into account to calculate the desired acceleration. If the RailCab is not operating in convoy mode, this is neither necessary nor possible, as a preceding RailCab does not exist. Consequently, the DistanceController can be deactivated to save system resources.

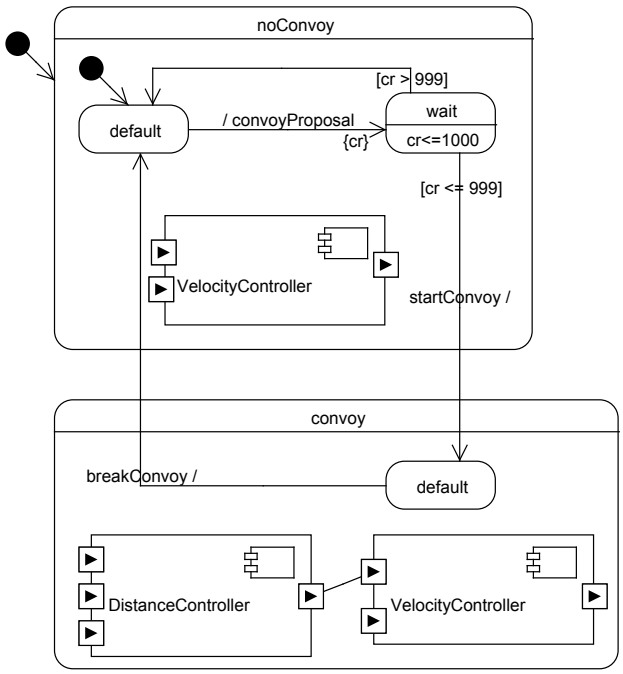

Figure 2.17: Reconfiguration Chart for the RailCab's Rear Role Behavior

The reconfiguration chart of the registree similarly defines the configurations for the RadioController and the SteeringController (see Figure 2.18). As in state unregistered it is not possible to receive any track data from the base station, the RadioController is only active if the registree role statechart is in state registered. Here, it feeds the output of the RadioController into one of the inputs of the SteeringController.

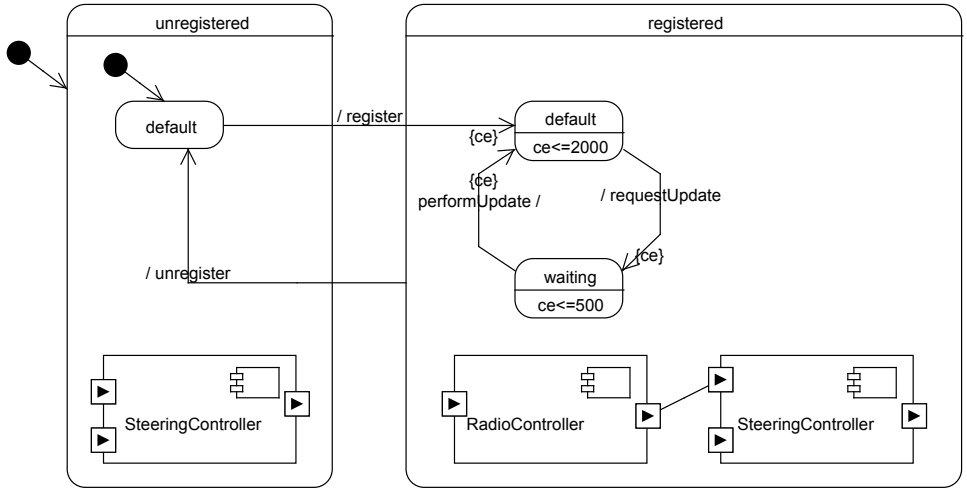

Figure 2.18: Reconfiguration Chart for the RailCab's Registree Role Behavior

Observe that the configurations also have to be regarded when several role statecharts are refined to construct a component behavior. This is necessary due to the limitation of resources in mechatronic systems. It might for example not be safe for the system, that the DistanceController is required by two implementing role statecharts at the same time. Consequently, this conflict has to be resolved when creating the component behavior. It can also be noticed, however, that the configuration specifications in reconfiguration charts can be treated as orthogonal states for the corresponding superstate. Accordingly, restricting combinations of states and combinations of configurations can be dealt with in the same way for the construction of the component behavior. For this reason the synthesis procedure proposed in this paper concentrates on state combinations rather than distinguishing between states and configurations.
In this chapter we introduced the fundamental concepts of this paper. This includes the modeling language MECHATRONIC UML as well as the underlying timed automaton model and the corresponding property specification language TCTL. As the behavioral models of MECHATRONIC UML can be transformed to semantically equivalent timed automata models [BGHS04], the remainder of this paper concentrates on the timed automaton model. In the next chapter we define

the synthesis procedure, which computes the component behavior for a given set of role behaviors and specified composition rules.

Chapter 3

Synthesis of Component Behavior

This chapter forms the main part of this paper. Here, we present the formalisms to define composition rules (section 3.1) and the synthesis algorithm to automatically synthesize the behavior of a MECHATRONIC UML component.

The behavioral models of MECHATRONIC UML, by means of real-time statecharts and reconfiguration charts (see section 2.2), are based on the semantics of timed automata. This means that every real-time statechart and reconfiguration chart can be transformed into a semantically equivalent timed automaton (cf. section 2.2.1, section 2.2.3 and [BGHS04]). Consequently, we define the complete synthesis procedure on the basis of timed automata. This also includes the definition of composition rules.

We begin this chapter by defining the formalisms to specify composition rules (section 3.1). After that, we describe the synthesis procedure, which also includes the definition of an appropriate refinement relation for the synthesized timed automaton (section 3.2).

3.1 Composition Rules

Generally speaking, system properties can be specified in terms of safety and liveness properties for a given behavioral specification [Lam77, Hen92]. *Safety properties* in general state that something bad will never happen during the execution of a program. *Liveness properties* in contrast state that something good will happen eventually. Transferring this to the context of automata synchronizations, these properties always concern two or more automata. Consequently, a *safety property for synchronization* states that something bad will never happen, when executing the corresponding automata in parallel, while a *liveness property for synchronization* expresses that something good will eventually happen during this parallel execution. When we mention these synchronization properties in the fol-

lowing, we will simply refer to them as safety or liveness properties, unless there is a possibility of confusion with the general form of these properties.

Transferring these properties to the composition rules proposed in this paper, we are able to specify both safety and liveness properties. Safety properties can be specified (1) by means of *state composition rules* in terms of forbidden state combinations of the parallel execution and (2) by means of *event composition automata* by adding further time constraints to time guards of selected transitions. Liveness properties in turn can be specified through state composition rules and event composition automata by adding further time constraints to location invariants of location combinations of the parallel execution (cf. progress conditions of timed automata, section 2.1).

In order to specify synchronization behavior for a given set of timed automata, we introduce the syntax and semantics of the composition rule formalisms in this section. As already mentioned above, composition rules are subdivided into state composition rules as well as event composition automata.

3.1.1 State Composition Rules

When roles of coordination patterns are assigned to components, certain combinations of states within their behavioral models might be forbidden due to given system requirements. Consequently, we need a formalism to synchronize the affected timed automata by means of restricted state combinations. This formalism has to fulfill the following requirements: (1) It needs to be easy to use for system developers, such that a developer does not need to learn an entirely new language and (2) it has to be so exact that it can be utilized to automatically remove the specified state combinations from a parallelly composed timed automaton. This formalism is described and defined formally in the following by the notion of *state composition rules*.

Before we give the formal definition of the syntax of state composition rules, we exemplify the formalism using the example of the RailCab project (see section 1.1).

Assume an instance of a RailCab component RC1 taking part in both a convoy and a registration coordination pattern (cf. section 2.2.1). As a consequence, the corresponding automata of the coordination roles are executed in parallel inside of the RC1 component. In this section we abstract from coordination patterns being specified using realtime statecharts and use timed automata instead.[1] The timed automaton for the rear role is depicted in figure 3.1; the timed automaton for the

[1] Note that every realtime statechart can actually be transformed into a semantically equivalent timed automaton (see section 2.2.1). We do not consider this transformation in this section, though, in order to simplify the illustrating examples.

registree role is depicted in figure 3.2. Both automata correspond to the behaviors described in the previous chapter (see section 2.2.1).

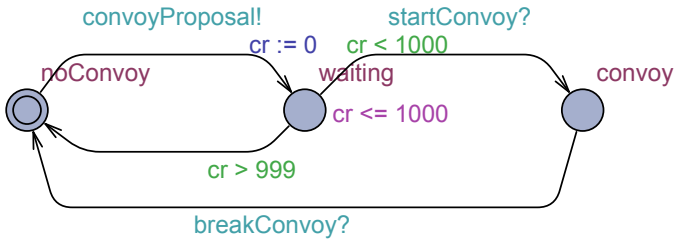

Figure 3.1: Rear Role Timed Automaton of the Convoy Pattern

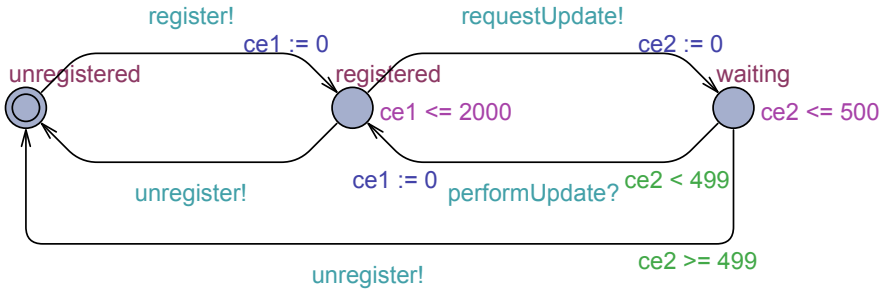

Figure 3.2: Registree Role Timed Automaton of the Registration Pattern

When running both automata within one component, assume that a system requirement states that a RailCab has to be registered in order to form a convoy. Or specified explicitly as a safety property, it shall never happen that a RailCab is in state convoy and in state unregistered at the same time.

If this requirement had to be realized manually it would be a time consuming and error prone task for a developer. Both role automata would have to be refined, such that the earlier verified requirements do not have to be verified again. Furthermore, the developer might not even know if it can be realized at all within the scope of a refinement which preserves earlier verification results.

We propose the formalism of *state composition rules* to specify these types of requirements on top of the independent role automata. The above stated requirement can for example be expressed with the following state composition rule r_1:

$$r_1 \quad = \quad \neg((unregistered, true) \wedge (convoy, true)).$$

Syntactically, a state composition rule consists of a set of so-called location predicates connected by the Boolean relations meet and join, all together surrounded by a negation. A location predicate specifies a location in combination with a set of clock valuations, while this combination is not permitted in relation with other location predicates specified in that rule. The clock valuations of a location predicate are specified by clock constraints already known from the timed automata formalism itself. In the state composition rule r_1, the location predicates $(unregistered, true)$ and $(convoy, true)$ are connected with a Boolean meet. The clock valuations of both location predicates are specified with $true$, as for all clock valuations the constellation of state unregistered and convoy is not permitted.

Assume another, weaker form of this requirement, stating that the combination of locations unregistered and convoy is allowed, but not for more than 50 time units. Note that this requirement actually forms a liveness property for the system, as it expresses that the combination of locations unregistered and convoy is allowed, but must be left within 50 time units. In the formalism of state composition rules this is specified by the following rule r_2:

$$r_2 \quad = \quad \neg((unregistered, true) \wedge (convoy, cr > 50)).$$

In this state composition rule r_3, the combination of states $unregistered$ and $convoy$ is forbidden for any clock valuation of the clocks of $rear$, where $cr > 50$. In detail, this is expressed using the location predicate $(convoy, cr > 50)$, which describes all states of the system, where the rear role automaton is in state $convoy$ and the clock valuation of cr is smaller than 50. Summarizing this, we are able to add clock constraints to location predicates, such that certain constellations of system states can be further forbidden for a given valuation of clocks only.

As we have given an example of state composition rules above, we proceed with the formal definition of the syntax of state composition rules. We use this formalization for the definition of the synthesis algorithm (section 3.2) in order to integrate state composition rules automatically in the component's behavioral model.

Before we can give a formal definition of the syntax of location predicates used in state composition rules, we need to define the form of clock constraints allowed for location predicates. We do this by the type of *upwards closed clock constraints* in the following.

Definition 3.1.1 (Upwards Closed Clock Constraint)
For a set C of clocks, the set $\Phi_{uc}(C) \subset \Phi(C)$ of upwards closed clock constraints *is inductively defined by the grammar*

$$\varphi ::= x \sim n \mid x - y \sim n \mid \varphi \wedge \varphi \mid true,$$

where $x, y \in C$, $\sim \in \{\geq, >\}$, $n \in \mathbb{N}$.

Upwards closed clock constraints only allow to specify clock constraints or conjunctions of clock constraints which describe lower bounds for a clock or the difference of two clocks. In the context of state composition rules this is used to describe the lower bound of the forbidden time interval for a location. Upper bounds are not permitted at this point, as their evaluation might result in a location invariant for a location which is not downwards closed. This in turn is not permitted for the syntax of timed automata (see section 2.1.1). To give a small example for this, assume the location convoy having the location invariant $I(convoy) = cr < 500$. Furthermore assume the state composition rule r_2 referring to the location predicate $(convoy, cr < 50)$ instead of $(convoy, cr > 50)$. As the state composition rule specifies the forbidden time interval, the evaluation of the rule r_2 would result in a location invariant of convoy being $I(convoy) = cr \geq 50 \wedge cr < 500$, which is not allowed by the syntax of timed automata (see Definition 2.1.4).

Using the definition of upwards closed clock constraints, we can give a proper formal definition of the syntax of *location predicates* in the following.

Definition 3.1.2 (Location Predicate)
For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$, a location $l \in L$ and an upwards closed clock constraint $\varphi \in \Phi_{uc}(C)$ the set $\Gamma(A)$ of location predicates $\gamma = (l, \varphi)$ is defined by

$$\Gamma(A) = L \times \Phi_{uc}(C).$$

As already shown in the examples, a location predicate is the combination of a location and downwards closed clock constraint for a timed automaton A, used to define a forbidden set of clock valuations φ for a timed automaton location l. If all clock valuations shall be forbidden for a location, this can be expressed by $(l, true)$, as $true \in\in \Phi_{uc}(C)$. Semantically, this means that the location itself is not allowed in combination with the other location predicates defined in the corresponding state composition rule. Examples for state predicates have already been given above in the state composition rules r_1 and r_2.

We finally define the syntax of *state composition rules* in the following, which combines several location predicates for two different timed automata in the Boolean relations meet and join and explicitly declares this relation as forbidden using the negation.

Definition 3.1.3 (State Composition Rule)
For two timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ and $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ the set $R^S(A_1, A_2)$ of state composition rules *ρ is defined*

by the grammar

$$\begin{aligned} \rho &::= \neg\rho_\gamma \\ \rho_\gamma &::= \rho_\gamma \wedge \rho_\gamma \mid \rho_\gamma \vee \rho_\gamma \mid \gamma \end{aligned}$$

where $\gamma \in \Gamma(A_1) \cup \Gamma(A_2)$.

The first part of the grammar defines that each state composition rule is surrounded by a negation, in order to emphasize the fact that it specifies a forbidden combination of location predicates for the parallel execution of the given timed automata. The second part specifies the possibility of using the Boolean operators meet and join in order to express which combination of location predicates is forbidden. Note that for reasons of simplicity, a negated atomic location predicate is also allowed by the grammar, although this cannot specify a compositional property for the input automata. For the same reason, we do not distinguish explicitly between location predicates of A_1 and location predicates of A_2. Only using location predicates of A_1 or A_2 also would not lead to the specification of a compositional property for A_1 and A_2. Examples of state composition rules have already been given with r_1 and r_2. A more sophisticated rule, stating that additionally cr must be smaller or equal to 250, if the rear role automaton is in location waiting and the registree role automaton is in state unregistered also $(waiting, cr > 250)$, is specified by the rule r_3:

$$r_3 \;=\; \neg((unregistered, true) \wedge ((waiting, cr > 250) \vee (convoy, true))).$$

In this section we introduced compositional rules, which are used to specify safety or liveness properties for the composition of two timed automata. We also gave formal definitions for the syntax of state composition rules. Later, we show how state composition rules can be applied automatically for the parallel execution of those automata (section 3.2.2). In the next section we introduce the notion of event composition automata, which are used to describe compositional properties for event occurrences or sequences of event occurrences.

3.1.2 Event Composition Automata

With the help of state composition rules we are able to synchronize the behavior of pattern role automata with respect to specified state combinations. They do not provide the possibility though, to synchronize the automata on the basis of events and event sequences. For this, we define *event composition automata* in this section.

We start describing the basic concepts of event composition automata and exemplify them using the example of the convoy and the registration coordination pattern of the RailCab project again (see previous section).

For the specification of communication behavior for roles of coordination patterns, timed automata are used in order to define the possible timed event sequences for each role. For the specification of intra-component synchronization behavior between these roles, we propose the formalism of *event composition automata*. For those, we also apply the syntax of timed automata themselves, as event composition automata are also used to describe possible event sequences of the component behavior. In contrast to pattern role automata, event composition automata do not add any further event occurrences, which means that they do not consume or provide any signals from the channels of the corresponding role automata. In other words, event composition automata are only monitoring event occurrences for a given set of role automata while they do not distinguish between sending or receiving events. They do, however, allow to add further timing constraints to the monitored event occurrences, also in terms of location invariants for the locations between the monitored events. This way, safety and liveness properties for the synchronization of several role automata can be specified.

For the pattern role automata of the rear role and the registree role (Figure 3.1 and Figure 3.2), assume a further system requirement stating that a railcab has to be registered to a base station for at least 2500 time units before starting a convoy. Again, this requirement specifies behavior which involves more than one role behavior and as a consequence has to be specified as synchronization behavior in order to preserve verification results. Furthermore, observe that this requirement cannot be implemented using a state composition rule, as it is based on the occurrence of the startConvoy? event of the rear role automaton. Accordingly, we specify the event composition automaton eca_1 (Figure 3.3) to implement this requirement.

The event composition automaton eca_1 switches from its initial state ec_initial to ec_registered along with the occurrence of the register! event. At the same time it resets its clock ec_c1. Resting in location ec_registered, it either switches back to ec_initial on the occurrence of unregister! or it switches to ec_registeredConvoy on the occurrence of startConvoy?. Note that the transition from ec_registered to ec_registeredConvoy is annotated with the time guard ec_c1 >= 2500, which implies that the corresponding component has to be in state ec_registered for at least 2500 time units. Thus, this clock constraint together with the clock reset before implements the above given requirement. Once in ec_registeredConvoy, eca_1 changes its location only on the occurrence of the event unregister!, as in this situation the monitoring has to be started once more from the initial location. In all other situations, the component changes its state of being registered and therefore this event composition rule does not have to add any further constraints.

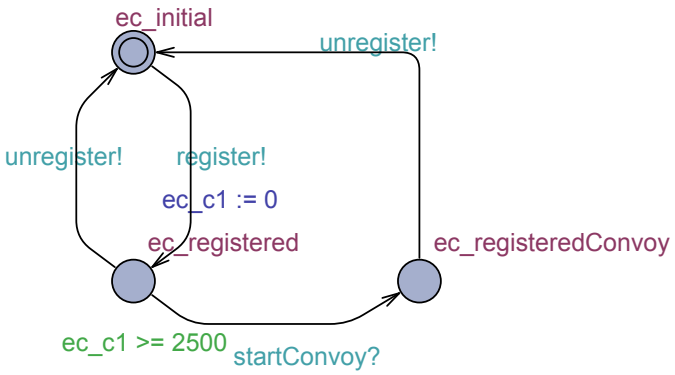

Figure 3.3: Event Composition Automaton eca_1

We introduced the formalism of event composition automata above by informally describing the basic concepts and exemplifying them using the convoy example. In the following, we give the formal definition of event composition automata. This is later used to describe how event composition automata can be automatically integrated into the parallel execution of the corresponding pattern role automata.

Definition 3.1.4 (Event Composition Automaton)
Let $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ *be two timed automata. An* event composition automaton $A_E \in R^A(A_1, A_2)$ *is again a timed automaton as a tuple* $(L_E, l_E^0, \Sigma_E, C_E, I_E, T_E)$ *, where*

- L_E *is a finite non empty set of locations,*
- $l_E^0 \subseteq L$ *is the initial location,*
- $\Sigma_E \subseteq \Sigma_1 \cup \Sigma_2$ *is the finite set of events to be observed,*
- $I : L \rightarrow \Phi_{dc}(C_E)$ *assigns each location a downwards closed clock constraint,*
- C_E *is a finite set of clocks, with* $C_E \cap (C_1 \cup C_2) = \varnothing$
- $T_E \subseteq L_E \times \Sigma_E \times \Phi(C_E) \times 2^{C_E} \times L_E$ *is a finite set of transitions* $t = (l, e, g, r, l') \in T_E$*, with*
  - $l \in L_E$ *is the source location,*
  - $e \in \Sigma_E$ *is the observed event,*
  - $g \in \Phi(C_E)$ *is the time guard,*

- $r \subseteq C_E$ *is a set of clocks to be reset, and*
- $l' \in L_E$ *is the target location.*

For two timed automata A_1 and A_2, we define the syntax of an event composition automaton as a timed automaton (see Definition 2.1.4), which only uses events from A_1 and A_2 and whose set of clocks is disjoint to the set of clocks of A_1 and to the set of clocks of A_2.

Semantically, an event composition automaton only observes event occurrences of the given role automata. Consequently, only those events can be used in an event composition automaton, as others can never be observed. Additionally, the set of clocks of the event composition automaton is restricted to be disjoint to the set of clocks of the role automata. This way, it is guaranteed that the event composition automaton cannot widen the time intervals of event sequences of the automata to be synchronized. If the event composition automaton could widen those time intervals, it might happen that earlier verified deadlines of the role automata cannot be met anymore. Consequently, verification results could not be guaranteed to be preserved after performing the synthesis.

Summarizing this, an event composition automaton observes a sequence of event occurrences and possibly adds further time constraints to the corresponding transitions and to locations inbetween the observed events.

In this section, we defined the formalism of event composition automata as an adequate formalisms to explicitly define synchronizations between separate role automata on the basis event sequences. In the scope of the synthesis procedure defined in this paper, event composition automata and state composition rules are used to specify safety and liveness properties for the synchronization of a set of pattern role automata. In the next section, we show how composition rules can be automatically applied to the role automata and how it is guaranteed that the externally visible behavior of the particular role automata is preserved.

3.2 Synthesis Algorithm

As we defined the input for the synthesis algorithm, which are the composition rules and the pattern role automata in the preceding sections, we can proceed with the specification of the algorithm itself. For reasons of simplification we only give examples and definitions for the procedure based on two role automata. The concept can be easily transfered to an arbitrary number of role automata, by extending the definitions concerning the two input timed automata to sets of timed automata.

The synthesis algorithm is divided into four distinct steps (see Figure 3.4), which will be described in detail in the following sections. First, the parallel com-

position of the role automata is computed (section 3.2.1), which forms an explicit model for the parallel execution of the pattern role automata. On this parallelly composed timed automaton the composition rules are applied, by removing the forbidden system states specified by the state composition rules (section 3.2.2) and by including the specified event composition automata in the parallelly composed automaton (section 3.2.3). In the last step, it is verified that the externally visible behavior of the particular role automata is preserved, as the changes made on the parallelly composed automaton by means of the appliance of composition rules might lead to violations of properties of the original role behaviors (section 3.3).

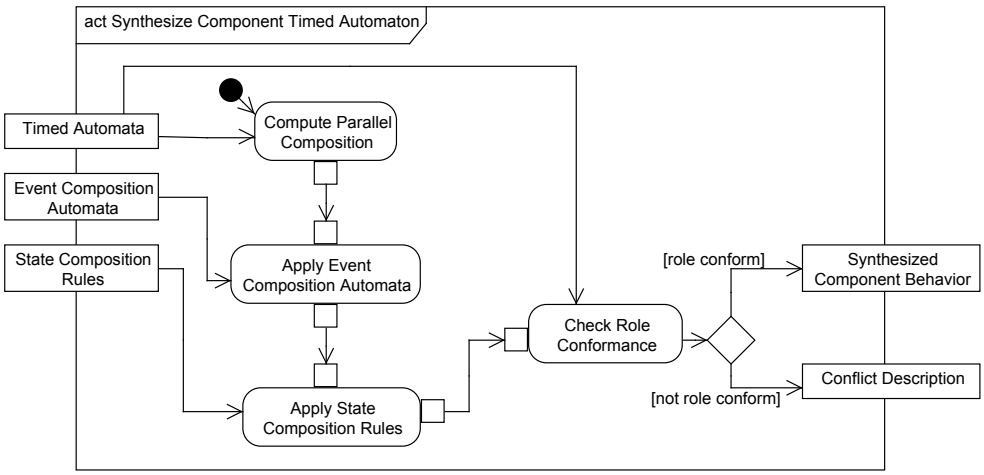

Figure 3.4: Synthesis Algorithm for Timed Automata

3.2.1 Parallel Composition

The parallel composition of the pattern role automata forms an explicit model for the parallel execution of these automata. This parallel composition is employed to apply the composition rules by means of state composition rules and event composition automata. The synchronization behavior specified by these composition rules is defined for the parallel execution of the concerned role automata. Hence, the explicit construction of the model is necessary in order to apply the composition rules afterwards. After accomplishing the synthesis procedure, this model can be applied for further analysis of intra-component properties but also for code generating purposes.

The parallel composition applied in this paper is derived from the parallel composition operator of the process algebra *Calculus of Communicating Systems*

(CCS) [Mil89]. This has already been transfered to timed automata by the notion of *networks of timed automata* in [YPD94, Pet99]. The theory of networks of timed automata has also successfully been implemented in the real-time model checker UPPAAL [BDL04].

The central idea of networks of timed automata is to construct a product automaton of the given automata, which allows both synchronization and interleaving of events. However, the pattern role automata applied to one MECHATRONIC UML component are defined such that they are independent from each other, in order to allow compositional model checking. Consequently, we do not need to consider synchronizations in the parallel composition defined here.

Before we formally define the parallel composition, we exemplify it by using the convoy and registration coordination pattern again. In order to reduce the state space in the illustration, we use the simplified versions of the rear role and the registree role automaton depicted in figure 3.5 and figure 3.6 (cf. Figure 3.1 and Figure 3.2).

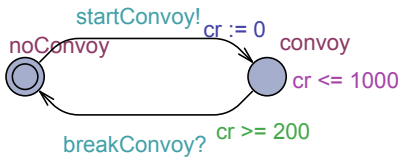

Figure 3.5: Simplified Rear Role Timed Automaton

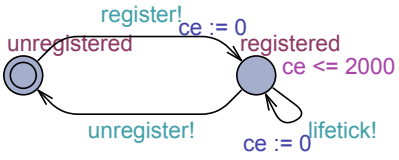

Figure 3.6: Simplified Registree Role Timed Automaton

The number of locations of both automata is reduced to two. The simplified rear role automaton can now immediately start the convoy by sending a startConvoy!. It can at most rest in convoy for 1000 time units but must stay in this location for at least 200 time units. In the simplified version of the registree role automaton, the data reception mechanism is replaced by a simple lifetick which has to be called at least every 2000 time units.

The parallel composition of the simplified rear role and the simplified registree role automaton is the product automaton depicted in figure 3.7. In this product automaton, each location of the rear role automaton is combined with each location of the registree role automaton. Accordingly, the location invariant of the composed location is the conjunction of each corresponding single invariant. The set of incoming and outgoing transitions of a composed location is the union of the incoming and outgoing transitions of each corresponding single location. This way, an interleaving of events is achieved, as an event of the rear role automaton and an event of the registree role automaton can never occur at the same transition. A parallel execution of two distinct events is still possible, though, as the transitions are executed in zero-time and time does not have to pass in locations.

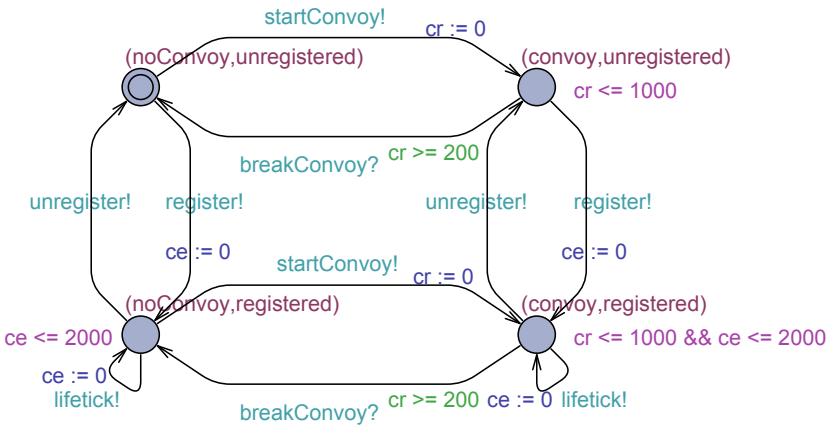

Figure 3.7: Parallely Composed Timed Automaton

A possible path in this automaton might for example be register! from (noConvoy,unregistered) to (noConvoy,registered) followed by startConvoy! from (noConvoy,registered) to (convoy,registered), while all clocks' values rest at zero. Semantically, this means that register! and startConvoy! occur at the same time. Altogether, this automaton describes exactly the behavior of the parallel execution of both role automata being started at the same time.

We proceed with the formal definition of the parallel composition. This is in its general form again a timed automaton, which will later be used to define the appliance of composition rules.

Definition 3.2.1 (Parallel Composition)
Let $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ *be two timed automata with* $C_1 \cap C_2 = \varnothing$ *and* $\Sigma_1 \cap \Sigma_2 = \varnothing$. *We define the parallel composition* $A_1 \parallel A_2$ *as a product automaton* $A_P = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$, *where*

- $L_P = L_1 \times L_2$,
- $l_P^0 = (l_1^0, l_2^0)$,
- $\Sigma_P = \Sigma_1 \cup \Sigma_2$,
- $I_P : L_P \rightarrow \Phi(C_1) \cup \Phi(C_2)$ *with* $I_P((l_1, l_2)) = I_1(l_1) \wedge I_2(l_2)$,
- $C_P = C_1 \cup C_2$,
- $T_P \subseteq L_P \times \Sigma_P \times \Phi(C_P) \times 2^{C_P} \times L_P$, *with*
  - $((l_1, l_2), e_1, g_1, r_1, (l_1', l_2)) \in T_P \Leftrightarrow (l_1, e_1, g_1, r_1, l_1') \in T_1$, *and*

– $((l_1, l_2), e_2, g_2, r_2, (l_1, {l_2}')) \in T_P \Leftrightarrow (l_2, e_2, g_2, r_2, {l_2}') \in T_2$.

This definition of parallel composition defines exactly the above exemplified concepts of the parallel execution of the two non-synchronizing automata A_1 and A_2. The set of locations is constructed by the cross product of the set of locations of each automaton. Accordingly, the initial location l_P^0 is the combination of the two single initial locations l_1^0 and l_2^0. As the automaton describes exactly the behaviors of both input automata, the set of events Σ_P is also exactly the union of all events Σ_1 of A_1 and Σ_2 of A_2. The set of clocks C_P is constructed analogously. For the invariant of the composed location $I((l_1, l_2))$ the invariants of the single locations $I(l_1)$ and $I(l_2)$ are connected by a Boolean meet, as both invariants have to be considered in each state of the parallel execution. The set of transitions T_P exactly reflects the interleaving character of event occurrences of the parallel execution of both single automata. Either the transition corresponds to a transition of A_1 or to a transition of A_2. If the corresponding transition is in T_1, the location of A_1 changes from l_1 to ${l_1}'$ in the composed location; if it is in T_2 the location of A_2 changes from l_2 to ${l_2}'$ in the composed location. Note once more, that in contrast to networks of timed automata [YPD94] or the parallel composition of CCS [Mil89], we do not allow the synchronization of events, as the pattern role automata of MECHATRONIC UML have to be independent.

Above, we defined an explicit model for the parallel execution of the pattern role automata. Hence, we can apply state composition rules and event composition automata to this model as described in the following two sections.

3.2.2 Applying State Composition Rules

In the preceding section, we defined the parallel composition of pattern role automata as an explicit model for the parallel execution of those. State composition rules, as defined in section 3.1.1, specify both safety and liveness properties for this parallel execution. Accordingly, the parallelly composed timed automaton has to be modified according to the specified state composition rules. This way, an explicit model for the parallel execution of role automata is obtained, which also considers the specified safety and liveness properties of state composition rules.

Before we formally define how state composition rules are evaluated and applied to a parallelly composed timed automaton, we exemplify this by describing the appliance of the composition rules r_1 and r_2 (see section 3.1.1, pp. 52–56) to the parallelly composed automaton constructed in the preceding section (Figure 3.7).

We first examine $r_1 = \neg((unregistered, true) \wedge (convoy, true))$, which specifies that for any clock valuation the combination of locations unregistered and convoy is not permitted for the parallel execution. Consequently, we search

the locations of the parallelly composed automaton, and remove all locations from this automaton, which refer to the single locations unregistered and convoy. As the example presented here is constructed only of the two timed automata of the rear role and the registree role, the only location to be removed is the location (convoy,unregistered). The resulting automaton is depicted in figure 3.8.

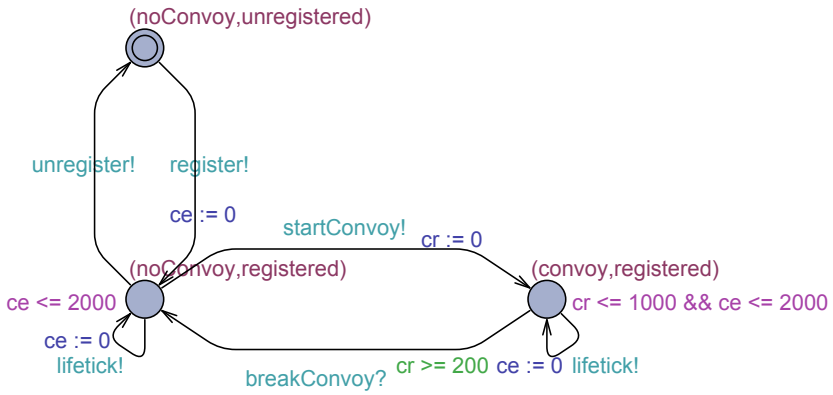

Figure 3.8: State Composition Rule r_1 Applied

Assume that in the general case it might be too restrictive not to allow the constellation of the locations unregistered and convoy at all, as specified by r_1. Accordingly, we apply the rule $r_2 = \neg((unregistered, true) \wedge (convoy, cr > 50))$ instead, as described in the following. This rule states that unregistered and convoy is allowed, but not for clock valuations, where $cr > 50$. Following, we search the locations again for the location (convoy,unregistered), but this we time do not remove this location. Instead, we negate the clock constraint $cr > 50$ and add this to the location invariant of (convoy,unregistered), resulting in the new location invariant $cr \leq 1000 \wedge cr \leq 50$, which is equal to $cr \leq 50$. The resulting parallelly composed automaton is depicted in figure 3.9. Here, the location invariant of the top right location (convoy,unregistered) changed from cr <= 1000 to cr <= 50 (cf. Figure 3.7).

Revisiting the appliance of the rule r_1, it is actually evaluated in the same way. The negation of the clock constraint $true$ results in $false$ and $cr \leq 1000 \wedge false$ also results in false for the location invariant of (convoy,unregistered). Obviously, a location with $false$ as its invariant can never be reached. Consequently, the composed location is simply removed from the automaton, including its incoming and outgoing transitions.

We briefly described how state composition rules are evaluated and applied to a parallelly composed timed automaton, using the convoy and registration pattern

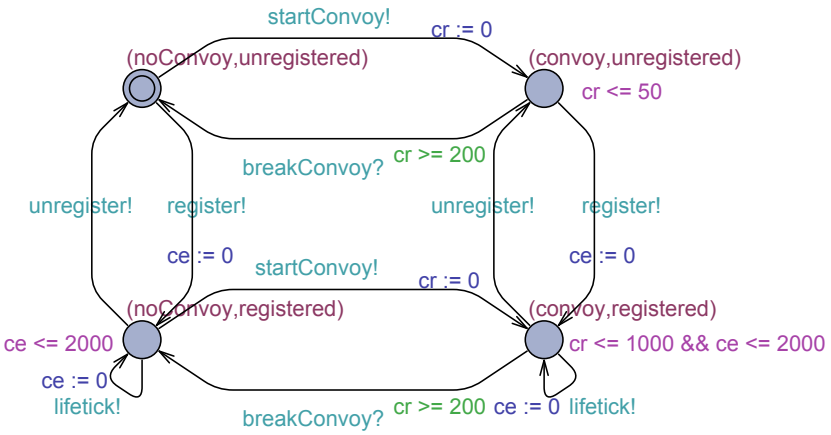

Figure 3.9: State Composition Rule r_2 Applied

example. We proceed giving the exact definitions of the evaluation of a state composition rule in the following.

For a given parallelly composed location l and a given state composition rule r, the first step is to evaluate the location predicates of r in order to find out if the location invariant of the location l is affected by the rule r. This is defined by the *location predicate evaluation* below.

Definition 3.2.2 (Location Predicate Evaluation)
Given two timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$, $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$, *their parallelly composition* $A_P = A_1 \parallel A_2 = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$, *a corresponding parallel composed location* $l_p = (l_1, l_2)$, *and a location predicate* $\gamma = (l, \varphi)$ *with* $l \in L_1 \cup L_2$ *and* $\varphi \in \Phi_{uc}(C_1) \cup \Phi_{uc}(C_2)$ *the* location predicate evaluation *is a function* $\gamma : L_P \rightarrow \Phi_{uc}(C_P) \cup \{false\}$ *defined with*

$$\gamma(l_p) = \begin{cases} \varphi, & \textit{iff } (l = l_1) \vee (l = l_2), \\ false, & else. \end{cases}$$

The *location predicate evaluation* returns the forbidden clock valuations in form of a clock constraint for a parallelly composed location (l_1, l_2) and a given location predicate (l, φ). If one of the locations referred to in the composed location is equal l, the forbidden set of clock valuations is exactly described by φ. If none of those locations is equal to l, permitting the clock valuations described by φ is not applicable, and therefore the set of forbidden clock valuations is described by $false$. This means that for the location (l_1, l_2) no clock valuation is restricted by the location predicate (l, φ).

We exemplify this for the location predicate $\gamma_1 = (convoy, cr > 50)$ and the locations $(noConvoy, unregistered)$ and $(convoy, registered)$ in the following equations:

$$\begin{aligned} \gamma_1((noConvoy, unregistered)) &= false, \\ \gamma_1((convoy, registered)) &= cr > 50. \end{aligned}$$

The state predicate evaluation is used as part of each *state composition rule evaluation* in order to return the permitted set of clock valuations for each composed location (l_1, l_2). The state composition rule evaluation is defined as follows.

Definition 3.2.3 (State Composition Rule Evaluation)
Given two timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$, $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$, *their parallel composition* $A_P = A_1 \parallel A_2 = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$, *a corresponding parallelly composed location* $l_p = (l_1, l_2)$, *and a state composition rule* $\rho \in R^S(A_1, A_2)$ *the* state composition rule evaluation *is a function* $\rho : L_P \rightarrow \Phi_{dc}(C_P) \cup \{false\}$ *defined with*

$$\rho(l_p) = \begin{cases} \neg\rho_1(l_p), & \text{iff } \rho \text{ is of the form } \neg\rho_1, \\ \rho_1(l_p) \wedge \rho_2(l_p), & \text{iff } \rho_\gamma \text{ is of the form } \rho_1 \wedge \rho_2, \\ \rho_1(l_p) \vee \rho_2(l_p), & \text{iff } \rho_\gamma \text{ is of the form } \rho_1 \vee \rho_2, \\ \gamma(l_p), & \text{iff } \rho_\gamma \text{ is the literal } \gamma. \end{cases}$$

where $\gamma \in \Gamma(A_1) \cup \Gamma(A_2)$.

Each state composition rule consists of the negation of location predicates or conjunctions and disjunctions of location predicates. Accordingly, all these forms have to be considered within the evaluation of a state composition rule.

The negated form of a state composition rule is simply evaluated by negating the result of the unnegated rule by the rules of Boolean algebra. Note that the negation of an upwards closed clock constraint is obtained by inverting the relational operators "$>$" to $\leq$ and "$\geq$" to "$<$". The conjunctive and disjunctive forms of state composition rules are in turn evaluated by applying the corresponding Boolean operators on the evaluations of those rules. Finally, the evaluation of an atomic location predicate is evaluated by means of the location predicate evaluation as defined above (Definition 3.2.2).

The evaluation of the state composition rule r_1 is exemplified for the composed location $(convoy, unregistered)$ in the following equation:

$$\begin{aligned} & r_1((convoy, unregistered)) \\ &= \neg((unregistered, true) \wedge (convoy, true)) \\ &= \neg(true \wedge true) \\ &= \neg true \\ &= false. \end{aligned}$$

As already described above, the evaluation of $r_1((convoy, unregistered))$ results in $false$. In conjunction with the invariant $I((convoy, unregistered))$ this is again equal to $false$, which results in the removal of the location $(convoy, unregistered)$ from the parallelly composed timed automaton.

To exemplify the case where the evaluation results in a new invariant for the corresponding location, we also give the evaluation of r_2 for the location $(convoy, unregistered)$ in the following:

$$\begin{aligned} & r_2((convoy, unregistered)) \\ = \; & \neg((unregistered, true) \wedge (convoy, cr > 50)) \\ = \; & \neg(true \wedge cr > 50) \\ = \; & \neg(cr > 50) \\ = \; & cr \leq 50. \end{aligned}$$

In conjunction with the invariant $I((convoy, unregistered)) = cr \leq 1000$ this results in the new invariant $I'((convoy, unregistered)) = cr \leq 50$ (see also Figure 3.9).

On the basis of the definitions of the location predicate evaluation and the state composition rule evaluation we proceed with the definition of the *state composition conform* timed automaton, which is the resulting automaton after a given set of state composition rules have been applied.

Definition 3.2.4 (State Composition Conformance)
Let $A_P = A_1 \parallel A_2 = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$ be the parallel composition of the timed automata A_1 and A_2. Further let $R_1^S \subseteq R^S(A_1, A_2)$ be a set of state composition rules specified over A_1 and A_2. The state composition conform*, parallelly composed timed automaton $A_{SC} = (L_{SC}, l_{SC}^0, \Sigma_{SC}, C_{SC}, I_{SC}, T_{SC})$ is defined with*

- $L_{SC} = L_P \setminus L_R$, *where* $L_R = \{l_p \mid l_p \in L_P$ *and* $\forall\, \rho_1, \ldots, \rho_n \in R_1^S : I(l_p) \wedge \rho_1(l_p) \wedge \ldots \wedge \rho_n(l_p) = false\}$,
- $l_{SC}^0 = l_P^0 \Leftrightarrow l_P^0 \in L_{SC}$,
- $\Sigma_{SC} = \Sigma_P$,
- $I_{SC} : L_{SC} \rightarrow \Phi(C_{SC})$ *with* $I_{SC}(l_p) = I_P(l_p) \wedge \rho_1(l_p) \wedge \ldots \wedge \rho_n(l_p), \forall\, \rho_1, \ldots, \rho_n \in R_1^S$,
- $C_{SC} = C_P$,
- $T_{SC} \subseteq L_{SC} \times \Sigma_{SC} \times \Phi(C_{SC}) \times 2^{C_{SC}} \times L_{SC}$, *with* $(l_p, e, g, r, {l_p}') \in T_{SC} \Leftrightarrow (l_p, e, g, r, {l_p}') \in T_P \wedge l_p, {l_p}' \in L_{SC}$.

For a given set of state composition rules R_1^S, the state composition conform timed automaton is defined as the timed automaton where each of the composition rules $\rho \in R_1^S$ has been applied to each of the locations $l_P \in L_P$.

The set of state composition conform locations L_{SC} is the original set of locations L_P, without those locations in L_R which are restricted by the state composition rules. The restricted locations L_R are those, whose invariant $I_P(l_p)$ in conjunction with each of the state composition rule evaluations $\rho_n(l_p)$ is $false$.

The initial location l_{SC}^0 of the state composition conform is the same as the one of the parallelly composed timed automaton, as long as it is not removed by the appliance of the state composition rules. The set of events and the set of clocks stays the same.

The invariant of each location $I_{SC}(l_p)$ is defined as the conjunction of the original invariant $I_P(l_p)$ with each of the state composition rules' evaluations $\rho_n(l_p)$. Note at this point, that the invariant does not change for state composition rules that are not applicable for the current location. The reason for this is that the evaluation in this case simply results in the clock constraint $true$ (see Definition 3.2.2), which is the identity element for conjunction in Boolean algebra.

The set of transitions T_{SC} of the state composition conform timed automaton is the set of transitions T_P of the parallel composition without those transitions which are incoming or outgoing transitions of restricted locations.

We described and formally defined how state composition rules are applied to a parallelly composed timed automaton in order to obtain a state composition conform timed automaton. In the next section we show how event composition automata are applied to a parallelly composed timed automaton.

3.2.3 Applying Event Composition Automata

We described how synchronization properties specified by state composition rules are applied to the parallel composition of pattern role automata in the preceding section. Event composition automata specify additional synchronization behavior for this parallel composition by means of a special type of timed automata (see section 3.1.2). Accordingly, event composition automata also have to be applied to the explicit model of the parallel execution of the pattern role automata, which is the parallelly composed timed automaton (see section 3.2.1). This forms the third step in the synthesis algorithm, which implies that state composition rules (if specified) have already been applied to the parallel composition. Consequently, in this section we describe how event composition automata are applied to a parallelly composed, state composition conform timed automaton.

We proceed with the basic concepts of the event composition automata appliance, followed by an exemplification using again the parallel composition of the

simplified rear role and registree role automaton. Finally, we give the formal definition of an event composition conform timed automaton in the last part of this section.

Similar to the parallel composition used for the parallel execution of the role automata (see section 3.2.1), applying event composition automata can also be compared to the parallel composition operator of the process algebra *Calculus of Communicating Systems (CCS)* [Mil89] or the *networks of timed automata* formalism defined in [YPD94]. The fundamental difference is that for the event composition automaton appliance only synchronization of events is taken into account, as event composition automata do not define any new event occurrences for the parallel execution. Consequently, the result is a product automaton where the locations of the parallel composition are multiplied with the locations of the event composition automaton. Furthermore, the transitions of the event composition automaton are synchronized with the transitions of the parallel composition, although they do not take the channel concept into account. This means that a synchronization between a parallelly composed transition and an event composition automaton transition does not change the event, as the event composition automaton only observes the event occurrences of the parallel execution. This type of synchronization can also be denoted as *silent*, as it does not change the external behavior of the parallel composition. Observe that this also forms a noticeable difference from the synchronizations described by the parallel composition of CCS and the concept described in networks of timed automata.

What the appliance can change is (1) the set of clock resets of synchronized transitions, (2) the time guards of synchronized transitions, and (3) the location invariants of corresponding parallelly composed locations. All of this is only possible in the scope of the set of clocks of the event composition automaton, which has to be disjoint to the set of clocks of the parallelly composed timed automaton. This way, it is achieved that the additional constraints do not widen the time intervals specified within the parallel composition, which then again guarantees that earlier verified deadlines are still met after applying the event composition automata. The addition of further time constraints is described in detail below.

For the exemplification of the event composition automaton appliance we use the modified version eca_2 (Figure 3.10) of the event composition automata eca_1 (Figure 3.3) which is described in section 3.1.2 (pp. 56–59). This automaton is modified, such that it corresponds to parallel composition of the simplified versions of the rear role and the registree role automaton (cf. Figure 3.7). The only modification is the bottom transition's event which is changed from startConvoy? to startConvoy! in accordance with the simplified rear role automaton (Figure 3.5).

We apply the event composition automaton eca_2 to the parallel composition of the simplified rear role and registree role automaton, where the state composition

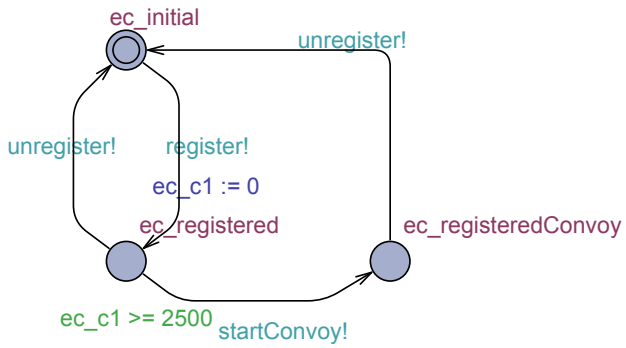

Figure 3.10: Modified Version eca_2 of the Event Composition Automaton eca_1 (cf. Figure 3.3)

rule r_1 has already been applied (Figure 3.8). This results in the timed automaton depicted in figure 3.11. Note that every location of this automaton refers to both the locations of the role automata as well as the locations of the event composition automaton eca_2. Furthermore, observe that only those locations which are reachable through synchronized transitions starting in the initial composed location $(noConvoy, unregistered, ec_initial)$ are part of the automaton. The location $(noConvoy, unregistered, ec_registered)$ for example can never be reached from the initial location and therefore it is not included.

Informally, the construction can be described as follows. Starting in the initial location $(noConvoy, unregistered, ec_initial)$, for each outgoing transition of the parallelly composed location, the event composition automaton also changes its location, if there is a transition in the event composition automaton, which refers to exactly the same event occurrence. In this case, the time guard and clock resets of the event composition automaton transition are integrated into the synchronizing transition of the parallel composition. This is for example the case for the register! transitions from (noConvoy,unregistered) to (noConvoy,registered) in the parallelly composed timed automaton and from ec_initial to ec_registered in the event composition automaton, where the clock reset ec_c1 is added to the transition. Following the same concept, the time guard ec_c1 >= 2500 is added to the startConvoy! transition from (noConvoy,registered,ec_registered) to (convoy,registered,ec_registeredConvoy). Although not present in the example, location invariants can be added through conjunctions as it has been already described for state composition rules.

Furthermore, observe that the event composition automaton location, which is referred to by a composed location, only changes from the source to the target location of a transition, if the event composition au-

tomaton location has in fact an outgoing transition referring to the same event occurrence. This can be exemplified examining the breakConvoy? transition from location (convoy,registered,ec_registeredConvoy) to (noConvoy,registered,ec_registeredConvoy). Here, the event composition automaton location does not change as ec_registeredConvoy has no outgoing transition for the event breakConvoy?.

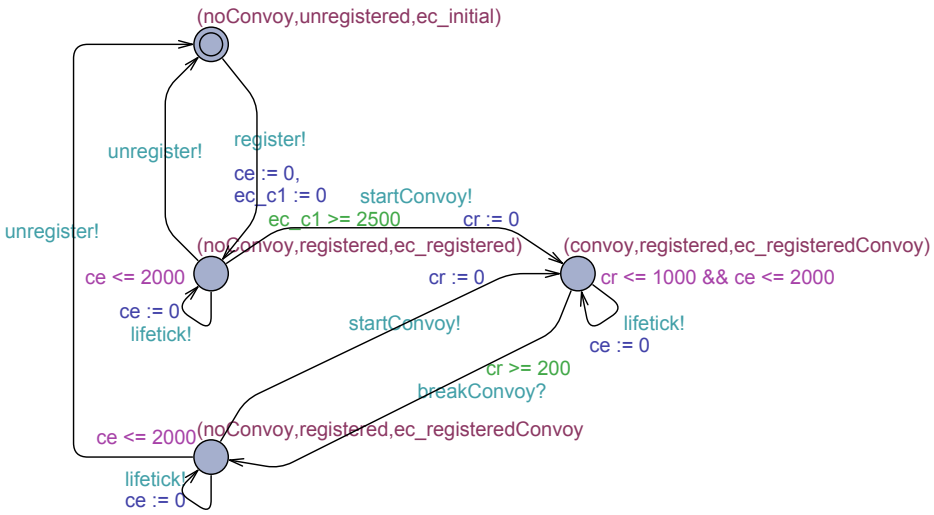

Figure 3.11: Event Composition Rule eca_2 Applied to the Timed Automaton Depicted in Figure 3.8

The informal description given above exemplifies the concept of adding time constraints to transitions and location invariants. This is done by means of silent synchronizations between the parallelly composed timed automaton and the event composition automaton. We proceed with the formal definition of the *event composition conform* timed automaton, which is the state composition conform, parallelly composed timed automaton to which an event composition automaton has been applied.

Definition 3.2.5 (Event Composition Conformance)
Let $A_{SC} = (L_{SC}, l^0_{SC}, \Sigma_{SC}, C_{SC}, I_{SC}, T_{SC})$ *be a state composition conform, parallelly composed timed automaton originating from the timed automata* $A_1 = (L_1, l^0_1, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l^0_2, \Sigma_2, C_2, I_2, T_2)$ *with* $C_1 \cap C_2 = \varnothing$ *and* $\Sigma_1 \cap \Sigma_2 = \varnothing$. *Furthermore, let* $A_E = (L_E, l^0_E, \Sigma_E, C_E, I_E, T_E) \in R^A(A_1, A_2)$ *be an event composition automaton for* A_1 *and* A_2. *We define the* event composition conform *and state composition conform, parallelly composed timed automaton* $A_{EC} = (L_{EC}, l^0_{EC}, \Sigma_{EC}, C_{EC}, I_{EC}, T_{EC})$ *with*

- $L_{EC} \subseteq L_1 \times L_2 \times L_E$, *with* $(l_1, l_2, l_e) \in L_{EC}$ *iff* $(l_1, l_2) \in L_{SC}$ *and* $I_{SC}((l_1, l_2)) \wedge I_E(l_e) \neq false$) *and* (l_1, l_2, l_e) *is reachable trough* T_{EC},
- $l^0_{EC} = (l^0_1, l^0_2, l^0_e)$, *iff* $(l^0_1, l^0_2, l^0_e) \in L_{EC}$,
- $\Sigma_{EC} = \Sigma_1 \cup \Sigma_2$,
- $I_{EC} : L_{EC} \rightarrow \Phi(C_1) \cup \Phi(C_2) \cup \Phi(C_E)$ *with* $I_{EC}((l_1, l_2, l_e)) = I_{SC}((l_1, l_2)) \wedge I_E(l_e)$,
- $C_{EC} = C_1 \cup C_2 \cup C_E$,
- $T_{EC} \subseteq L_{EC} \times \Sigma_{EC} \times \Phi(C_{EC}) \times 2^{C_{EC}} \times L_{EC}$, *with*
  - $((l_1, l_2, l_e), e_1, g_1, r_1, (l_1{}', l_2, l_e)) \in T_{EC} \Leftrightarrow$
    $((l_1, l_2), e_1, g_1, r_1, (l_1{}', l_2)) \in T_{SC} \wedge$
    $\forall\, l_e{}' \in L_E : (l_e, e_1, g_e, r_e, l_e{}') \notin T_E$,
  - $((l_1, l_2, l_e), e_2, g_2, r_2, (l_1, l_2{}', l_e)) \in T_{EC} \Leftrightarrow$
    $((l_1, l_2), e_2, g_2, r_2, (l_1, l_2{}')) \in T_{SC} \wedge$
    $\forall\, l_e{}' \in L_E : (l_e, e_2, g_e, r_e, l_e{}') \notin T_E$,
  - $((l_1, l_2, l_e), e_1, g_1 \wedge g_e, r_1 \cup r_e, (l_1{}', l_2, l_e{}')) \in T_{EC} \Leftrightarrow$
    $((l_1, l_2), e_1, g_1, r_1, (l_1{}', l_2)) \in T_{SC} \wedge (l_e, e_1, g_e, r_e, l_e{}') \in T_E$,
  - $((l_1, l_2, l_e), e_1, g_1 \wedge g_e, r_1 \cup r_e, (l_1, l_2{}', l_e{}')) \in T_{EC} \Leftrightarrow$
    $((l_1, l_2), e_2, g_2, r_2, (l_1, l_2{}') \in T_{SC} \wedge (l_e, e_2, g_e, r_e, l_e{}') \in T_E$.

The event composition conform automaton forms the explicit model for the parallel execution of the pattern role automata, while also respecting the specified state composition rules and event composition automaton.

The locations of the event composition conform automaton L_{EC} are a subset of the cross product $L_1 \times L_2 \times L_E$, including only those locations which are (1) state composition conform, (2) whose invariant $I_{EC}((l_1, l_2, l_e)) = I_{SC}((l_1, l_2)) \wedge I_E(l_e)$ is not equal to $false$, and (3) which is in fact reachable trough the transitions T_{EC} of this automaton.

Equal to the definition of state composition conformance (Definition 3.2.4), the invariant of each event composition conform location (l_1, l_2, l_e) is constructed by the conjunction of the invariant of the state composition conform location $I_{SC}(l_1, l_2)$ and the invariant of the event composition automaton location $I_E(l_e)$.

The definition of the set of transitions T_{EC} of A_{EC} differentiates between four cases. The first two cases describe the transitions which are not synchronized between A_{SC} and A_E. Here it is further distinguished that it is either a transition where A_1 changes its location or where A_2 changes it location. In both cases there is no corresponding transition in T_E which could be synchronized. The last two

cases describe the transitions which are synchronized between A_{SC} and A_E. Here, it is also further distinguished if A_1 changes its location or A_2.

Composed transitions which refer to a transition $t_e \in T_E$ of the event composition automaton but not to a transition $t_{sc} \in T_{SC}$ of the state composition conform automaton are not considered, as t_e can in this case not be synchronized with a transition of the state composition conform automaton. Note that these kind of transitions might actually exist, although the event composition automaton is defined to only listen to those events, which are either in A_1 or in A_2. Some event occurrences might be removed by the state composition conformance, though. This cannot be taken into account for the specification of event composition automata, as this takes place earlier than the state composition rule appliance from a developer's point of view.

For reasons of simplification, we refer to a state composition conform and event composition conform, parallelly composed timed automaton simply as *composition conform*.

Up to this point, we presented how composition rules by means of state composition rules and event composition automata are specified and automatically applied to the explicit model for the parallel execution of the pattern role automata, which is their parallel composition. The appliance of composition rules restricts the timing behavior of the parallel composition, by adding time constraints and removing state combinations. Consequently, it might happen that some relevant behavior is completely removed, if a remaining time interval equals zero, or if a single state of on role is completely removed. As a result, properties of the particular role automata, which have been verified in advance, are not necessarily preserved anymore after the appliance of those rules. How this can be detected is resented in the following section introducing the notion of *observational timed bisimulation* and *role conformance*.

3.3 Preserving Role Behavior

After composition rules have been applied to the parallelly composed timed automaton, it is not ensured anymore that the visible behavior of each of the particular role automata is still preserved. Assume for example, the appliance of an additional state composition rule $r_4 = \neg((registered, true) \wedge (convoy, cr > 100))$ to the composed timed automaton given in figure 3.11. This results in a new location invariant (cr<=100 && ce<=2000) for the location (convoy,registered,ec_registeredConvoy). As a consequence, the outgoing breakConvoy? transition can never be enabled, as its time guard cr>=200 can never evaluate to true. Accordingly, the relevant behavior of the convoy role is not anymore included in the composition conform automaton. Furthermore, note that this is

not always trivial to see when specifying composition rules, as some relevant behavior is removed not before two or more rules are applied. The rule r_4 applied on the original parallel composition (see Figure 3.7), for example, would not remove the executability of the breakConvoy? transition, as the automaton could switch to (unregistered,convoy) to execute the breakConvoy? transition.

In this section we deal with this problem by (1) defining a suitable equivalence relation which has to hold between the original parallel composition and the composition conform timed automaton (section 3.3.1) and (2) describing a procedure which checks if the equivalence relation holds (section 3.3.2) and further modifies the composition conform automaton in order to try to establish the equivalence relation if it does not hold (section 3.3.3).

3.3.1 Observational Timed Bisimulation

To discover violations against the original role behavior, we have to find a suitable equivalence relation which has to hold between the composition conform timed automaton and each of the role automata. As the behavior of the parallel composition of these role automata is equivalent to the behaviors of all role automata running in parallel (see section 3.2.1), we can also use this parallel composition for the equivalence check, instead of each single role automaton. Furthermore, we have to examine the semantical representations of both automata, such that we are able to make statements about the time intervals of the paths in both automata. Consequently, we have to define the equivalence relation on the basis of the timed transition systems (see Definition 2.1.9, Timed Automaton Semantics) of both automata.

In order to find a suitable equivalence relation, we first have to state which properties specified by U-TCTL formulas (see section 2.1.3) have to be preserved by the synthesis procedure. Obviously, preserving all types of U-TCTL formulas would lead to the fact, that we could hardly add any synchronization behavior in terms of composition rules. To exemplify this, observe that the U-TCTL formula $\exists\Diamond\,(convoy \wedge cr < 2500)$ exactly specifies those (timed) paths of the composition conform timed automaton in figure 3.11, which have been removed by the appliance of the composition rules r_1 and eca_2. If this formula should also evaluate to true after the synthesis procedure, we would not have been able to apply the given composition rules. In order to find the types of U-TCTL formulas, which specify relevant behavior, we assume the following characteristics of real-time systems, expressable by U-TCTL formulas. In the following, υ, υ_1 and υ_2 denote state properties of desired situations while $\hat{\upsilon}$ denotes a state property of an undesired situation:

1. *It may be the case that nothing happens:* $\exists\Box\ (\neg\upsilon \wedge \neg\hat{\upsilon})$.
   For the type of real-time systems that interact with an environment (also referred to as *reactive systems*), those systems only react to an occurrence of some event initiated in the environment. If no event occurs, it might happen that the system simply does not react, not in a good way and not in a bad way. The rear role automaton, for example, might forever stay in noConvoy, if no opportunity to build a convoy appears. Assuming that $convoy$ is a desired and $(processing \wedge c > 1000)$ is an undesired state property, this can be formalized to $\exists\Box\ (\neg convoy \wedge \neg(processing \wedge c > 1000))$.

2. *In may be the case that eventually something good happens:* $\exists\Diamond\ \upsilon$.
   This characteristic corresponds to the specification of a liveness property (see section 3.1). As for reactive systems it is possible that nothing happens (see 1.), it can only be specified, that something good will in some cases eventually happen. The rear role automaton for example will in some cases eventually be in state processing, where the value of clock c is equal to 500: $\exists\Diamond\ (processing \wedge c = 500)$.

3. *The occurrence of one good thing always leads to the occurrence of another good thing:* $\upsilon_1 \rightsquigarrow \upsilon_2$.
   This characteristic corresponds to the bounded response property (see section 2.1.3, p. 36), as a special type of a safety property. Observe that it does not state that one good thing (υ_1) happens at all (in accordance with 1.). If something good happens, however, it is always followed by another good thing (υ_2). The formula $processing \rightsquigarrow ((convoy \vee noConvoy) \wedge c \leq 1000)$, for example, specifies that once in state processing, the rear role automaton will always switch to state convoy or noConvoy within 1000 time units.

4. *It is always the case that something bad never happens:* $\forall\Box\ \hat{\upsilon}$.
   If something bad ($\hat{\upsilon}$) never happens in a system, this system is safe. Accordingly, this characteristic corresponds to the specification of a safety property. Such a safety property for the rear role automaton would be that it can never happen that the automaton is in state convoy, and the value of clock c is greater that 1000: $\forall\Box\ (processing \wedge c > 1000)$.

Taking all these characteristics of (reactive) real-time systems into account, it is straight forward to see that, dealing with safety critical systems, all types of formulas which specify safety properties are of vital importance. If any of these formulas is not preserved by the synthesis procedure, this might lead to high financial damage or even life threatening situations during the runtime of the system. The types of formulas defining safety properties are described in the

third and in the fourth characteristic and are further covered by the sub-language U-ATCTL of U-TCTL (see Definition 2.1.24, U-ATCTL Formula).

Unfortunately, a system that does not act in any way is a safe system, as it does not violate any U-ATCTL formula. The reason for this is that the path where the system does nothing, as described by the first characteristic, has to be included in any U-ATCTL formula. If this was not the case, every system would have to act in some way, even without a triggering event from the environment. To exemplify that preserving U-ATCTL is not sufficient in order to preserve the relevant behavior, assume a composition conform automaton to be the single location (noConvoy,unregistered). In this automaton, the system may only rest forever in this single location. As this is a valid path of the original parallelly composed timed automaton (Figure 3.7), all U-ATCTL formulas satisfied by the original automaton are also satisfied by this single location automaton. But obviously, the relevant behavior of both role automata is no longer included in this automaton. Consequently, preserving all properties which can be specified by U-ATCTL properties is not enough to preserve the relevant behavior.

As a result, we also have to guarantee that each timed automaton transition sequence is preserved in at least some of the possible time intervals specified by the original parallel composition of the role automata. This can be realized by additionally preserving all U-ECTL formulas (see Definition 2.1.27), which do not refer to any clock constraints. This way, it cannot only be preserved that if the rear role automaton is in processing it will always switch to state convoy or noConvoy within 1000 time units ($processing \rightsquigarrow ((convoy \vee noConvoy) \wedge c \leq 1000)$); it can also be preserved that the location processing is in fact reachable in the composition conform automaton ($\exists\Diamond\, processing$).

At this point, we additionally have to take into account that real-time coordination patterns specify communication protocols, where each role has another role as its counter part and a (buffered) communication channel inbetween. Consequently, we have to distinguish between sending and receiving events. *Sending events*, which means putting the signal onto a communication channel, can take place at any time within the original time interval. This is possible because the sender can assume that the recipient listens on the channel until the end of the complete time interval, as this is specified by the original role behavior. For *receiving events*, however, it has to be preserved that the recipient listens at least at the end of the original time interval on the channel. This is necessary, as the recipient does not know at which point in time, within the bounds of the original interval, the sender puts the event on the channel. If the recipient stops listening earlier, it might happen that a signal is lost in the channel and the communicating role automata are in an inconsistent state.

Having the above described requirements in mind, we study the most general timed equivalence relations in the following, in order to find a suitable equivalence

relation which fulfills our requirements. The first relation we examine is the *timed simulation* relation, as defined in the following (cf. [WL97]).

Definition 3.3.1 (Timed Simulation)
Given two timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$*, with* $C_1 \subseteq C_2$*,* $\Sigma_2 \subseteq \Sigma_1$*. Furthermore, let* $S_{A_1} = (Q_{A_1}, q_1^0, \Sigma_1, T_{\delta_1}, T_{\Sigma_1})$ *and* $S_{A_2} = (Q_{A_2}, q_2^0, \Sigma_2, T_{\delta_2}, T_{\Sigma_2})$ *be their corresponding timed transition systems.* A_1 *simulates* A_2*, denoted* $A_2 \leq A_1$ *iff a* timed simulation relation $\Omega \subseteq Q_{A_2} \times Q_{A_1}$ *exists, with* $(q_2^0, q_1^0) \in \Omega$ *and* $\forall\, ((l_2, \nu_2), (l_1, \nu_1)) \in \Omega$ *the following properties hold:*

1. $\forall\, ((l_2, \nu_2), e, (l_2', \nu_2')) \in T_{\Sigma_2}\ \exists\, ((l_1, \nu_1), e, (l_1', \nu_1')) \in T_{\Sigma_1} \wedge$
   $\forall c \in C_1 : \nu_2'(c) = \nu_1'(c) :$
   $((l_2', \nu_2'), (l_1', \nu_1')) \in \Omega,$

2. $\forall\, ((l_2, \nu_2), (l_2, \nu_2 + \delta)) \in T_{\delta_2}\ \exists\, ((l_1, \nu_1), (l_1, \nu_1 + \delta)) \in T_{\delta_1}, \delta \in R_+ :$
   $((l_2', \nu_2 + \delta), (l_1', \nu_1 + \delta)) \in \Omega.$

Additionally, the following property has to hold for A_2*:* $\forall\ s_2 \in Q_{A_2}\ (\exists\, (s_2, e, s_2') \in T_{\Sigma_2} \vee \exists\, (s_2, s_2') \in T_{\delta_2})$.

As the elemental property, the timed simulation equivalence requires both initial timed system states of S_{A_1} and S_{A_2} to be in the equivalence relation. On this basis, the following two properties are required to hold for all pairs $((l_2, \nu_2), (l_1, \nu_1))$ in the relation: (1) For all outgoing event transitions of (l_2, ν_2) in the concrete transition system S_{A_2} there exists an equivalent outgoing event transition of (l_1, ν_1) in the more abstract timed transition system S_{A_1} with the same event and the same clock valuations concerning the clocks of A_1, such that both target states again form a tuple in the relation Ω; (2) for all outgoing delay transitions of (l_2, ν_2) there also exists an outgoing delay transition in (l_1, ν_1) with the same delay δ, such that both target states are also a tuple in the relation. Furthermore, each state of S_{A_2} has to have either an outgoing delay or an outgoing event transition, which means that there is no time stopping deadlock in A_2.

Consequently, all transitions in the timed transition system of S_{A_2} have to be in S_{A_1}. Additionally, no time stopping deadlocks may be introduced and therefore the timed simulation preserves all U-ATCTL formulas.[2] Furthermore, observe that a timed simulation always exists between the original parallelly composed timed automaton and the composition conform version of this automaton

[2]For a proof for the untimed versions of simulation and ACTL, we refer to [CGP99, pp. 176–177]. This is directly transferable to timed simulation, as long as no time stopping deadlock is introduced. For the relation between CTL and TCTL model checking we refer to [TY01, pp. 55–56].

by construction, as long as there is no time stopping deadlock introduced by the appliance of the composition rules. The reason for this is that the composition rules can never add neither a delay transition nor an event transition to the underlying timed transition system, as this is explicitly restricted by the syntax and semantics of composition rules.

Unfortunately, we have already shown above that preserving all U-ATCTL formulas does not preserve all relevant behavior of the pattern role automata. Consequently, we examine a stricter equivalence relation in the following by means of the *timed bisimulation* equivalence (cf. [TY01]).

Definition 3.3.2 (Timed Bisimulation)
Given two timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$*, with* $C_1 \subseteq C_2$*,* $\Sigma_2 \subseteq \Sigma_1$*. Furthermore, let* $S_{A_1} = (Q_{A_1}, q_1^0, \Sigma_1, T_{\delta_1}, T_{\Sigma_1})$ *and* $S_{A_2} = (Q_{A_2}, q_2^0, \Sigma_2, T_{\delta_2}, T_{\Sigma_2})$ *be their corresponding timed transition systems.* A_1 *and* A_2 *are bisimulation equivalent, denoted* $A_2 \cong A_1$ *iff a* timed bisimulation relation $\Omega \subseteq Q_{A_2} \times Q_{A_1}$ *exists, with* $(q_2^0, q_1^0) \in \Omega$ *and* $\forall\, ((l_2, \nu_2), (l_1, \nu_1) \in \Omega$ *the following properties hold:*

1. $\forall\, ((l_2, \nu_2), e, (l_2', \nu_2')) \in T_{\Sigma_2}\ \exists\, ((l_1, \nu_1), e, (l_1', \nu_1')) \in T_{\Sigma_1} \wedge$
   $\forall c \in C_1 : \nu_2'(c) = \nu_1'(c) :$
   $((l_2', \nu_2'), (l_1', \nu_1')) \in \Omega,$

2. $\forall\, ((l_2, \nu_2), (l_2, \nu_2 + \delta)) \in T_{\delta_2}\ \exists\, ((l_1, \nu_1), (l_1, \nu_1 + \delta)) \in T_{\delta_1}, \delta \in R_+ :$
   $((l_2', \nu_2 + \delta), (l_1', \nu_1 + \delta)) \in \Omega,$

3. $\forall\, ((l_1, \nu_1), e, (l_1', \nu_1')) \in T_{\Sigma_1}\ \exists\, ((l_2, \nu_2), e, (l_2', \nu_2')) \in T_{\Sigma_2} \wedge$
   $\forall c \in C_1 : \nu_1'(c) = \nu_2'(c) :$
   $((l_2', \nu_2'), (l_1', \nu_1')) \in \Omega,$

4. $\forall\, ((l_1, \nu_1), (l_1, \nu_1 + \delta)) \in T_{\delta_1}\ \exists\, ((l_2, \nu_2), (l_2, \nu_2 + \delta)) \in T_{\delta_2}, \delta \in R_+ :$
   $((l_2', \nu_2 + \delta), (l_1', \nu_1 + \delta)) \in \Omega.$

Additionally, the following property has to hold for A_2*:* $\forall\ s_2 \in Q_{A_2}\ (\exists\, (s_2, e, s_2') \in T_{\Sigma_2} \vee \exists\, (s_2, s_2') \in T_{\delta_2})$.

The timed bisimulation is in such way stricter, that it requires the properties of simulation to hold in both directions. This means that for each outgoing transition of a state (l_2, ν_2) there has to exist an equivalent outgoing transition in (l_1, ν_1) and for each outgoing transition of (l_1, ν_1) there has to exist an outgoing transition in (l_2, ν_2). As a result, all paths (starting in the initial timed system states) that are in S_{A_1} are also in S_{A_2} and vice versa. Correspondingly, the timed bisimulation preserves all properties, which can be specified through U-TCTL formulas.[3]

[3] For a proof for the more general logic TCTL we refer to [TY01, pp. 53–56].

Unfortunately, preserving all paths of the parallel composition of the pattern role automata is too strict, as we would not be able to add any synchronization behavior. What we actually want to obtain is an equivalence relation between simulation and bisimulation. This implies that the needed equivalence relation completely includes the notion of simulation. For the other direction, we do not need to preserve that all paths of the more abstract automaton are included in the more concrete automaton, but only one path for each possible event. This way we can obtain an equivalence relation, which preserves U-ATCTL and U-ECTL as desired.

To achieve this, we weaken the bisimulation relation by applying a different transition relation, as for example applied in [TY01] for the *delay bisimulation*. Our new *transitive transition relation* realizes the following two concepts: (1) It abstracts from delay transitions inbetween two states and (2) it abstracts from internal behavior from each role automaton's perspective.

Abstracting from delay transitions inbetween two states is a well known concept to weaken a simulation relation and has been proposed more than once [WL97, TY01, Sei07]. Informally, this means that a transitive transition relation (s, e, s') exists between the two states s and s', if the event transition is an outgoing transition of one of the delay successors of s.

Abstracting from internal behavior has also been applied in the area of behavioral synthesis [GV06, Sei07]. Though, in other approaches, the internal behavior needs to be modeled explicitly in terms of so-called τ-transitions. In our approach we apply this concept as described in the following. From each role automaton's perspective, all events of other role automata can be seen as internal behavior of the component. Transferring this concept to the transitive transition relation, this means that a transitive transition (s, e, s') exists between the two states s and s', if the event transition is an outgoing transition of a state, which is reachable exclusively trough event transitions with events e', where all events e' are of a different automaton than the event e.

Summarizing all these concepts results in the definition of the *transitive transition relation* given in the following.

Definition 3.3.3 (Transitive Transition Relation (Timed Transition System)) *Given a timed transition system* $S_A = (Q_A, q^0, \Sigma, T_\delta, T_\Sigma)$ *of a parallelly composed timed automaton* $A = (L, l^0, \Sigma, C, I, T)$ *composed from timed automata* $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$. *The* transitive transition relation $T_\tau(S_A) \subseteq Q_A \times T_\Sigma \times Q_A$ *is defined with*

$$T_\tau(S_A) = \{(s, e, s') \mid (s, e, s') \in T_{\tau_1}(S_A) \vee (s, e, s') \in T_{\tau_2}(S_A)\}$$

where $T_{\tau_1}(S_A) \subseteq Q_A \times T_\Sigma \times Q_A$, $T_{\tau_2}(S_A) \subseteq Q_A \times T_\Sigma \times Q_A$ *are defined with*

$$
\begin{aligned}
T_{\tau_1}(S_A) &= \{(s, e_1, s') \mid \exists\, e_1 \in \Sigma_1 : \\
&\quad (s, e_1, s') \in T_\Theta \vee \\
&\quad (\exists\, e_2 \in \Sigma_2 : (s, e_2, s'') \in T_\Theta \wedge (s'', e_1, s') \in T_{\tau_1}) \vee \\
&\quad (\exists\, (s, s'') \in T_\delta \wedge (s'', e_1, s') \in T_{\tau_1})\} \\
T_{\tau_2}(S_A) &= \{(s, e_2, s') \mid \exists\, e_2 \in \Sigma_2 : \\
&\quad (s, e_2, s') \in T_\Theta \vee \\
&\quad (\exists\, e_1 \in \Sigma_1 : (s, e_1, s'') \in T_\Theta \wedge (s'', e_2, s') \in T_{\tau_2}) \vee \\
&\quad (\exists\, (s, s'') \in T_\delta \wedge (s'', e_2, s') \in T_{\tau_2})\}
\end{aligned}
$$

The transitive transition relation as defined above takes the parallel composition of two timed automata into account. Furthermore, it is defined on the basis of the timed transition system of the parallelly composed timed automaton. All transitions which are in the set of transitive transitions $T_\tau(S_A)$ are either in the set $T_{\tau_1}(S_A)$ or in the set $T_{\tau_2}(S_A)$ of transitive transitions. The set $T_{\tau_1}(S_A)$ contains all transitive transitions where events are exclusively of A_1. Accordingly, the set is constructed by allowing delay transitions and transitions with events of A_2 inbetween the source and the target state. The set $T_{\tau_1}(S_A)$ is constructed analogously for the events of A_2.

As an example, we examine the outgoing startConvoy! transition of the location (noConvoy,unregistered) in the original parallel composition of the rear role and the registree role automaton (Figure 3.7). Observe that it is not present anymore in the composition conform version of this automaton (Figure 3.11). Instead, the composition conform automaton has to perform register! and at least one lifetick!, before a startConvoy! can be executed. Consequently, there is no outgoing startConvoy! event transition in the state $((noConvoy, unregistered, ec\_initial), cr = ce = ec\_c1 = 0)$ of the corresponding timed transition system. There is an outgoing transitive transition for the event startConvoy!, however, leading to $((convoy, registered, ec\_initial), cr = ec\_c1 = 2500 \wedge ce = 500)$, as all transitions inbetween are either delay transition or event transitions with events from the registree role automaton.

Summarizing this, the transitive transition relation realizes that, for a given timed automaton transition t of the parallel composition, the more concrete automaton can now either rest some time in a certain location or execute events of another automaton first, before executing the transition t. It does not consider, however, that an event might have been executed already at an earlier point in time, as required for sending events. This possibility is realized solely by the equivalence relation which is defined below. This equivalence relation, which we call *observational timed bisimulation*, realizes exactly all the above stated concepts as

(1) it completely includes timed simulation, (2) it applies the transitive transition relation for the direction from the more abstract to the concrete automaton and (3) it also distinguishes between sending and receiving events for this direction.

Definition 3.3.4 (Observational Timed Bisimulation)
Given two timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$*, with* $C_1 \subseteq C_2$*,* $\Sigma_2 \subseteq \Sigma_1$*. Furthermore, let* $S_{A_1} = (Q_{A_1}, q_1^0, \Sigma_1, T_{\delta_1}, T_{\Sigma_1})$ *and* $S_{A_2} = (Q_{A_2}, q_2^0, \Sigma_2, T_{\delta_2}, T_{\Sigma_2})$ *be their corresponding timed transition systems.* A_1 *and* A_2 *are observational bisimulation equivalent, denoted* $A_2 \preceq A_1$ *iff an* observational timed bisimulation *relation* $\Omega \subseteq Q_{A_2} \times Q_{A_1}$ *exists, with* $(q_2^0, q_1^0) \in \Omega$ *and* $\forall\ ((l_2, \nu_2), (l_1, \nu_1)) \in \Omega$ *the following properties hold:*

1. $\forall\ ((l_2, \nu_2), e, (l_2', \nu_2')) \in T_{\Sigma_2}\ \exists\ ((l_1, \nu_1), e, (l_1', \nu_1')) \in T_{\Sigma_1} \wedge$
   $\forall c \in C_1 : \nu_2'(c) = \nu_1'(c) :$
   $((l_2', \nu_2'), (l_1', \nu_1')) \in \Omega,$

2. $\forall\ ((l_2, \nu_2), (l_2, \nu_2 + \delta)) \in T_{\delta_2}\ \exists\ ((l_1, \nu_1), (l_1, \nu_1 + \delta)) \in T_{\delta_1}, \delta \in R_+ :$
   $((l_2', \nu_2 + \delta), (l_1', \nu_1 + \delta)) \in \Omega,$

3. $\forall\ ((l_1, \nu_1), e?, (l_1', \nu_1')) \in T_{\Sigma_1}\ \exists\ ((l_2, \nu_2), e?, (l_2', \nu_2')) \in T_\tau(S_{A_2}) \wedge$
   $((\nu_1 \neq \nu_1') \Rightarrow (\forall c \in C_1, \nu_1'(c) = 0 : \nu_2'(c) = 0)) :$
   $((l_1', \nu_1'), (l_2', \nu_2')) \in \Omega,$

4. $\forall\ ((l_1, \nu_1), e!, (l_1', \nu_1')) \in T_{\Sigma_1}\ (\ \exists\ ((l_2, \nu_2), e!, (l_2', \nu_2')) \in T_\tau(S_{A_2}) \vee$
   $\exists\ ((l_1, \nu_{1-}), e!, (l_1', \nu_{1-}')) \in T_{\Sigma_1}\ \exists\ ((l_2'', \nu_2''), e!, (l_2', \nu_2')) \in T_\tau(S_{A_2}),$
   $\forall c \in C_1 : \nu_{1-}(c) < \nu_1(c)\ ) \wedge$
   $((\nu_1 \neq \nu_1') \Rightarrow (\forall c \in C_1, \nu_1'(c) = 0 : \nu_2'(c) = 0)) :$
   $((l_2', \nu_2'), (l_1', \nu_1')) \in \Omega$

Additionally, the following property has to hold for A_2*:* $\forall\ s_2 \in Q_{A_2}\ (\exists\ (s_2, e, s_2') \in T_{\Sigma_2} \vee \exists\ (s_2, s_2') \in T_{\delta_2})$.

The first two properties for states $((l_2, \nu_2), (l_1, \nu_1))$ of the relation correspond to the definition of timed simulation.

The third property realizes that for each event transition of the more abstract automaton with a receiving event, there exists a transitive event transition in the more concrete automaton. This implies that there also has to be a corresponding receiving event transition in the concrete system for the last possible point in time. If this was not the case, this property would be violated. To exemplify this, assume a transition $t_1 = ((l_1, \nu_1), e?, (l_1', \nu_1')) \in T_{\Sigma_1}$ where the clock valuation ν_1 represents the last possible point in time. For this transition t_1 there also has to exist a corresponding transition t_2 in $T_\tau(S_{A_2})$, and as all delay transitions of S_{A_2} also

have to be in S_{A_1} (due to the second property), the transition t_2 has to be provided at the same point in time. Consequently, with this property we guarantee that no event is lost in a communication channel for the reason that A_2 stops listening to the channel too early.

The fourth property is defined similarly, but additionally it is considered that there might already exist an event transition in S_{A_1} earlier in time, for which there exists a transitive transition in S_{A_2}. In this case, the target states of the earlier transitive transition $((l_2'', \nu_2''), e!, (l_2', \nu_2'))$ of S_{A_2} has to be in the simulation relation together with (l_1', ν_1').

Furthermore, observe that for the third and the fourth property, the clock resets are also regarded by comparing the clock valuations ν_1 and ν_1' of the transition $((l_1, \nu_1), e, (l_1', \nu_1'))$ with each other. If they differ, all the clocks which are equal to zero in ν_1' also have to be equal to zero in ν_2' in $((l_2, \nu_2), e, (l_2', \nu_2')) \in T_\tau(S_{A_2})$, to ensure that no clock reset is lost in the concrete time automaton.

With the observational timed bisimulation defined above, we have a suitable equivalence relation, which, if established, preserves the relevant behavior in terms of U-ATCTL and U-ECTL formulas. This equivalence relation is not inherently established after the appliance of composition rules, however, as we have already shown above. What we have also shown, however, is that the simulation part of the relation is established by construction, as long as no time stopping deadlock is introduced. In the remainder of this section, we show how the zone automaton can be utilized to verify that for all paths of the original parallel composition, there exists at least one untimed, observational path in the composition conform automaton. By observational we mean that this path takes into account that the behavior of other roles can be regarded as internal component behavior. This way, we achieve that all U-ECTL formulas, which have been valid for each single role automaton, are still valid on the parallelly composed, composition conform automaton. We call this property *role conformance*. Note that a role conform, composition conform timed automaton is not necessarily also observational timed bisimilar to its original parallel composition, as there might exist time stopping deadlocks in the role conform automaton. This has to be checked additionally.

3.3.2 Role Conformance

As timed automata semantics generally implies an infinite state space (see section 2.1.1), we need an adequate discrete abstraction of the timed automaton model in order to perform any formal analysis on it. The *zone automaton* representation of Alur (see section 2.1.2.3) is currently the best available approach to this problem, as it combines an efficient method to reduce the state space while preserving all relevant timing properties. In addition to that, Alur's approach exactly fits our

needs as it abstracts from the actual time interval where a timed automaton transition is enabled, but instead, only models if a transition is enabled at all starting in a certain zone location.

The procedure of checking for role conformance is divided itself into two different steps (Figure 3.12): (1) The zone automaton for the state and event composition conform product automaton is constructed; (2) the language of this zone automaton is checked against the language of the parallel composition of the single role automata. Both procedures will be described in detail in the following two sections.

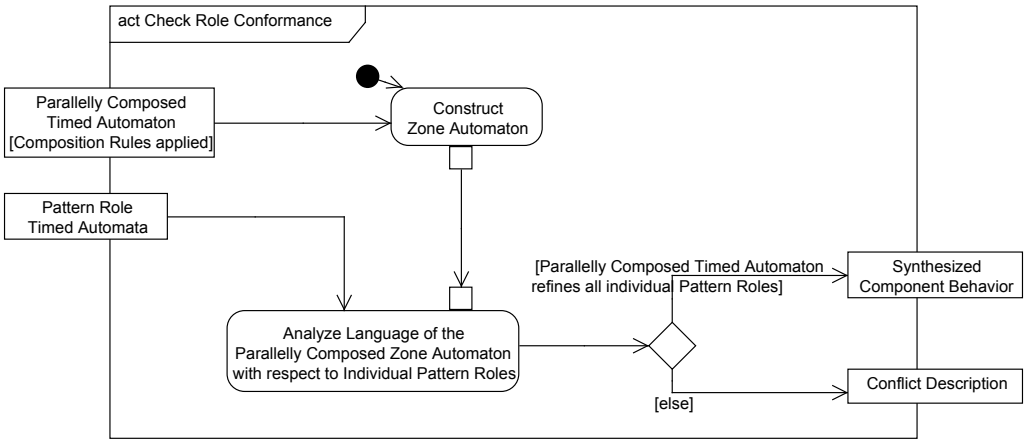

Figure 3.12: Algorithm to Verify Role Conformance

Before we formally define the procedure of verifying role conformance, we exemplify it by using the composition conform timed automaton constructed in the preceding sections. The first step in the procedure is to construct the zone automaton, in order to obtain a model of all reachable transitions with respect to their timing behavior. An extract of the zone automaton constructed from the composition conform automaton of the rear role, the registree role and composition rules $r1$ and eca_2 (Figure 3.11) is depicted in figure 3.13. We only show an extract of this zone automaton as the complete version has 186 zone locations and 396 transitions and therefore cannot be utilized to illustrate the procedure. The extract contains the initial zone location ((noConvoy,unregistered,ec_initial),cr==ce & ce==ec_c1 & ec_c1==0) as well as 1 direct and 9 indirect successors. The path register!, lifetick!, lifetick!, startConvoy!, lifetick!, breakConvoy?, unregister!, for example, represents a cycle visiting each location of the automaton at least once.[4]

[4]We refrain from describing the complete extract, as it can be understood by following the possible timed paths of the original timed automaton (Figure 3.11).

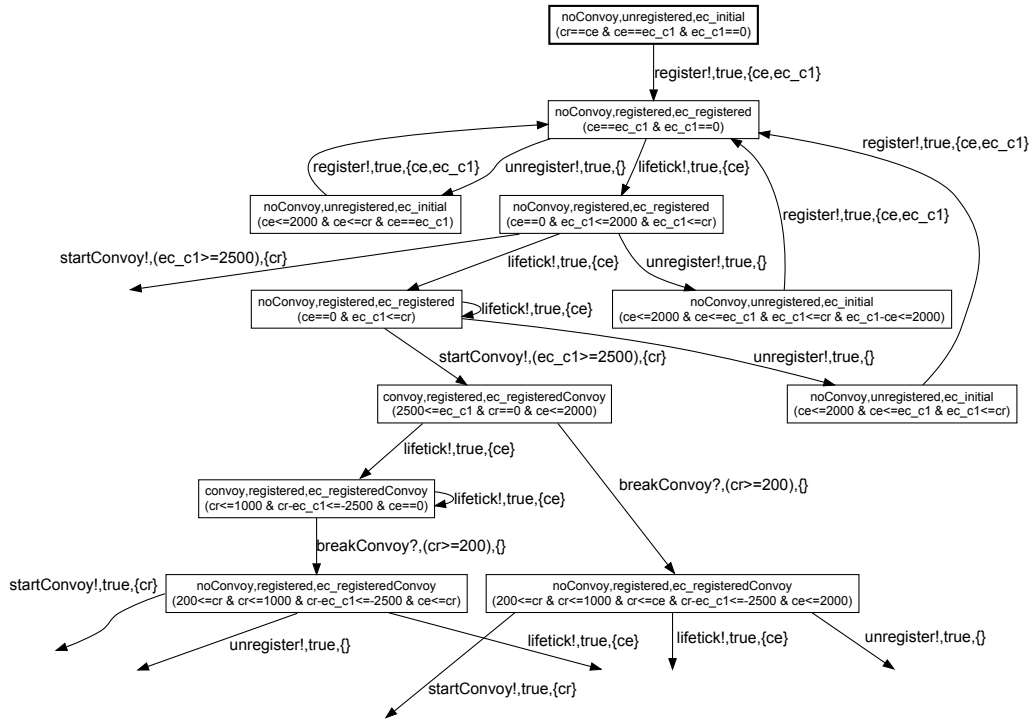

Figure 3.13: Extract of the Zone Automaton of the Composition Conform Timed Automaton of the Convoy Example (Figure 3.11)

In a role conform, composition conform timed automaton, each path of the original parallel composition of the role automata is still present with respect to time delay and internal behavior. To find out if this is the case in a given zone automaton, we have to check if each zone location still *offers* all events of the corresponding timed automaton location. For the zone location ((noConvoy,unregistered,ec_initial),cr==ce & ce==ec_c1 & ec_c1==0), for example, we have to check if it offers a register! and a startConvoy! event, as those are the events of the outgoing transitions of the location (noConvoy,unregistered) in the original parallel composition (see Figure 3.11). The notion of *offering behavior* has to take the concept of the transitive transition relation into account (see Definition 3.3.3, Transitive Transition Relation (Timed Transition System)). While transitive delay transitions are inherently considered by the construction of the zone automaton (see section 2.1.2.3), transitive transitions regarding behavior of other roles (internal behavior) have to be considered explicitly. This means, that for an event of the rear role automaton, there can be arbitrarily many event transitions of the registree role automaton inbetween.

To exemplify this procedure, we examine the root zone location ((noConvoy,unregistered,ec_initial),cr==ce & ce==ec_c1 & ec_c1==0) if it offers the events register! and startConvoy!. As register! is directly offered by the zone location, we only have to search for a startConvoy! event. For this, we are only allowed to follow transitions annotated with events of the registree role automaton, as startConvoy! is an event of the rear role automaton. This transition can be found even twice, once on the path register!, lifetick!, startConvoy! and once on the path register!, lifetick!, lifetick!, ..., startConvoy!. Consequently, the root zone location offers the same events as the corresponding timed automaton location; we call such a location *consistent*.

The above described is sufficient for sending events, but not for receiving events. For those, we additionally have to check if the latest point in time of the original transition is preserved by this location. We exemplify this by examining the outgoing breakConvoy? transition of the zone location ((convoy,registered,ec_registeredConvoy), 2500<=ec_c1 & cr==0 & ce<=2000). The time interval in which the breakConvoy? transition is enabled is calculated by (cf. section 2.1.2.3): (1) letting time elapse on the zone of the source zone location, (2) intersecting the resulting zone of step 1 with the invariant of the corresponding (composition conform) timed automaton location and (3) intersecting the resulting zone of step 2 with the time guard of the (composition conform) timed automaton transition.[5] This time interval has to be compared to the time interval it should have in order to be enabled to be enabled regarding the location invari-

[5]The procedure is actually very similar to computing the successor zone, as only clock resets are not applied.

ant and the time guard of the original parallelly composed timed automaton. The original time interval is computed the same way, but using the original location invariant and time guard. When comparing both of these time intervals, it must be considered that the composition conform time interval is allowed to start later in time. This means that only the upper bounds have to be compared. Consequently, we apply a so-called *weakest delay precondition operation* (or simply *down operation*) on both intervals, which basically removes all lower bounds. Now, the clock zones should be equal. To test this, we subtract the composition conform zone from the original zone. If the result is an empty zone, both intervals are equal and the regarded transition of the zone automaton represents the offered behavior of the breakConvoy? event. If the result is a non-empty zone, we have to search for a different transition, following transitions with events of the registree role automaton, as described above. If no such transition can be found, this zone location is labeled as inconsistent. For the example we omit this calculation as neither the location invariant of (convoy,registered,ec_registeredConvoy) differs from the location invariant of (convoy,registered) nor was the time guard of the breakConvoy? transition changed by any of the composition rules.

The overall procedure checks every zone location of the zone automaton for consistency, while removing the inconsistent locations and corresponding incoming and outgoing transitions. After any of the zone locations have been removed, all other zone locations have to be checked again, as their offered behavior might have changed due to the removal of transitions. If at any time all zone locations are consistent and the initial zone location has not been removed, the obtained timed automaton is role conform. Note that due to the removal of zone locations the role conform timed automaton does not correspond to the zone automaton anymore, as some timed paths which lead into deadlocks have been removed. This can be fixed by adding further time guards to transitions. This procedure is described in detail in the next section (section 3.3.3).

We proceed with the formalization of the procedure. As we have described above, we have to distinguish between offering of sending events and offering of receiving events, as the receiving events have to take the upper bounds of the original time interval into account. Furthermore, we distinguish between the offered behavior of a timed automaton location of the parallel composition of the role automata and the offered behavior of a zone location of the zone automaton of the composition conform timed automaton. We start with the definitions for the offered behavior of a timed automaton location, which computes the offered behavior for a timed automaton location, starting in a given zone.

Definition 3.3.5 (Offered Sending Behavior (Timed Automaton Location)) *For a timed automaton* $A = (L, l^0, \Sigma, C, I, T)$ *the* offered behavior *of an automaton location* $l \in L$ *and a clock zone* $\vartheta \in \Psi(C)$ *is defined by the function*

$offers : L \times \Psi(C) \rightarrow 2^{\Sigma}$ *with*

$$offers_!(l, \vartheta) = \{e! | \exists\ (l, e!, g, r, l') \in T : (\vartheta^{\Uparrow} \wedge I(l) \wedge g) \neq false)\}.$$

The *offered sending behavior* for a given timed automaton location l and a clock zone ϑ is a set of events which contains all the sending events, which are reachable from location l starting in the zone ϑ. Those are basically all events annotated at outgoing transitions of the location l. But additionally it has to be checked, if the transition is enabled at all, starting in the given zone. Consequently, we have to (1) let time elapse on the input zone ϑ, (2) intersect it with the invariant of the location l and (3) intersect it with the guard of the transition. This way, we find out whether or not the corresponding event of an outgoing transition is offered for the given zone ϑ.

To exemplify this, we examine the location ((convoy,unregistered),cr<=50) of the state composition conform automaton depicted in figure 3.9. Furthermore, we assume that the outgoing breakConvoy? transition was labeled with a breakConvoy! instead. The input zone is given with $(cr = 0 \wedge ce \geq 0)$. The computation to find out if the breakConvoy! event is offered is exemplified in the following:

$$\begin{aligned}
&(cr = 0 \wedge ce \geq 0)^{\Uparrow} \wedge (cr \leq 50) \wedge (cr \geq 200) \\
&= (cr \leq ce) \wedge (cr \leq 50) \wedge (cr \geq 200) \\
&= (cr \leq ce \wedge cr \leq 50) \wedge (cr \geq 200) \\
&= false.
\end{aligned}$$

As the result is $false$, the only offered event of the location ((convoy,unregistered),cr<=50) would be register!, i.e.:

$$offers_!((convoy, unregistered), cr \leq 50), (cr = 0 \wedge ce \geq 0)) = \{register!\}.$$

We proceed with the definition of the offered receiving behavior of a timed automaton location. As we have already described above, the offered receiving behavior has to take the upper bounds of the original interval into account, while it has to abstract from lower bounds. Consequently, we need an operation on clock zones, which removes the lower bounds, similar to the time elapse operation (see Definition 2.1.13), which removes the upper bounds. We call this the *weakest delay precondition* or *down operation* (cf. [BY03, p. 106]) as defined in the following.

Definition 3.3.6 (Weakest Delay Precondition on Clock Zones)
For a clock zone $\vartheta \in \Theta(C)$ the weakest delay precondition *(also referred to as* down operation*) denoted $\vartheta^{\Downarrow}$ is defined with*

$$\begin{aligned}
\vartheta^{\Downarrow} = \{\nu \quad | \quad & \forall\ \nu' \in \vartheta, \forall\ c \in C, \forall\ \delta \in \mathbb{R}_+ : \\
& (\forall\ c \in C : \delta \leq \nu'(c)) \Rightarrow \nu(c) = \nu'(c) - \delta\}.
\end{aligned}$$

The *weakest delay precondition* or *down operation* on a clock zone basically removes all lower bounds in that zone. However, constraints on clock differences also have to be considered. Consequently, all possible values δ are subtracted from each valuation of the clock zone, as long as no valuation becomes negative. This means that for a certain clock valuation ν' all clocks' values have to be greater or equal to zero after the subtraction of $delta$. The down operation is applied when calculating the *offered receiving behavior* as follows.

Definition 3.3.7 (Offered Receiving Behavior (Timed Automaton Location)) *For a timed automaton* $A = (L, l^0, \Sigma, C, I, T)$ *the* offered behavior *of an automaton location* $l \in L$ *and a clock zone* $\vartheta \in \Psi(C)$ *is defined by the function* $offers : L \times \Psi(C) \rightarrow 2^{\Sigma}$ *with*

$$offers_?(l, \vartheta) = \{(e?, \vartheta_e) \quad | \quad \exists\, (l, e?, g, r, l') \in T, \vartheta_e = (\vartheta^{\Uparrow} \wedge I(l) \wedge g)^{\Downarrow} : \vartheta_e \neq false)\}.$$

In contrast to the offered sending behavior, the *offered receiving behavior* is a set of pairs of an event $e?$ and a clock zone ϑ_e. Again, the input for the $offers_?$ function is a timed automaton location l and a clock zone ϑ. The procedure to calculate if the corresponding transition of the event is enabled at all is equivalent to the one of offered sending behavior. Though in this procedure, the resulting time interval is used to represent the zone, where the event $e?$ is enabled. Therefore, the lower bounds are removed by applying the down operation. This has already been performed at this point, as the lower bounds are not relevant for the later comparison with the interval of the corresponding event of the composition conform automaton.

As an example, we examine the location ((convoy,registered),cr<=1000 && ce <=2000) and its outgoing breakConvoy? transition of the automaton in figure 3.9. The calculation of the corresponding clock zone ϑ_e is exemplified in the following, where the input zone is $(cr \leq 50 \wedge ce = 0)$:

$$\begin{aligned}
&\vartheta_{breakConvoy?} \\
&= ((cr \leq 50 \wedge ce = 0)^{\Uparrow} \wedge (cr \leq 1000 \wedge ce \leq 2000) \wedge (cr \geq 200))^{\Downarrow} \\
&= ((cr - ce \leq 50 \wedge ce \leq cr) \wedge (cr \leq 1000 \wedge ce \leq 2000) \wedge (cr \geq 200))^{\Downarrow} \\
&= ((cr - ce \leq 50 \wedge ce \leq cr \wedge cr \leq 1000) \wedge (cr \geq 200))^{\Downarrow} \\
&= (cr - ce \leq 50 \wedge ce \leq cr \wedge cr \leq 1000 \wedge cr \geq 200)^{\Downarrow} \\
&= cr - ce \leq 50 \wedge ce \leq cr \wedge cr \leq 1000.
\end{aligned}$$

Observe the removal of the lower bound $ce \geq 200$ by the appliance of the down operation. As has already been described above, this is removed to make the zone comparable only regarding the upper bounds and the bounds on clock differences.

Above, we have defined the computation of the offered behavior of a timed automaton location with respect to a given clock zone. In the following we define the offered behavior of a zone location originating from a composition conform, parallelly composed timed automaton. As we have already defined for the observational timed bisimulation (Definition 3.3.4), we have to take into account that for a certain event, events of other roles can be seen as internal behavior of the corresponding component. Consequently, we have to define a *transitive transition relation* also for the zone automaton, in order to compute the offered behavior of a zone location.

Definition 3.3.8 (Transitive Transition Relation (Zone Automaton))
For a zone automaton $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ *constructed from a composition conform timed automaton* $A = (L, l^0, \Sigma, C, I, T)$ *which again is composed from timed automata* $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ *the set of* transitive transitions T_τ *is defined with*

$$T_\tau = \{((s, e, s'), s_{pre}) \mid ((s, e, s'), s_{pre}) \in T_{\tau_1} \vee ((s, e, s'), s_{pre}) \in T_{\tau_2}\}$$

where

$$\begin{aligned}
T_{\tau_1} = \{((s, e_1, s'), s_{pre}) \quad | \quad & \exists\, e_1 \in \Sigma_1 : (s, e_1, s') \in T_\Theta, s_{pre} = s \vee \\
& (\exists\, e_2 \in \Sigma_2 : (s, e_2, s'') \in T_\Theta, s_{pre} = s'' \wedge \\
& \exists\, e_1 \in \Sigma_1 : ((s'', e_1, s'), s_{pre}) \in T_{\tau_1})\} \\
T_{\tau_2} = \{((s, e_2, s'), s_{pre}) \quad | \quad & \exists\, e_2 \in \Sigma_2 : (s, e_2, s') \in T_\Theta, s_{pre} = s \vee \\
& (\exists\, e_1 \in \Sigma_1 : (s, e_1, s'') \in T_\Theta, s_{pre} = s'' \wedge \\
& \exists\, e_2 \in \Sigma_2 : ((s'', e_2, s'), s_{pre}) \in T_{\tau_2})\}.
\end{aligned}$$

The *transitive transition relation* for a zone automaton differs from the transitive transition relation for a timed transition system (Definition 3.3.3) in two main points: (1) The delay transitivity is not considered explicitly, as it is implicitly considered by the construction of the zone automaton (see section 2.1.2.3) and (2) the zone location s_{pre}, which directly offers a transition with the corresponding event, is annotated at each transitive transition, as it is needed for the comparison of the time intervals of the receiving event. We call this zone location *last predecessor* of a transitive transition relation. All other parts of the definition are equal to the definition for the timed transition system, except that this definition is defined on the basis of a zone automaton.

We exemplify the transitive transition relation by examining the zone location ((noConvoy,unregistered,ec_initial),cr==ce & ce==ec_c1 & ec_c1==0) of the zone automaton extract of figure 3.13. Assume that $T_{\tau - s_1 - registree}$ represents the set of transitive transitions for the registree role automaton and that $T_{\tau - s_1 - rear}$ represents those for the rear role timed automaton, while only the outgoing transitive transitions of the root zone location are regarded.

As the first and only outgoing transition of the initial zone location is annotated with register!, the only (transitive) transition regarding the registree role events is this register! transition, i.e. that

$$T_{\tau-s_1-registree} = \{((s_1, register!, s_2), s_1)\},$$

with

$$\begin{aligned} s_1 &= ((noConvoy, unregistered, ec\_initial), cr = ce = ec\_c1 = 0) \text{ and} \\ s_2 &= ((noConvoy, registered, ec\_registered), ce = ec\_c1 = 0). \end{aligned}$$

For the events of the rear role automaton, we have to look for the startConvoy! transitions following the paths register!, lifetick!, startConvoy! and register!, lifetick!, lifetick!, startConvoy!. Consequently, the set $T_{\tau-rear}$ contains two transitive transition tuples, i.e. that

$$T_{\tau-s_1-rear} = \{((s_1, startConvoy!, s_5), s_3), ((s_1, startConvoy!, s_6), s_4)\}$$

where

$$\begin{aligned} s_3 &= ((noConvoy, registered, ec\_registered), \\ &\quad ce = 0 \wedge ec\_c1 \leq 2000 \wedge ec\_c1 \leq cr), \\ s_4 &= ((noConvoy, registered, ec\_registered), ce = 0 \wedge ec\_c1 \leq cr) \text{ and} \\ s_6 &= ((convoy, registered, ec\_registeredConvoy), \\ &\quad ec\_c1 \geq 2500 \wedge cr = 0 \wedge ce \leq 2000). \end{aligned}$$

and s_5 is not depicted. Accordingly, the set of all outgoing transitive transitions $T_{\tau-s_1}$ of s_1 is the union of the sets $T_{\tau-s_1-rear}$ and $T_{\tau-s_1-registree}$, i.e. that $T_{\tau-s_1} = T_{\tau-s_1-registree} \cup T_{\tau-s_1-rear}$.

The transitive transition relation is applied in the calculation of the offered behavior of a zone location. This offered behavior is defined in the following, starting with the definition of the *offered sending behavior*.

Definition 3.3.9 (Offered Sending Behavior (Zone Location))
For a zone automaton $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ constructed from a composition conform timed automaton $A = (L, l^0, \Sigma, C, I, T)$ which again is composed from timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ and $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ the offered behavior *of an zone location (l, ϑ) is defined by the function $offers$: $L \times \Psi(C) \rightarrow 2^\Sigma$ with*

$$offers_!(s) = \{e! \mid ((s, e!, s'), s_{pre}) \in T_\tau\}.$$

The *offered sending behavior* of a zone location returns the set of sending events offered by the input zone location s. Those events are all the events which are reachable trough transitive transitions starting in s. As for sending events, the time interval in which the event is enabled is not of relevance, the zone location s_{pre} of each transitive transition tuple is simply not further regarded in this definition. The set of offered sending events for the initial zone location ((noConvoy,unregistered,ec_initial),cr==ce & ce==ec_c1 & ec_c1==0) of the example zone automaton in figure 3.13 has been computed in the previous example of the transitive transition relation and accordingly is $\{register!, startConvoy!\}$. We proceed with the definition of *offered receiving behavior* of a zone location.

Definition 3.3.10 (Offered Receiving Behavior (Zone Location))
For a zone automaton $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ *constructed from a composition conform timed automaton* $A = (L, l^0, \Sigma, C, I, T)$ *which again is composed from timed automata* $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ *the* offered behavior *of an zone location* (l, ϑ) *is defined by the function* $offers : L \times \Psi(C) \rightarrow 2^\Sigma$ *with*

$$offers_?(s) = \{(e?, \vartheta_e) \mid \exists\,((s, e?, s'), (l, \vartheta)) \in T_\tau, \exists\,(l, e?, g, r, l') \in T, \vartheta_e = (\vartheta^{\Uparrow} \wedge I(l) \wedge g)^{\Downarrow} : \vartheta_e \neq false\}.$$

Similar to the offered receiving behavior of a timed automaton location (Definition 3.3.7), the offered receiving behavior of a zone location consists of tuples of receiving events $e?$ and corresponding clock zones ϑ_e. These clock zones again represent the time interval in which the event is offered. For this time interval, the last predecessor of the transitive transition relation is taken into account. This is necessary, as the clock zone possibly changes along the transitions and locations inbetween. Therefore, the clock zone right before the actual transition offering $e?$ is needed to compute the actual time interval for the event. The time interval is computed as already described for the offered receiving behavior of a timed automaton location (see Definition 3.3.7). Note also that the location invariant of the last predecessor and the time guard of the corresponding outgoing transition is used, as those define the correct upper bounds for the time interval. Again, the down operation is applied to remove the lower bounds. We omit the exemplification, as it is already described in detail at the beginning of this section.

To verify that a zone location originating from the composition conform timed automaton is consistent, we have to analyze if the set of offered events is equal to the set of offered events of the original parallel composition of the role automata. In detail, this includes to analyze the offered sending and receiving events. For the sending events, it is sufficient to solely verify that the sets of reachable events are equal. For receiving events, the time intervals in which the corresponding transitions are enabled additionally have to be compared. As the lower bounds

have been removed from both of these time intervals, the comparison is performed by subtracting one clock zone from the other. This operation of *subtraction* on clock zones is defined in the following.

Definition 3.3.11 (Subtraction on Clock Zones)
For two clock zones $\vartheta_1, \vartheta_2 \in \Theta(C)$ *the* subtraction $\vartheta_2 - \vartheta_1$ *is defined with*

$$\vartheta_2 - \vartheta_1 = \{\nu \quad | \quad \nu \in \vartheta_2 \wedge \nu \notin \vartheta_1\}.$$

The *subtraction* on clock zones is defined equally to subtraction on sets, as a clock zone is nothing else than a set of clock valuations (see Definition 2.1.10, p. 25). Accordingly, when subtracting a clock zone ϑ_1 from a clock zone ϑ_2, we simply remove all clock valuations which are in ϑ_1 and ϑ_2 from ϑ_2 and return the modified ϑ_2. If the resulting clock zone is empty, ϑ_1 and ϑ_2 are equal. This property is taken advantage from, when comparing two clock zones for equality, in order to find out if one transition offers a receiving event until the end of the complete original time interval of the role automaton.

We proceed with the final definition of *role conformance* on the basis of a composition conform timed automaton and the original parallel composition of the role automata.

Definition 3.3.12 (Role Conformance)
Let $A = (L, l^0, \Sigma, C, I, T)$ *be a composition conform, parallelly composed timed automaton originating from timed automata* $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ *and* $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ *and the event composition automaton* $A_{EC} = (L_{EC}, l_{EC}^0, \Sigma_{EC}, C_{EC}, I_{EC}, T_{EC})$. *Further let* $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ *be the corresponding zone automaton and let* $A_P = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$ *be the parallel composition* $A_1 \parallel A_2$. *We define* A *to be* role conform *iff*

$$\exists\, Z'_A = (S'_\Theta \subseteq S_\Theta, s^0, \Sigma, C, T'_\Theta \subseteq T_\Theta)$$

and

$$\begin{aligned}
&\forall\, ((l_1, l_2, l_e), \vartheta) \in S'_\Theta : \\
&offers_!(((l_1, l_2, l_e), \vartheta)) = offers_!((l_1, l_2), \vartheta) \wedge \\
&offers_?(((l_1, l_2, l_e), \vartheta)) \supseteq offers_?((l_1, l_2), \vartheta),
\end{aligned}$$

where $(l_1, l_2) \in L_P$ *and*
$(e?, \vartheta_e) \in offers_?(((l_1, l_2, l_e), \vartheta)) = (e_p?, \vartheta_{e_p}) \in offers_?((l_1, l_2), \vartheta) \Leftrightarrow$
$e? = e_p? \wedge \vartheta_{e_p} - \vartheta_e = false.$

A composition conform timed automaton A is also *role conform* to the parallel composition $A_P = A_1 \parallel A_2$ of the original role automata A_1 and A_2, if there

exists a corresponding zone automaton Z'_A, with possibly less zone locations and corresponding transitions, where each zone location is consistent. This means that each zone location $((l_1, l_2, l_e), \vartheta) \in Z'_A$ offers the same behavior as its corresponding timed automaton location $(l_1, l_2) \in A_P$ starting in the zone ϑ. Furthermore, a zone location offers the same behavior if the sets of sending events are equal and if the set of receiving events of the zone location contains all tuples $(e?, \vartheta_e)$ of the original timed automaton location. Note that it might contain more tuples, as there might exist offered receiving events, where the time interval is smaller than the original one.

We only presented an extract of the example zone automaton of the composition conform timed automaton constructed in the preceding sections. Consequently, we complete the example by affirming that all zone locations of this zone automaton are in fact consistent and therefore the corresponding composition conform timed automaton is also role conform. Additionally, we employed the model checker UPPAAL to verify that the automaton does not contain any time-stopping deadlocks. Consequently, the example automaton is also observational timed bisimulation equivalent to the parallel composition of the original role automata.

However, this does not always have to be the case. Furthermore, the definition of role conformance only requires a zone automaton Z'_A to be existent, not that the original zone automaton Z_A, which corresponds to the composition conform timed automaton produced by the synthesis algorithm, is equal to Z'_A. Consequently, those zone locations of the original zone automaton Z_A which are not consistent have to be removed, in order to avoid situations where the timed automaton executes transitions which lead foreseeable into deadlocks. How this can be achieved is presented in the next section.

3.3.3 Preserving Deadlock Freedom

For a role conform, composition conform timed automaton A it is not ensured that some paths exist, where not all behavior of the corresponding role automata is refined properly. Though, as the composition conform automaton is also role conform, we know that a zone automaton Z'_A exists which has two relevant properties: (1) it is equal to the zone automaton Z_A of the composition conform timed automaton except that is has less zone locations and (2) its corresponding timed automaton refines the behavior of the individual role automata on all paths of the zone automaton. Consequently, to ensure that no transition of the role conform time automaton leads into a deadlock situation, we have to transform the zone automaton, where some zone locations have been removed, into a timed automaton.

To exemplify the procedure, we modify the simple rear role automaton and the simple registree role automaton (see Figure 3.5 and Figure 3.6, p. 61) as depicted

in figure 3.14 and figure 3.15. Both automata can now rest at most one time unit in each location. Furthermore, observe that they also have to rest at least one time unit in location convoy and registered respectively. When they switch back to their initial location, their corresponding clock is reset.

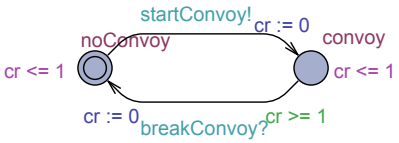

Figure 3.14: Modified Simple Rear Role Timed Automaton

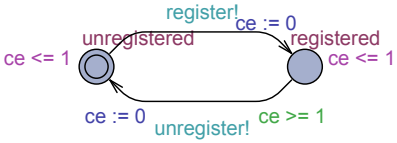

Figure 3.15: Modified Simple Registree Role Timed Automaton

We now construct the parallel composition of these automata and apply the state composition rule $r_1 = \neg((unregistered, true) \wedge (convoy, true))$. As we do not apply any event composition automaton, the resulting automaton (Figure 3.16) is composition conform.

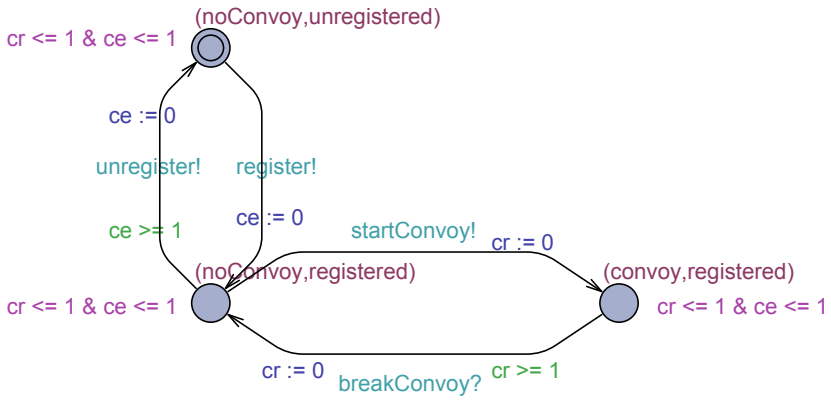

Figure 3.16: Composition Conform Timed Automaton of the Modified Simple Rear Role and Simple Registree Role Automata (Figure 3.14 and Figure 3.15)

In order to test for role conformance, we now create the corresponding zone automaton using the procedure described in section 2.1.2.3. The resulting zone automaton is depicted in figure 3.17.

Observe that every zone location of this zone automaton is consistent, except the location ((convoy,registered),cr-ce==-1 & ce==1), which has no outgoing transitions. If we remove this location, we obtain a zone automaton where every zone location offers the required events of the original role automata. Therefore, the composition conform timed automaton is also role conform. Though, it contains

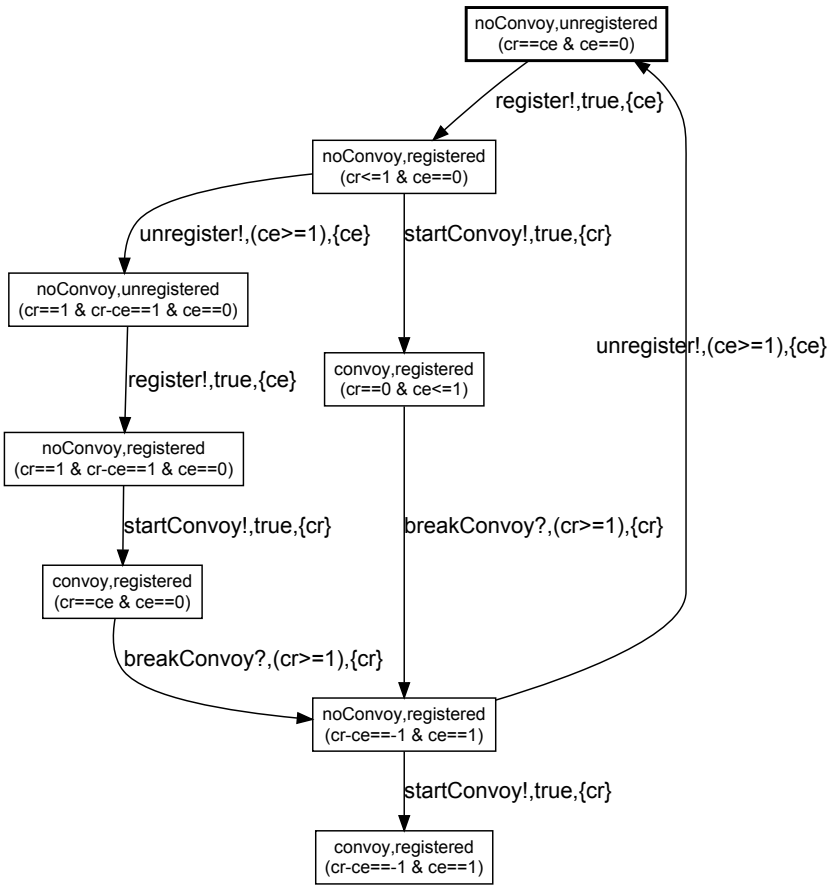

Figure 3.17: Zone Automaton for the Composition Conform Timed Automaton of figure 3.15

paths which do not offer the required events of the individual role automata correctly. These are all paths which lead to location ((convoy,registered),cr-ce==-1 & ce==1) through the startConvoy transition. Therefore, we have to remove this (timed) transition from the timed automaton, in order to obtain a timed automaton, where no execution of a transition leads into a deadlock. This removal is described by the following definition of a *zone automaton transition removal*.

Definition 3.3.13 (Zone Automaton Transition Removal)
For a zone automaton $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ constructed from a composition conform timed automaton $A = (L, l^0, \Sigma, C, I, T)$, a timed automaton transition $t = (l, e, g, r, l') \in T$, a source zone location $s = (l, \vartheta) \in S_\Theta$ and a zone automaton transition $t_\Theta = (s, e, s')$, the transition t_Θ is removed from the timed automaton A, and therefore also from the zone automaton Z_A by replacing the guard g of transition t with the guard g_r defined with

$$g_r = g \wedge (true - (\vartheta^{\Uparrow} \wedge I(l))).$$

In order to remove this zone automaton transition from the corresponding timed automaton, we have to replace the time guard of the transition t by a modified time guard g_r. This time guard intersects the original time guard g with the interval which is not restricted. This interval is computed by subtracting the restricted interval of the source zone location from the universal set of clock valuations $true$. The result is guard g_r which includes the clock valuations of the original guard g and removes those clock valuations of the guard which enable the transition in the concerned zone location l.

For the guard of the startConvoy! transition of the composition conform timed automaton, the computation results in the new guard of the transition depicted in figure 3.18. Observe that the guard contains a disjunction, which is actually not allowed by the definition of a general clock constraint (see Definition 2.1.1, p. 14, General Clock Constraint). This can be resolved by splitting the transitions into two transitions with same event and clock resets, but obtaining their time guards from the different constraints of the original disjunctive time guard. In the example, we omitted this procedure to simplify the illustration.

The zone automaton for the modified version of the role conform automaton is depicted in figure 3.19. Obviously, every path of this zone automaton correctly refines the behavior of the individual role automata. Correspondingly, we successfully removed the transition leading into a deadlock from the corresponding timed automaton.

It is possible, though, that there exist time stopping deadlocks in any of the zone locations which is not visible by only analyzing the outgoing transitions of all zone locations. Consequently, we have to utilize the model checker UPPAAL to search for time stopping deadlocks. Unfortunately, UPPAAL even finds

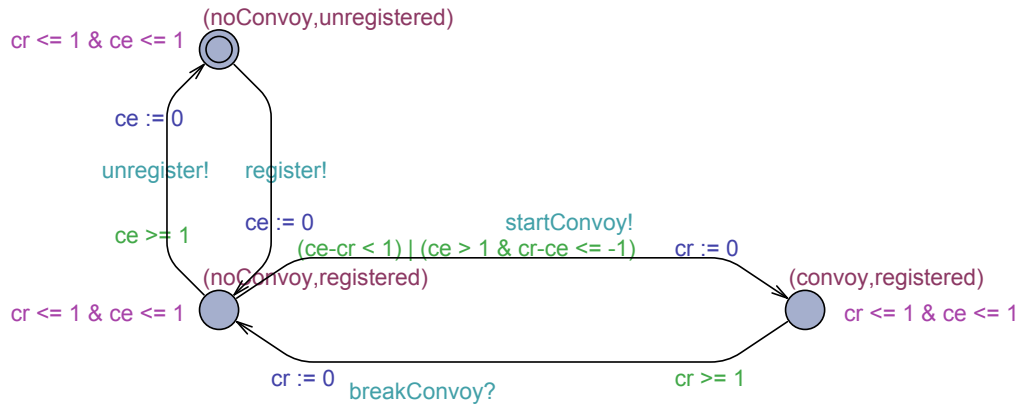

Figure 3.18: Modified Timed Automaton of the Role Conform Timed Automaton of Figure 3.16

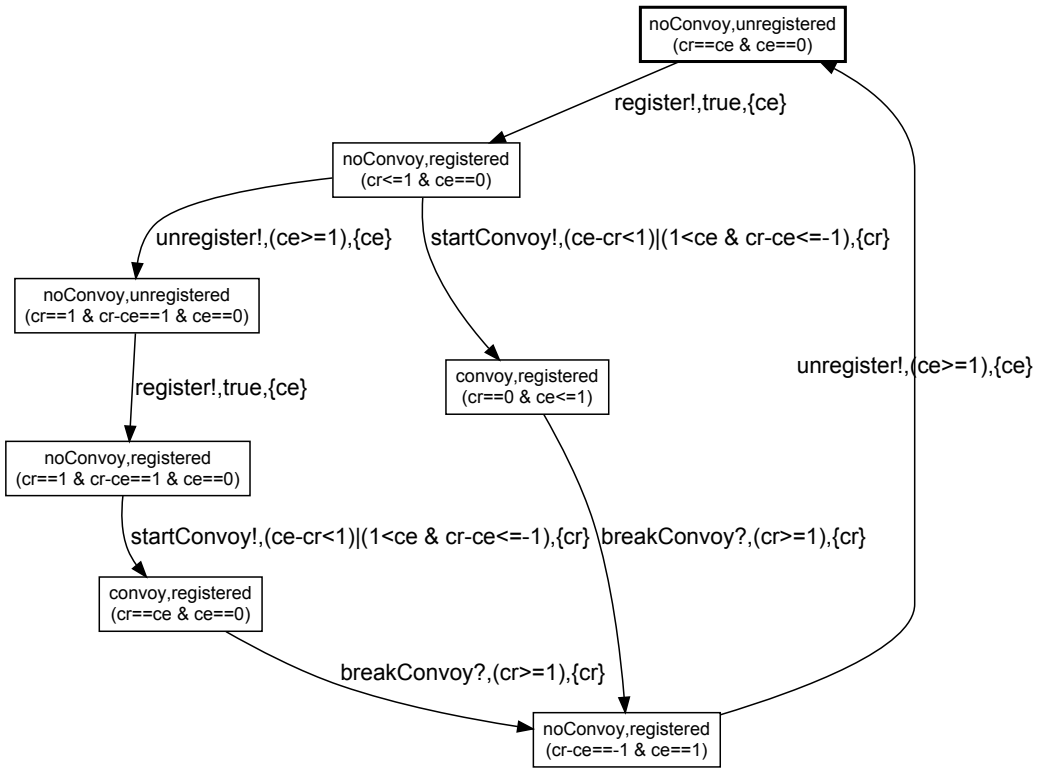

Figure 3.19: Resulting Zone Automaton from the Modified Timed Automaton of Figure 3.18

a deadlock which is described by the following path (cf. Figure 3.18): register! at $ce = cr = 0$, startConvoy! at $ce = cr > 0$, *deadlock* at $ce > 0 \wedge cr = 0$. This is a deadlock because the time guard of the only outgoing transition of location (convoy,registered) requires to rest in the location for at least one time unit, but this is not possible due to the location invariant and the value of ce being greater than zero. Furthermore, this deadlock cannot be removed by simply further restricting time guards of location invariants. As a consequence, we reason that such deadlocks still have to be removed manually by the developer. During this process, the developer may not widen any time intervals, as this would violate the simulation relation between the original parallel composition and the composition conform timed automaton. After the removal, the resulting timed automaton has to be checked once more for role conformance. If it passes this test, the resulting automaton is observational timed bisimulation equivalent to each of the role automata and therefore forms a component behavior which is a correct refinement of each of the single role behaviors.

In this chapter we presented the overall synthesis approach as the main contribution of this paper. This includes the definition of composition rules, the construction of an explicit behavioral model for the parallel execution of the individual role automata and finally the appliance of the composition rules. As this might result in the violation of before verified behavior of the role automata, we also proposed a method to check for a correct refinement by the notion of role conformance. The defined notion of role conformance together with the defined refinement relation of observational timed bisimulation guarantees that not only the universally quantified properties are preserved but also the untimed existentially quantified properties. To establish the refinement relation in the resulting timed automaton, however, time-stopping deadlocks have to be discovered and removed manually, if they have not already been removed automatically.

Chapter 4

Evaluation

In this chapter, we present implementation results. These presented implementations were developed in the scope of this paper in order to evaluate the proposed concepts. The implementation took place in a two phase process, as described in the following.

In the first phase, the proposed refinement relation (observational timed bisimulation and role conformance) was implemented. This was carried out by (1) integrating the existing synthesis implementation of Seibel [Sei07] into Fujaba4Eclipse, the Eclipse version of the Fujaba Real-time Tool Suite[1] and (2) modifying the included refinement relation. This has also been part of an ICSE Research Tool Demonstration [HGH+09].

Due to the discrete time semantic applied by Seibel, his implementation suffers a state explosion problem already for very small values used for the upper bounds of clocks (see section 2.1.2.2). In the second phase therefore, the appliance of a continuous time model (see section 2.1.2.3, Zone Automaton) was evaluated by prototypical implementing the zone automaton for the considered examples. Both of these phases will be described in detail in the following two sections.

4.1 Refinement Relation

In order to evaluate the applicability of the refinement relation proposed in this paper, it has been implemented using the prototypical synthesis implementation developed by Seibel [Sei07]. For this reason, the first step was to integrate Seibel's implementation into the research prototype of the development environment of MECHATRONIC UML, which is Fujaba4Eclipse. In this section we will shortly

[1] http://www.fujaba.de

introduce those parts of Fujaba4Eclipse which are relevant for this paper. After that, we describe the evaluated example stemming from the case study used throughout this paper (see section 1.1) along with the descriptions of extensions made on Fujaba4Eclipse.

As described in the introduction (chapter 1), MECHATRONIC UML supports the model driven development of mechatronic systems by applying component-based software development in combination with real-time coordination patterns. Consequently, this is also reflected in Fujaba4Eclipse. The screenshot depicted in figure 4.1 shows on the left-hand side the *project explorer* which displays an overview of the developed diagrams grouped by the type of each diagram. The diagram types which are of relevance for the following example are (from top to bottom): Application Specific Component Diagram, Pattern Diagrams, Protocol Statecharts, Realtime Statecharts, Software Component Diagram. We describe these diagram types briefly in the following while we generally adhere to the order of the development process.

Software component diagrams are used to define the basic components which are later used in the system. These diagrams do typically not contain any ports or interfaces. The components of the case study are the RailCab and the BaseStation component.

Pattern diagrams define the available real-time coordination patterns. Those define a number of roles, which can later be applied to one or more components. In our example, the specified pattern diagrams are Convoy and Registration containing the roles registree, registrar, rear and front respectively.

For each role, a *real-time statechart* is automatically generated, which defines the communication behavior of that role. In the conventional approach of MECHATRONIC UML, these real-time statecharts have to be edited in order to specify the actual communication behavior of the roles of a coordination pattern [GTB+03]. Examples of the rear role and the registree role realtime statechart are depicted in figure 4.2 and 4.3. Observe that the sets of events of realtime statecharts belonging to different coordination patterns have to be disjoint as required by the compositional model checking approach of MECHATRONIC UML.

Once all role behaviors are specified, an *application specific component diagram* can be created to apply coordination patterns to components in an application specific setting. Such a diagram is depicted on the right-hand side of the screenshot in figure 4.1. This diagram displays the setting applied for the case study used throughout this paper. Each of the depicted RailCabs components applies one role of the convoy coordination pattern, such that they can potentially operate in convoy mode together. The RailCab applying the rear role additionally applies a registree role of a registration pattern, such that it can register to the BaseStation applying the registrar role.

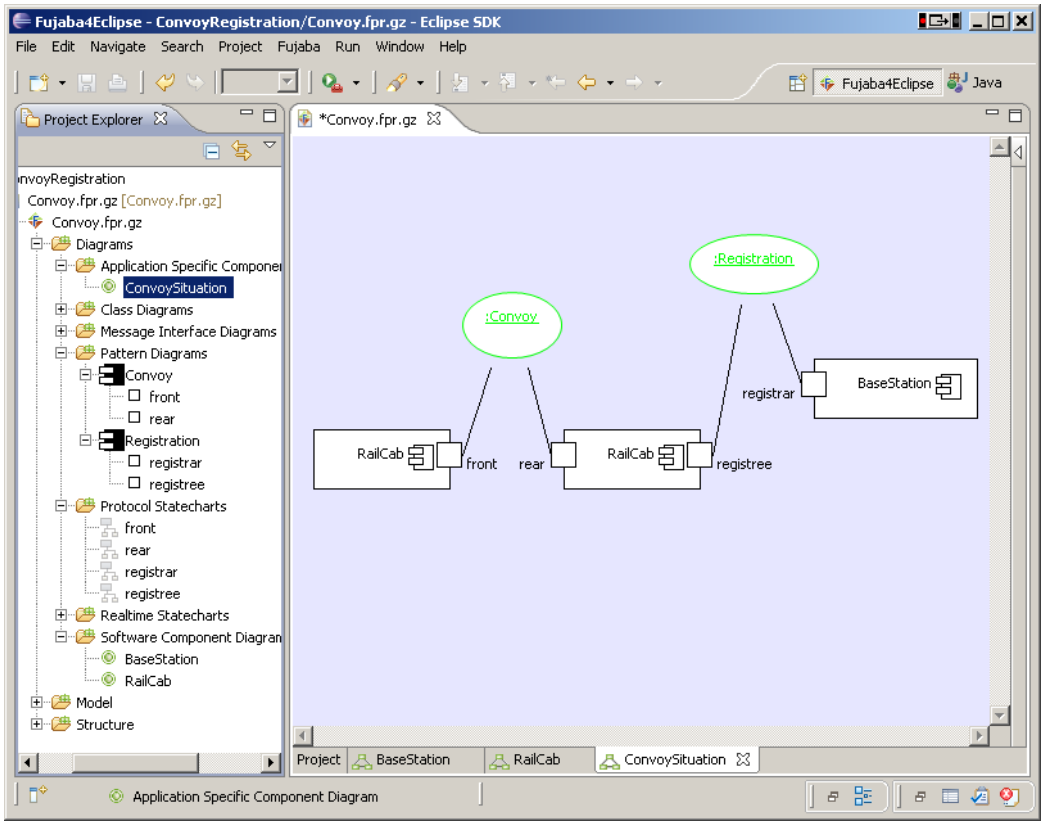

Figure 4.1: Fujaba4Eclipse Plug-in Showing the Project Explorer and an Application Specific Component Diagram

As soon as a role is applied to a component, the realtime statechart of the role is automatically copied to a *protocol statechart*. Being syntactically equal, the protocol statechart can then be refined, without influencing the more general behavior of the role. In our example, all roles are exactly applied once in the application specific component diagram. Accordingly there exists one protocol statechart for each role. Note, that these protocol statecharts do not have to be refined manually within the scope of this paper.

As we gave an overview of the basic principles of developing software for mechatronic systems in Fujaba4Eclipse, we proceed with the description of the evaluated case study. As the final synthesis procedure will be presented computing the component behavior of the RailCab component which applies the rear role and the registree role, we give short descriptions of the corresponding protocol statecharts in the following.

Both protocol statecharts (Figure 4.2 and Figure 4.3) generally correspond to the behaviors of the timed automata presented in section 3.1.1 (Figure 3.1 and Figure 3.2, p. 53). Therefore, we only describe the differing parts. In both statecharts the upper bounds for clock values are reduced to a minimal value. This was changed, as the underlying implementation of Seibel applies a discrete time semantic, which generates one state for each integer valued clock valuation (see section 5.3). Furthermore, location invariants are deduced automatically from the highest values used in time guards of the outgoing transition of the corresponding location. This was integrated in order to reduce the state space. Note that in this implementation, we further abstract from the fact that time can pass during the execution of realtime statechart transitions (cf. [BGHS04]). This simplifies the internal transformations between timed automata and realtime statechart while delivering the same synthesis results.

Before the synthesis of both of these statecharts can be reformed, we specify a state composition rule as depicted in figure 4.4. This state composition rule corresponds to the state composition rule r_1 of section 3.1.1. Accordingly, it specifies that the RailCab component is not allowed to be in state unregistered and state convoy at the same time. Observe that we applied a notation for the state composition rule, which distinguishes between states of different roles by referring to the role for each location. The location predicate registree.unregistered accordingly refers to the location unregistered of the registree role protocol statechart.

As soon as all composition rules are specified, the synthesis procedure is initiated by pressing the right mouse button while the cursor is over the RailCab component and choosing Synthesize Component Behavior (see Figure 4.4).

In the background of the application, the implemented extension now creates a discrete time abstraction, called *finite integer semantic (FIS)*, of each protocol automaton as proposed by Seibel [Sei07, pp. 30–33]. After that, a parallel composition is applied to both constructed FIS and the states specified by the state com-

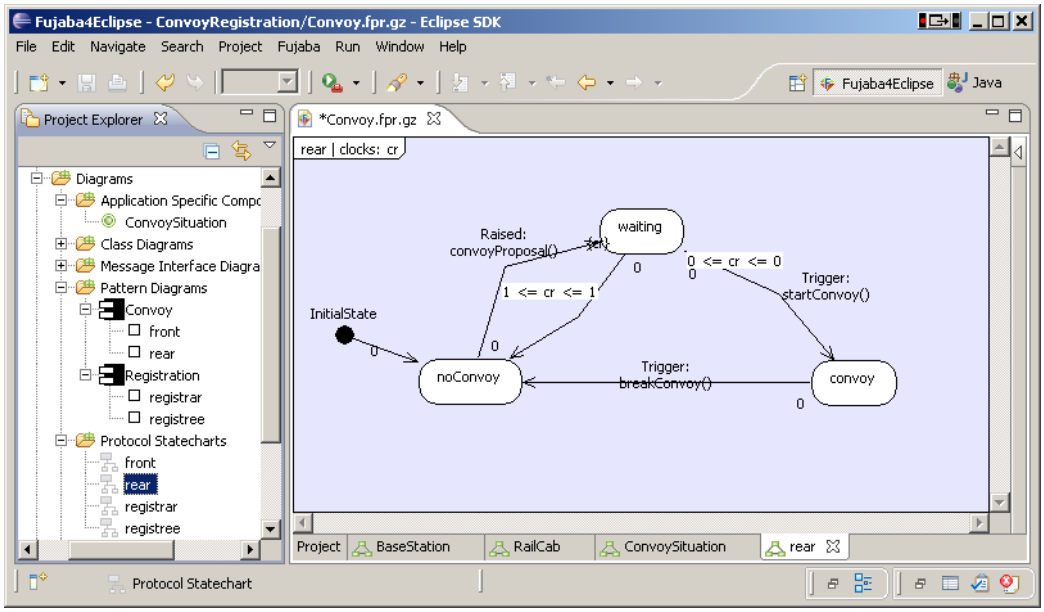

Figure 4.2: Rear Role Protocol Statechart

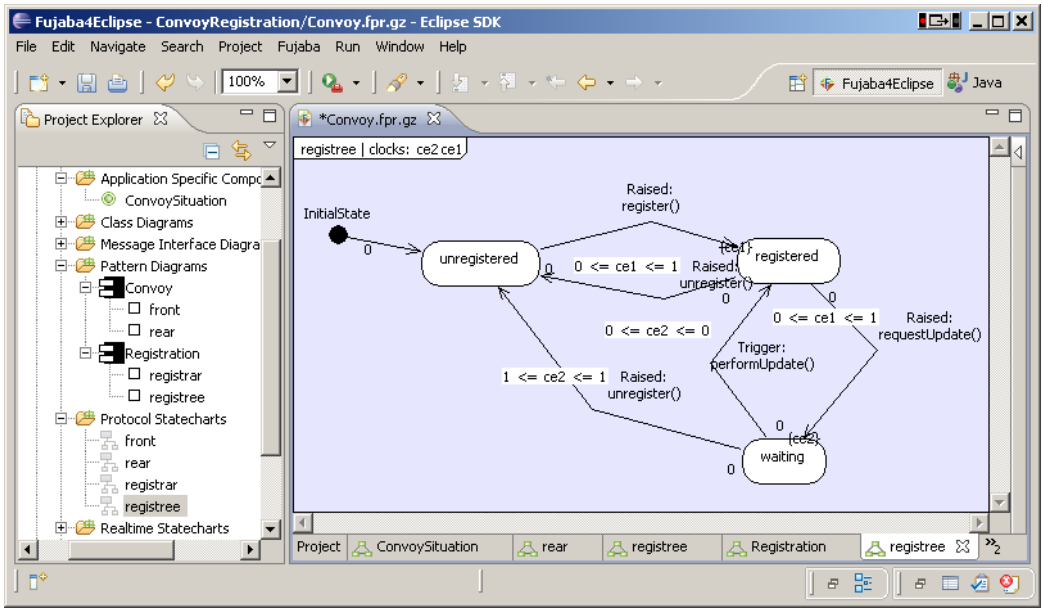

Figure 4.3: Registree Role Protocol Statechart

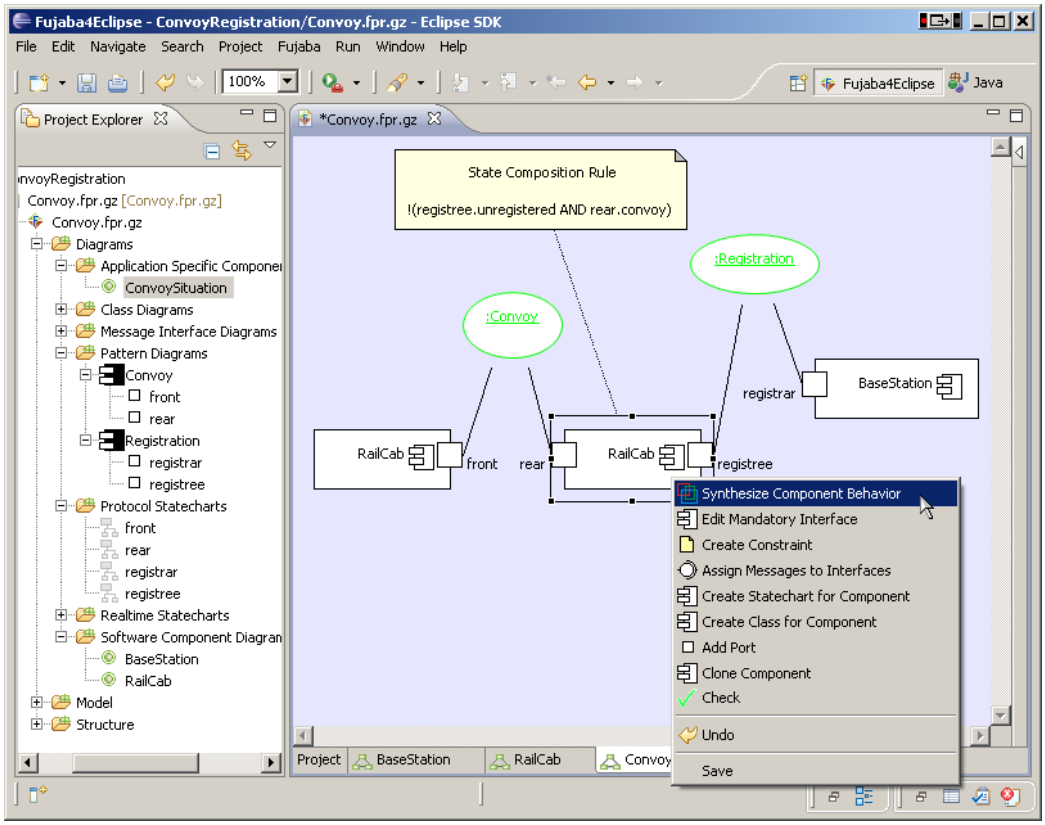

Figure 4.4: Specified State Composition Rule and the Menu Entry to Synthesize the Component Behavior

position rule are removed from that parallel composition. On that resulting FIS, our refinement check is implemented. This means that every state of the FIS is checked concerning the offered behavior of the original parallel composition (cf. section 3.3.2). Within this procedure, from on protocol statecharts perspective, events of the other protocol statechart are treated is internal component behavior. If all states offer the same behavior with respect to this equivalence relation, the result is a *protocol conform* (or role conform) FIS which is then transformed back to a real-time statechart.

The result of the synthesis procedure is the role conform realtime statechart depicted in figure 4.5. For a better understanding of this statecharts we implemented a layout algorithm which positions the parallelly composed states in a way that all states of one role form a state of the other role. In our example all states of the registree behavior (states (registree.unregistered,...), (registree.registered,...) and (registree.waiting,...)) form a node of the rear role behavior. As the primary result, observe that the state (registree.unregistered,rear.convoy) has been removed by the composition rule appliance, while the relevant behavior of each protocol statechart is still present.

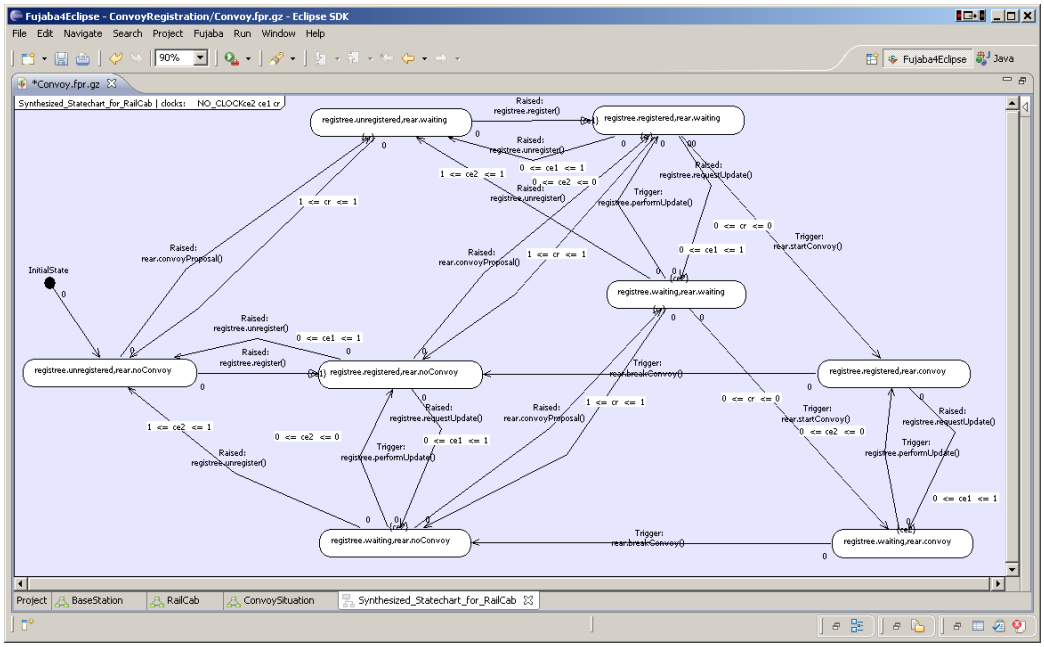

Figure 4.5: Synthesized Component Behavior of the RailCab Component

In this section we presented our evaluating implementations concerning the refinement relation proposed in this paper. The main result is, that this refinement relation seems suitable for the compositional approach of MECHATRONIC

UML. Though, this was implemented using a discrete time abstraction for timed automata which suffers a state explosion problem already for very small clock values. Consequently, we present the experimental results of the continuous time abstraction of timed automata in the next section.

4.2 Timed Automata Abstraction

As discussed in section 2.1.2.2, a discrete time semantics is not feasible for mechatronic system because of several reasons. One of this reason is the state explosion problem, which is present if realistic bounds for clocks are used in timed automata. In this section, we present experimental results comparing our zone automaton implementation with the implementation of the *finite integer semantic (FIS)* developed by Seibel [Sei07].

As the input for the role conformance algorithm is a time abstracting model of the parallel composition of two or more timed automata, the size of this model is a crucial parameter to influence the runtime of the algorithm. Consequently, we focus on the construction of the time abstracting model corresponding to the parallel composition of two automata.

Before we present the examined automata, we describe both implementations briefly in the following. Seibel's synthesis approach based on the FIS is implemented in Java and takes a set of timed automata modeled in UPPAAL as the input. The first step in his implementation is to create a separate FIS for each timed automaton. After that, the parallel composition of these separate FIS is computed. Our zone automaton approach makes use of the UPPAAL UDBM library[2], which provides efficient algorithms for clock zones and operations on clock zones implemented in C. In this approach a clock zone is represented by a *difference bound matrix (DBM)*, which has already been proposed by Dill in [Dil89] as an efficient data structure to represent convex sets like clock zones. For a detailed overview on DBMs and operations on DBM we refer to [BY03]. In order to make use of the UDBM library, we utilized the provided Ruby binding and implemented the algorithms also in Ruby. Our implementation takes an already parallelly composed timed automaton as the input and computes the corresponding zone automaton.

The automata examined in this section are the simplified version of the rear role and the registree role automaton already presented in section 3.2.1 (p. 61). In order to examine the influence of the specified bounds for clock values, we parameterized the three bounds using the parameters earliestConvoy, maxConvoy and maxRegister (see Figure 4.6 and Figure 4.7). This way, we are able to ex-

[2] http://www.cs.aau.dk/ adavid/UDBM/

amine both, the highest existing bound values as well as the differences between bound values in one or in different automata of the parallel composition.

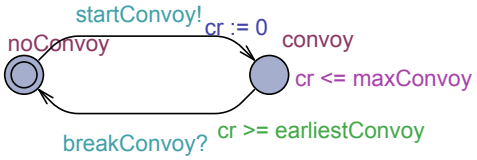

Figure 4.6: Parameterized Simple Rear Role Timed Automaton

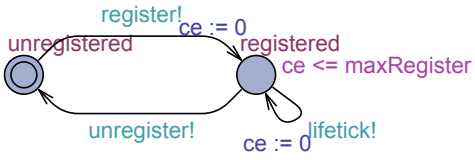

Figure 4.7: Parameterized Simple Registree Role Timed Automaton

The examined configurations of the parameters as well as their evaluations results are presented in the table depicted in figure 4.8. We started the experiment by applying small values for all parameters and step by step increasing the values while also varying the difference between earliestConvoy and maxConvoy and between maxConvoy and maxRegister. In general, it can be observed that, moving towards higher clock values, the number of states and transitions but also the computation time increases much faster in the FIS implementation. In the last three experiments, the computation could not even be accomplished by the FIS implementation due to stack overflow errors.

(ealiestConvoy,maxConvoy,maxRegistered)	Approach	States	Transitions	Time (ms)
(0,1,1)	FIS	16	100	5
	ZA	16	38	21
(1,2,2)	FIS	33	226	15
	ZA	44	102	68
(4,5,10)	FIS	200	1600	172
	ZA	67	153	104
(1,5,10)	FIS	245	2102	287
	ZA	122	283	344
(9,10,20)	FIS	645	5397	1917
	ZA	195	440	708
(2,10,20)	FIS	848	7683	4208
	ZA	122	283	344
(3,15,20)	FIS	1233	11301	11662
	ZA	254	591	1396
(2,10,40)	FIS	xxx	xxx	xxx
	ZA	192	443	729
(3,15,30)	FIS	xxx	xxx	xxx
	ZA	122	283	338
(5,20,20)	FIS	xxx	xxx	xxx
	ZA	72	168	141

Figure 4.8: Evaluation Results for the Different Configurations of the Parameters

In order to examine the results in more detail, we illustrated the results as presented in figure 4.9, figure 4.10 and figure 4.11. When comparing the differences in the number of states and the number of transitions between the two approaches,

it can be noticed that in the results of the FIS with a higher number of states, the number of transitions increases to much higher values. In the results of the zone automaton on the other hand, this difference is not that evident. Though, a higher number of transitions has a negative influence on the role conformance algorithm, as this searches all possible paths in order to compare the offered events of a state.

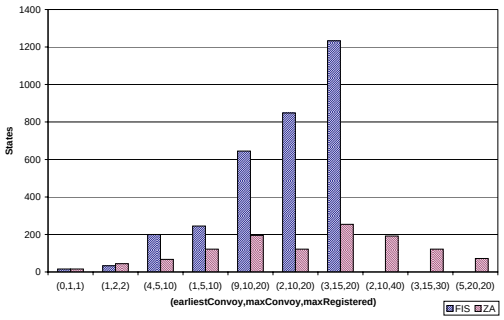

Figure 4.9: Comparison of the Number of States

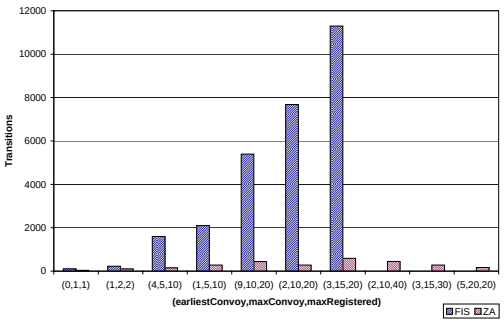

Figure 4.10: Comparison of the Number of Transitions

When comparing the computation times of the different computations of both approaches, it can be observed that the computation of the zone automaton does not increase proportionally to the absolute values of the parameters in contrast to the computation of the FIS. For configurations (1,5,10) and (2,10,20) the computation times for the zone automaton are even equal. This can be explained by the fact that the zone automaton does not take the absolute values into account which used for bounds but instead considers the time intervals resulting from differences between clock values.

Summarizing this evaluation, the zone automaton approach seems to be a feasible approach for efficiently abstracting from the infinite state space inherently arising from timed automata. It can also be observed, however, that the efficiency varies depending the differences between the values used in clock constraints in the corresponding timed automata models. Unfortunately, Seibel's implementation could not utilized to integrate the zone automaton approach into Fujaba4Eclipse, as already the parallel composition of the input automata is performed on their corresponding finite integer semantics.

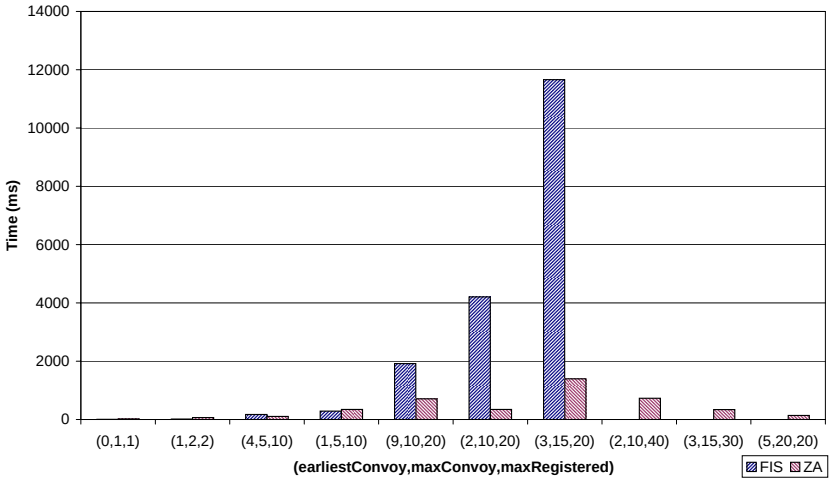

Figure 4.11: Comparison of the Computation Time

Chapter 5

Related Work

In this chapter we discuss research approaches which are in some points related to our synthesis approach. By related we mean that all approaches have in common that a behavioral synthesis is performed in order to apply a given rule (requirement) to a given behavioral model, such that the relevant behavior of the original model is preserved, or the appliance is rejected. We classify these related approaches into three major groups: Controller Synthesis, Synthesis of Untimed Component Behavior and Synthesis of Discrete Time Behavior.

5.1 Controller Synthesis

The field of controller synthesis [AMP95, AMPS98, AT02, BK06, GGR08] deals with the problem of synthesizing a behavioral model for a *controller* which interacts with some *environment*.

In a controller, interaction is specified through alternating actions between the controller and the environment. Consequently, for the behavioral model a special type of timed automaton, a *timed game automaton* [AMP95, MPS95], is applied. In a timed game automaton, transitions are partitioned into those controllable by the controller and those controllable by the environment.

The input timed game automaton is typically underspecified (also called *open*), such that additional properties have to be integrated by the synthesis procedure in order to ensure required safety or liveness properties. For the description of the additional properties, formulas of the *linear temporal logic (LTL)* [Pnu77], the *metric temporal logic (MTL)* [CMP94] or the *timed computation tree logic (TCTL)* [ACD90, ACD93] are utilized and therefore form the second part of the input of the synthesis procedure.

The starting point of the controller synthesis procedure is the open timed game automaton. On this open model the synthesis procedure is performed, which in-

tegrates the requirements given as LTL, MTL [BK06] or TCTL formulas [AT02] into the given model. The output is either a *closed* timed game automaton or a diagnostic description, referring to the reason why a property could not be integrated.

The main difference to our synthesis approach is that the given behavioral model of controller synthesis does not take a compositional character of this model into account as this is not necessarily given in the underlying controller behavior. In our approach this is given by the independent pattern role automata. Consequently, the compositionality can also not be considered for the specification of the properties which have to be synthesized. Altogether, this results in a different equivalence relation between the original and the synthesized model which in turn results in different synthesis algorithms.

5.2 Synthesis of Untimed Component Behavior

Giese and Vilbig proposed in [GV06] a synthesis procedure for the behavior of interacting components. In this approach, the interaction between different components is specified using *contracts* [Gie00]. A contract is used, similar as an interface in the *Unified Modeling Language (UML)* [OMG09], to specify available operations but additionally to specify valid interaction sequences of these operations. These valid interaction sequences are defined using state diagrams, which may include special internal actions specified through so-called τ-transitions. These internal actions are not further distinguished and therefore form *representational non-determinism* [WM97].

Furthermore, a contract is specified independently from other contracts, similarly to coordination patterns in MECHATRONIC UML. Consequently, when one component takes part in several contracts, state combinations appear, which might be forbidden due to system requirements. Consequently, a *state restriction* formalism is introduced to specify forbidden state combinations.

The synthesis procedure defined by Giese and Vilbig can be divided into three steps: (1) the parallel composition of the state diagrams is constructed, (2) the restricted state combinations are removed from this parallelly composed state diagram and (3) it is verified that the resulting state diagram still refines each of the participation contracts properly. If this is the case, the result is a maximal contract conform state diagram for each participating component with respect to defined state restrictions.

As historically, the approach proposed in this paper originated from the approach by Giese and Vilbig, the fundamental synthesis approach, described by the 3 steps mentioned above, is very similar (abstracting from the notion of time present in this paper). Though, both approaches differ in the applied behavioral

model and the applied equivalence relation defined between the original and the synthesized model.

The behavioral state diagram defined in contracts specifies a sequence of events (or actions) but does not distinguish between sending and receiving events. In MECHATRONIC UML this is distinguished by defining several roles for one coordination pattern instead of one contract. Contract based specification of interactions is sufficient for software component specifications, where the direction of the action calls is implicitly defined by the type of the contract (required or provided). Though, this is not applicable for specifying communication behavior as in MECHATRONIC UML's coordination patterns. Here, the direction of the event has to be considered, for example, in order to define the time interval in which one component has to listen on the communication channel.

The second difference is the applied equivalence relation, which is used to define if a synthesized component behavior is contract conform or not. The equivalence relation of Giese and Vilbig (called contract conformance) is based on the representational non-determinism possibly present in contracts through τ-transitions. This representational non-determinism is not applicable in such a late development phase, where MECHATRONIC UML's coordination patterns are applied. Though, the basic idea of arbitrary internal behavior inbetween the relevant actions of one role is preserved in this paper.

5.3 Synthesis of Discrete Real-time Component Behavior

Seibel extends the approach of Giese and Vilbig in [Sei07] by the notion of time concerning the behavioral models and the state restrictions. The type of timed behavioral models applied are timed automata [AD90, HNSY92] as also applied in the model checker UPPAAL [BDL04] (see also section 2.1).

Timed automata generally imply a concept of sending and receiving events and therefore require two different behavioral models for two communicating components. The concept of contracts, however, requires that one state diagram is defined for two interacting components. Consequently, Seibel replaces the concept of contracts by a *port* concept, similar to ports in UML [OMG09]. In this port concept, communication between components is specified using one port for each participating component, while the behavior of a port is defined by a timed automaton. Resulting from that, the synthesis procedure is defined on timed automata.

Seibel extends the restriction formalism as he adds time constraints to state restrictions similar to our state composition rules. Furthermore, he introduces

restriction automata which are concurrently executed to the port automata and whose events have to be complementary to those of the port automata. This way, restriction automata can listen to events of the port automata, such that they restrict certain event sequences.

For the synthesis procedure and formal analysis on timed automata a discrete time semantics, called *finite integer semantic (FIS)* (cf. section 2.1.2.2), is constructed which discretizes the time to integer steps. On this FIS, both the parallel composition as well as the state restrictions are applied.

The synthesis procedure basically divides into the same three steps of (1) parallel composition, (2) restriction appliance and (3) checking for *protocol conformance*, which corresponds to the contract conformance by Giese and Vilbig. Though, the complete procedure is defined on the FIS. Consequently, if a protocol conform FIS could be synthesized, the protocol conform is transformed back to a timed automaton, representing the component behavior.

As Seibel's approach extends the approach of Giese and Vilbig, the fundamental idea of the synthesis procedure which is constructing the parallel composition, applying composition rules and verifying if the original behavior is still present, is again similar to our proposed procedure. In addition to that, the applied port concept is very similar to the role concept in MECHATRONIC UML. The only difference is that roles are defined independently from a particular component in real-time coordination patterns. The idea to further apply automata as composition rules is derived from the idea of restriction automata. Though, the realization is different: state composition automata do not interact with the role automata in contrast to Seibel's restriction automata.

The main difference between our approach and Seibel's approach is the discrete abstraction of time which is applied by Seibel, and which is not applicable for MECHATRONIC UML (see section 2.1.2.2). Furthermore, Seibel applies the parallel composition and the state restrictions on the FIS, while we only use the abstraction to verify role conformance. At last, Seibel also allows for representational non-determinism by the explicit use of τ-transitions representing internal component behavior, instead of treating the behavior of other ports as internal component behavior. Consequently, the applied equivalence relations differ in this point again. Though, Seibel proposes to additionally allow an arbitrary number of delay transitions between the relevant actions of the port automata, which is equally applied in our observational timed bisimulation (see section 3.3.1).

Chapter 6

Conclusion and Future Work

In this section we conclude the overall results of this paper. In addition to that, we describe those concepts of the procedure, where additional work is suggested for the future.

6.1 Conclusion

In this paper we proposed an approach to automatically synthesize the behavior for a MECHATRONIC UML component which takes part in several real-time coordination patterns. The current approach of MECHATRONIC UML suffers from the manual refinement and synchronization of the coordination role behaviors which has to be accomplished to construct the component behavior. Therefore, we propose to specify dependencies between several role automata separately by means of composition rules. Additionally, we defined a procedure to automatically integrate the composition rules for a given set of role automata.

We propose two kinds of composition rules. State composition rules remove either complete location combinations from the parallel composition of the individual role automata or only specified time intervals of the defined location combinations; event composition automata add further time constraints to locations and transitions of the parallel composition by introducing new clocks, while referring only to those clocks in the added time constraints. Both types of composition rules only remove transitions from the underlying timed automaton semantics. This preserves all universally quantified properties which might have been verified on the individual role automata in advance, as long as no time-stopping deadlock is introduced. This additionally has to be checked for the synthesized timed automaton after the synthesis procedure is performed. The real-time model checker UPPAAL is proposed to be applied for this.

We pointed out, however, that only preserving the universally quantified properties does not necessarily preserve the relevant behavior of a coordination role being part of a mechatronic real-time system. Therefore, we defined an appropriate refinement relation by means of the observational timed bisimulation, which also preserves the untimed existentially quantified properties. We assume that this indeed preserves all relevant behavioral properties of the coordination roles when automatically refined for the implementation in a component. We further assume that this is also appropriate for the compositional approach of MECHATRONIC UML, where the compositional model checking approaches of abstraction and assume/guarantee reasoning are applied.

In order to establish the proposed refinement relation we developed an automatic procedure to check for role conformance, which describes the untimed existential part of the refinement relation. This procedure employs an efficient abstraction of the timed automaton, the zone automaton. The zone automaton is an appropriate model for the reachability analysis of a timed automaton regarding the offered events of each location. During this analysis certain types of deadlocks can be removed automatically. Nevertheless, the model checker UPPAAL also has to be applied also at this point, in order to find time-stopping deadlocks. If time-stopping deadlocks are found, the system developer has to try to remove these deadlocks manually and check once more for role conformance. If the model is role conform and does not contain any time-stopping deadlocks, the result is a component behavior which preserves the relevant behavior of the participating coordination roles while also taking the specified composition rules into account.

6.2 Future Work

Although the described synthesis approach already defines a sound procedure for the automatic construction of the behavior of a MECHATRONIC UML component, several aspects are suggested for future work.

The most obvious point is the complete evaluation of the approach regarding a realistic set of case studies. Although we were able to accomplish partial evaluations of the proposed concepts, the complete approach was only studied using one example of the RailCab research project. This could be analyzed thoroughly inspecting the identified real-time coordination patterns of May [May08], for example. This way, it could also be evaluated if the proposed composition rule formalism is sufficient to specify existing dependencies between several coordination roles.

Concerning the composition rule formalism, it might also be of interest to specify composition rules which describe parts of the synchronization behavior which may not be removed from the parallel composition. Those rules might be

called *positive composition rules*, as they would not remove any behavior of the underlying timed automaton semantics, but ensure that the described behavior is preserved.

In addition to that, specification patterns [DAC98, DAC99, KC05] could be applied for the specification of synchronization properties defined by composition rules. This way, a set of rule patterns of best-practice could be established to facilitate the specification of composition rules.

With regards to time-stopping deadlocks, the whole approach could be extended to also support integer variables in timed automata and real-time statecharts respectively, like they are for example available in the timed automaton formalism of UPPAAL. This way, it might even be possible to remove all time-stopping deadlocks automatically from the role and composition conform timed automaton.

Bibliography

[ABB+98] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon D. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998. 2

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-time Systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), 4-7 June 1990, Philadelphia, Pennsylvania, USA*, pages 414–425. IEEE Computer Society, June 1990. 1, 22, 34, 111

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104(1):2–34, 1993. 22, 23, 34, 111

[ACH+92] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, David L. Dill, and Howard Wong-Toi. Minimization of Timed Transition Systems. In *Proceedings of the Third International Conference on Concurrency Theory (CONCUR '92)*, Lecture Notes in Computer Science (LNCS), pages 340–354, London, UK, 1992. Springer-Verlag. 23

[AD90] Rajeev Alur and David L. Dill. Automata for Modeling Real-time Systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science (LNCS)*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 11, 12, 22, 113

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 22

[AD96] Rajeev Alur and David L. Dill. Automata-theoretic Verification of Real-time Systems. In Constance Heitmeyer and Dino Mandrioli, ed-

itors, *Formal Methods for Real-Time Computing*, volume 5 of *Trends in Software*, pages 55–82. John Wiley & Sons, May 1996. 23

[Alu98] Rajeev Alur. Timed Automata. In *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems*, 1998. 23, 24, 30, 31

[Alu99] Rajeev Alur. Timed Automata. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99), July 6-10, 1999, Trento, Italy*, volume 1633 of *Lecture Notes in Computer Science (LNCS)*, pages 8–22. Springer Verlag, 1999. 23, 24, 26, 30, 31

[AM04] Rajeev Alur and P. Madhusudan. Decision Problems for Timed Automata: A Survey. In *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 (LNCS)*, pages 1–24. Springer Berlin / Heidelberg, 2004. 12

[AMP95] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic Controller Synthesis for Discrete and Timed Systems. In *Hybrid Systems II*, pages 1–20, London, UK, 1995. Springer-Verlag. 111

[AMPS98] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller Synthesis for Timed Automata. In *Proceedings of the 5th IFAC Cconference on System Structure and Control (SSC'98)*, pages 469–474. Elsevier Science, July 1998. 111

[AT02] Karine Altisen and Stavros Tripakis. Tools for Controller Synthesis of Timed Systems. In Paul Pettersson and Wang Yi, editors, *Proceedings of the 2nd Workshop on Real-Time Tools (RT-TOOLS'02)*, August 2002. 111, 112

[Bü62] J. Richard Büchi. On a Decision Method in Restricted Second-Order Arithmetic. In *Proceedings of the 1960 International Congress for Logic, Methodology, and Philosophy of Science*, pages 1–1. Stanford University Press, 1962. 12

[BDFP00] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Expressiveness of Updatable Timed Automata. In Mogens Nielsen and Branislav Rovan, editors, *Proceedings 25th International Symposium of Mathematical Foundations of Computer Science*

(MFCS 2000), Bratislava, Slovakia, August 28 - September 1, 2000, volume 1893 of *Lecture Notes in Computer Science (LNCS)*, pages 232–242. Springer, 2000. 14

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004. 1, 12, 23, 61, 113

[Bey01] Dirk Beyer. Efficient Reachability Analysis and Refinement Checking of Timed Automata Using BDDs. In Tiziana Margaria and Thomas F. Melham, editors, *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001), Livingston, Scotland, UK, September 4-7, 2001*, volume 2144 of *Lecture Notes in Computer Science (LNCS)*, pages 86–91, 2001. 22, 23

[Bey02] Dirk Beyer. *Formale Verifikation von Realzeit-Systemen mittels Cottbus Timed Automata*. Dissertation, Brandenburgische Technische Universität Cottbus, November 2002. 22

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proceedings of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, pages 1–20, October 2004. 13, 34, 41, 48, 51, 102

[BGO06] Sven Burmester, Holger Giese, and Oliver Oberschelp. Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In J. Braz, H. Araújo, A. Vieira, and B. Encarnacao, editors, *Informatics in Control, Automation and Robotics I*. Springer, March 2006. 47

[BGT05] Sven Burmester, Holger Giese, and Matthias Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In Uwe Assmann, Arend Rensink, and Mehmet Aksit, editors, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science (LNCS)*, pages 47–61. Springer Verlag, August 2005. 2, 47

[BK06] Borzoo Bonakdarpour and Sandeep S. Kulkarni. Automated Incremental Synthesis of Timed Automata. In Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology, 11th International Workshop, FMICS 2006 and 5th International Workshop PDMC 2006, Bonn, Germany, August 26-27, and August 31, 2006, Revised Selected Papers*, volume 4346 of *Lecture Notes in Computer Science (LNCS)*, pages 261–276. Springer-Verlag Berlin Heidelberg, 2006. 111, 112

[BLR05] Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. Diagonal Constraints in Timed Automata: Forward Analysis of Timed Systems. In Paul Pettersson and Wang Yi, editors, *Proceedings of the Third International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2005), Uppsala, Sweden, September 26-28, 2005*, volume 3829 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005. 14

[BN01] Dirk Beyer and Andreas Noack. Efficient Verification of Timed Automata Using BDDs. In *Proceedings of the 6th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS 2001), Paris, France, July 16-17, 2001*, pages 95–113, 2001. 22, 23

[Bur06] Sven Burmester. *Model-Driven Engineering of Reconfigurable Mechatronic Systems*. Dissertation, University of Paderborn, 8 2006. 44, 47

[BY03] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003. 16, 23, 24, 25, 26, 27, 29, 33, 87, 106

[CE82] Edmund M. Clarke and E. Allen Emerson. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2(3):241–266, December 1982. 34

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts / London, England, 4th edition, 1999. 23, 34, 77

[CMP94] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Compositional Verification of Real-time Systems. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science, 4-7 July 1994, Paris, France*, pages 458–465, 1994. 111

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In Mark A. Ardis and Joanne M. Atlee, editors, *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*, pages 7–15. ACM, 1998. 117

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE'99*, pages 411–420, 1999. 117

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 2

[Dil89] David L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407/1990 of *Lecture Notes in Computer Science (LNCS)*, pages 197–212. Springer-Verlag Berlin / Heidelberg, 1989. 106

[GB03] Holger Giese and Sven Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany, 6 2003. 11, 41

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proceedings of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pages 179–188. ACM Press, November 2004. 47

[GGR08] Stephanie Geist, Dmitry Gromov, and Jörg Raisch. Timed Discrete Event Control of Parallel Production Lines with Continuous Outputs. *Discrete Event Dynamic Systems*, 18(2):241–262, 2008. 111

[Gie00] Holger Giese. Contract-Based Component System Design. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS-33), 4-7 January, 2000, Maui, Hawaii*. IEEE Computer Society, 2000. 112

[GPV94] Aleks Göllü, Anuj Puri, and Pravin Varaiya. Discretization of Timed Automata. In *Proceedings of the 33rd IEEE Conference on Deci-*

sion and Control (CDC'94), Lake Buena Vista, Florida, USA, 1994, volume 1, pages 957–958, December 1994. 22

[GTB+03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, pages 38–47. ACM Press, September 2003. 2, 5, 6, 13, 100

[GV06] Holger Giese and Alexander Vilbig. Separation of Non-Orthogonal Concerns in Software Architecture and Design. *Software and System Modeling (SoSyM)*, 5(2):136 – 169, June 2006. 2, 7, 79, 112

[GV08] Orna Grumberg and Helmuth Veith, editors. *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag Berlin Heidelberg, 2008. 1, 34

[Hen92] Thomas A. Henzinger. Sooner is Safer than Later. *Information Processing Letters*, 43(3):135–141, 1992. 12, 51

[HGH+09] Stefan Henkler, Joel Greenyer, Martin Hirsch, Wilhelm Schäfer, Kathan Alhawash, Tobias Eckardt, Christian Heinzemann, Renate Löffler, Andreas Seibel, and Holger Giese. Synthesis of Timed Behavior from Scenarios in the Fujaba Real-Time Tool Suite. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), May 16-24, 2009, Vancouver, Canada*, pages 615–618. IEEE Computer Society, May 2009. 99

[HKWT95] Thomas A. Henzinger, Peter W. Kopke, and Howard Wong-Toi. The Expressive Power of Clocks. In Zoltán Fülöp and Ferenc Gécseg, editors, *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming (ICALP '95),Szeged, Hungary, July 10-14, 1995*, volume 944 of *Lecture Notes in Computer Science (LNCS)*, pages 417–428, London, UK, 1995. Springer-Verlag. 12

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS)*, pages 394–406. IEEE Computer Society Press, 1992. 1, 11, 12, 16, 113

[KC05] Sascha Konrad and Betty H. C. Cheng. Real-time Specification Patterns. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 372–381. ACM, 2005. 36, 117

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. 51

[LMST03] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Simone Tini. Concurrency in Timed Automata. *Theoretical Computer Science*, 309(1-3):503–527, December 2003. 14

[May08] Karl Alexander May. Identifikation von Koordinationsmustern in autonomen mechatronischen Echtzeitsystemen. Bachelor Thesis, University of Paderborn, 2008. 116

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. 61, 63, 69

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract). In Ernst W. Mayr and Claude Puech, editors, *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS 95), Munich, Germany, March 2-4, 1995*, volume 900 of *Lecture Notes in Computer Science (LNCS)*, pages 229–242. Springer, 1995. 111

[NKF94] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, 20:760–773, 1994. 2

[OMG09] Object Management Group. *Unified Modeling Language (UML) 2.2 Superstructure Specification*, February 2009. Document formal/2009-02-02. 2, 41, 112, 113

[Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Department of Computer Systems, Uppsala University, February 1999. 12, 61

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science, 1977*, pages 46–57. IEEE Computer Society Press, 1977. 111

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science (LNCS)*, pages 337–351. Springer-Verlag Berlin / Heidelberg, 1982. 34

[Sei07] Andreas Seibel. Behavioral Synthesis of Potential Component Real-Time Behavior. Diploma Thesis, Software Engineering Group, University of Paderborn, June 2007. 7, 22, 79, 99, 102, 106, 113

[SW07] Wilhelm Schäfer and Heike Wehrheim. The Challenges of Building Advanced Mechatronic Systems. In *Future of Software Engineering (FOSE '07), 2007*, pages 72–84. IEEE Computer Society, 2007. 1

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Addison-Wesley, 1998. 2, 5

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107–119, New York, NY, USA, 1999. ACM. 7

[TY01] Stavros Tripakis and Sergio Yovine. Analysis of Timed Systems Using Time-Abstracting Bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001. 77, 78, 79

[WL97] Carsten Weise and Dirk Lenzkes. Efficient Scaling-Invariant Checking of Timed Bisimulation. In Rüdiger Reischuk and Michel Morvan, editors, *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS 97), Lübeck, Germany, February 27 - March 1, 1997*, volume 1200 of *Lecture Notes in Computer Science (LNCS)*, pages 177–188. Springer Verlag Berlin Heidelberg, 1997. 77, 79

[WM97] Michał Walicki and Sigurd Meldal. Algebraic Approaches to Nondeterminism: An Overview. *ACM Computing Surveys*, 29(1):30–81, March 1997. 112

[Yov97] Sergio Yovine. Kronos: A Verification Tool for Real-Time Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):123–133, December 1997. 1, 12, 23

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-time Communicating Systems by Constraint-solving. In Dieter Hogrefe and Stefan Leue, editors, *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Formal Techniques, Berne, Switzerland, 1994*, volume 6 of *IFIP Conference Proceedings*, pages 243–258. Chapman & Hall, 1994. 61, 63, 69