# Verification of DEVS Models for Reconfigurable Systems

**Diploma Thesis**

TECHNISCHE UNIVERSITÄT DARMSTADT

**Julia Weingart**

**Advisor: Prof. Dr.–Ing. Sorin A. Huss**

**Tutor: Dipl.–Inf. Felix Madlener**

Darmstadt, the 14th September 2009

## Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe alle Stellen, die ich aus den Quellen wörtlich oder inhaltlich entnommen habe, als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, am 14.09.09.

## Acknowledgments

This page is devoted to those, who encouraged me over the whole period of my study and especially to those, who supported developing and preparing this diploma thesis.

This diploma thesis was done at the Department of Integrated Circuits and Systems at the Technical University of Darmstadt. First of all I would like to thank my tutor Dipl.-Inf. Felix Madlener for his valuable suggestions and observations as well as for constructive criticism during my work under his supervision.

I owe special thanks to Roman Mertyn for his fruitful discussions, which helped me to solidify my ideas, when no doubt he had more important things to do and for offered access for necessary information.

Great thanks to my tutor, George Khujadge and Kasia Heller, who read my thesis very carefully.

Of course, I am indebted to my friends and to my chief Dipl.-Bib. Silvia Röpke-Dönges who supported me during this time.

Especially, I want to thank my parents: Lilia and Anatoli Weingart, and my sister Natalia Derugin for their love, understanding and endless support in my life.

# Index of contents

# 1. Introduction

## 1.1. Motivation

Usually talking about reconfigurable computing systems means to talk about reconfigurable hardware systems, like field-programmable gate arrays (FPGA). Reconfigurable computing systems are a disruptive innovation currently going to complete the most important breakthrough after introduction von Neumann paradigm. They exploit reconfigurable devices, including truly revolutionary field-programmable gate arrays (FPGAs) device and benefits of both hardware and software advantages, providing huge power, area and performance [21]. FPGA contains an array of computational elements, whose functionality is determined by multiple programmable configuration bits. These elements, sometimes known as logical blocks, are connected using a set of routing resources that are also programmable. In this way, custom digital circuits may be mapped to the reconfigurable hardware by computing the logic functions of the circuit within the logic blocks, and using the configurable routing to connect the blocks together to form the necessary circuit [14].

Reconfigurable systems involve the manipulation of the logic within the FPGA at the run-time of the system, i.e. that the hardware design may be changed in response to the specifications placed upon the system while it is running. Hence, reconfigurable systems enable a dynamic and flexible adaptation of hardware structure to support different application tasks. In this case, the FPGA operates as an execution engine for a variety of different hardware functions. This means that FPGA-based systems can be programmed and reprogrammed many times implementing a wide range of tasks [21]. Bondalapati and Prasanna [4] defined in their research three main classes of reconfiguration for hardware resources: *statically, partially* and *dynamically reconfigurable* architectures. In the *statically reconfigurable* architectures a hardware system is configured just once following the execution requirements and can be reshaped as a whole. A *partial reconfigurable* architecture permits reconfiguration of smaller parts of the whole chip, while the remaining part of the chip preserves its functionality. As a consequence, a new dimension is added to the hardware design space in this kind of reconfiguration. The *dynamic reconfiguration* like partial reconfiguration permits reconfiguration of some part of the chip and in opposite to partial reconfiguration the other part of the components continues their execution.

Consequentially, this uniqueness of the FPGAs devices allows the engineers to execute more hardware than they have gates to conform. This functionality can be exploited especially well when there are parts of the hardware that are occasionally idle. Thus, by using reconfigurable computing systems it is possible to achieve greater functionality with the simplest hardware resources compared to classical hardware solutions. Furthermore, FPGAs cost less and have shorter design and implementation cycle, i.e. reduced time-to-market.

To sum up the facts given above, hardware systems are continuously growing in functionality and changing in their structure. This leads to an increased level of complexity in current hardware systems. As a result of this design complexity also increases the probability of substantial design errors. To achieve the development of reliable configurable systems despite their complexity, formal methods can be used due to the fact that extensive testing or simulation can easily miss significant errors when the

system is too large or just due to the fact that methods mentioned above are not sufficient enough for determining the hardware design correctness. Formal verification has been demonstrated to work reasonably well in practice [6]. The success of the formal verification method has attracted a fair deal of attention from both sides: the researchers' side and the industry, with existing progress being made on many fronts [19, 25].

Formal methods are mathematically-based languages, techniques and tools for modeling, specification and verification of the systems that can reliably operate despite the systems' growing complexity and functionality. Therefore, a formal verification is a proof that the designed model follows a supposed or asserted system behavior is true. The method of formal verification is an exhaustive process compared to the existing testing techniques, because it examines the whole dynamic behavior of a system when simple testing techniques explore only a particular execution.

Unlike most programming languages, formal methods offer strict defined semantics, i.e. a mathematical description of a property. This specification of semantics allows formal analyzing of the system model before the actual implementation and to find out potential nonconformities at the early design stages.

Formal methods can be applied and integrated in any stage of system's development. The main goals of the formal methods defined by Wing [40] are:

- To specify and assure non-functional behavior such as safety, reliability, performance, real-time and human-factors.
- To built more usable and more robust tools.
- To demonstrate that existing techniques scale up to handle real-world problems and to scale up the techniques themselves.

The use of formal methods reveals ambiguities, incompleteness and inconsistencies in the system.

To allow the application of formal verification onto real-world problems, foremost formal systems models must be developed. Examples for modeling real-time systems are frequently used Timed Automata [10], I/O-interval structures [38], Statecharts [20, 18] and Discrete Event System Specification (DEVS) formalisms [41] that provide excellent modeling facilities. These well known approaches suitable for automatic verification are based on the theory of the conventional finite state machines.

For a formal description of reconfigurable systems as a part of the real-time systems the formalism of parallel DEVS was taken as the basis in this diploma thesis. This Model of Computation was extended towards reconfigurability and is denoted as RecDEVS [28]. RecDEVS as a novel formal method of computation allows the formal description of the dynamically reconfigurable hardware systems especially.

In general, DEVS formalism [42] is an abstract basis for model specification due to its system theoretical basis and its undependability of any particular simulation implementation. It fits best the

timed behavior of reconfigurable systems. Furthermore, DEVS formalism provides hierarchical and modular composition, universality and uniqueness that support development of simulation models and integration of the diverse kinds of models needed in the different phases of hardware design process respectively. It is not only capable of hierarchical and modular modeling, but it is also widely used for verification methods too [15, 39].

## 1.2. Goals

The main goal of this diploma thesis is to develop and evaluate solutions which permit the verification of DEVS models for reconfigurable systems with the help of the formal verification methods, especially by using Model Checking technique. Therefore, the theoretical background of Model Checking is explained as well as the basic concept of DEVS. More precisely the fundamentals of parallel DEVS are explained and the applicability of the Model Checking technique on DEVS models is demonstrated.

One important task is to find a convenient tool for formal verification of DEVS models used in this work and to test its practicability on some examples including not only an elementary implementation of the DEVS model but also a complex one.

To make the right decision among a big variety of the existing verification tools used in industry for different purposes and to choose only one of them is a comprehensive and time-consuming task. One should properly know the theoretical background of the subject plus up-to-date knowledge in this field. The following criteria that are given by Clarke and Wing [12] characterize tools to be attractive for engineers. Here are some relevant criteria which select the most appropriate Model Checking tool for the defined task:

- Early payback. Methods and tools should provide significant benefits almost as soon as people begin to use them.
- Ease of use. Tools should be easy to use as compilers, and their output should be as easy to understand.
- Efficiency. Tools should make efficient use of a developer's time.
- Ease of learning. Notations and tools should provide a starting point for writing formal specification for developers who would not otherwise write them. The knowledge of formal specification needed to start realizing benefits should be minimal.
- Error detection oriented. Methods and tools should be optimized for finding errors, not only for certifying correctness. They should support generating counterexample as a means of debugging.
- Evolutionary development. Methods and tools should support evolutionary system development by allowing partial specification and analysis of selected aspects of a system.

The second task of this thesis is to explain mapping of the DEVS formalism into a Model Checking tool specific representation. This representation can then be used for verification in this Model Checker. The model mapping is exemplified with a complex DEVS example.

## 1.3. Overview

Chapter 1 includes introductionary material of reconfigurable hardware systems, an overview of this diploma thesis and defines main goals of the research. Chapter 2 gives a short review of the notions of the verification techniques including detailed description of the Model Checking approach. The most qualified Model Checking tools with full information about their advantages and disadvantages are briefly introduced in the same chapter.

Chapter 3 specifies DEVS formalism and transformation of DEVS models into the UPPAAL Model Checker in general. Exemplified transformation of the AutoVision example system [13, 29] and its detailed description is introduced in Chapter 4.

Chapter 5 defines the requirement specification language of the UPPAAL Model Checker, illustrates a verification process of the given example and gives an overview of the evaluated results. Finally, Chapter 6 closes the paper with conclusions and future prospects.

## 2. Verification

Formal verification [19, 25] is an important technique that is increasingly used nowadays in the hardware design establishing its correctness, i.e. establishing a relationship between an implemented model and a specification. The main goal of formal verification is revealing design errors by enhancing observability and exhaustive checking of the system model. The process of formal verification facilitates debugging process in the hardware design and reveals an incompleteness and inconsistence of the designed model as well. Hence, with the help of formal verification it is possible to prove whether the implementation satisfies a desired specification or not.

Formal verification requires usage of some kind of formalism to express the implementation of the model, specification and the relationship between them. The DEVS formalism that is used throughout this work to describe the implementation of the system model will be introduced in Chapter 3.1 as well as its usability in the performed study. To describe the specification requirements on the model, temporal logic is used in this paper due to its reasoning explicitly about the time [19].

In the meantime, some of the commonly encountered forms of proof methods are available for formal verification. Two general approaches for formal verification are highlighted in the theory: *Logical Inference (theorem proving)* and *Model Checking*.

*Logical Inference* is based on a formal version of mathematical reasoning about the system, usually using *theorem proving* approach. Theorem proving is a technique to verify an implemented model by finding a proof of proposed properties from the mathematical logic of the system. The model implementation and its properties are expressed as formulas in some mathematical logic. Primarily, theorem proving uses Higher Order Logic as a specification and describing language. This logic contains a set of axioms and a set of inference rules, i.e. theorem proving is a deductive process. The process of theorem proving establishes the proof of a property from the axioms of the system. The decision problem of the theorem may vary from trivial to insoluble depending on the underlying logic.

The proofs can be constructed by hand or with the help of automatic theorem proving (ATP) programs. Theorem proving tools rely on the techniques like structural induction and they deal with the infinite state spaces. Automated theorem provers require some form of manual intervention to control the proof process. The theorem proving process is a time-consuming and often error-prone process.

Another alternative of the formal verification technique, *Model Checking*, was developed in the 1980s independently by Clarke and Emerson [30] in the United States and by Queille and Sifakis [33] in France. In their approach a system is modeled as a state-transition graph and specifications are typically expressed in the propositional logic. Furthermore, with its' growing popularity many studies were done to use the Model Checking process for real-time systems [27]. Consequently, this implies the extension of propositional logic to temporal logic, which is proved to be useful for specifying concurrent systems and making statements about their behavior over time.

The advantage of Model Checking in contrast to theorem proving is that this approach is completely automatic and fast efficient according the used algorithms to examine complex and large systems. In

addition, it produces a counterexample when the implemented model does not satisfy the required specifications. Thus, the user can determines whether a failure is a design fault or whether it was caused by an illegal input. The disadvantage of this technique is that Model Checking is less powerful than theorem proving, because structural induction is not explicitly facilitated in this technique.

Actually, Model Checking is used to verifying finite state spaces, because it uses no Higher Order Logic. But in some cases, it can be used for verifying infinite state systems in combination with various abstraction and induction principles [10]. The Model Checking approach is extensively described in the next section.

## 2.1. Model Checking

Model Checking [10] is a well established formal method for the automatical verification of finite state concurrent systems, e.g. sequential designs and communication protocols, and also for automatically proving the correctness of these systems. Model Checking is an efficient technique that is usually implemented using highly optimized data structure and algorithms [34]. The advantage of Model Checking over traditional approaches based on simulation, testing and deductive reasoning is that it is based on exploration of the whole state space of the model and exhaustive exploration of all possible behaviors of the system, while other approaches explore only some of the possible variants of systems' behavior. The main goal of Model Checking is to prove if a given property specified as a propositional temporal logic formula satisfies the given state-transition graph of a system or not.

Figure 1 shows the execution of the system verification via Model Checking tool. So, Model Checking tool gets a model to be verified as an input. Further, Model Checker gets a specification to be verified in regard to the correctness on the input model. Model Checker delivers the result of the verification as an output. Hence, if the model satisfies the designed specification, it returns "ok!" or "true" and otherwise it returns "false", i.e. it checks whether the specification is fulfilled by the input model or not. If the property fails, the Model Checking tool explains via counterexample why a system is not correct detecting the source of the error, what is not possible by the logical inference.

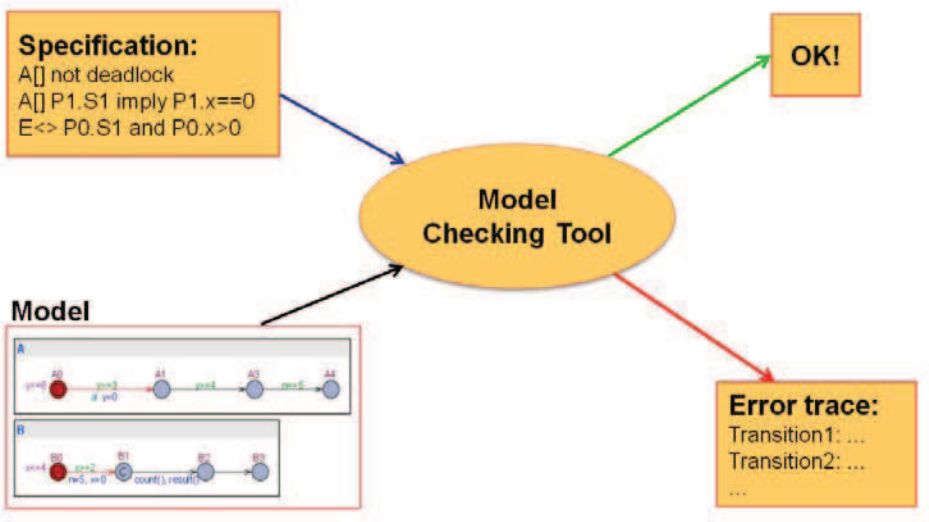


**Figure 1:** Verification process via Model Checking tool

Of great importance is the fact that Model Checking can be applied in the early stages of a design process. Therewith, Model Checking can be used to detect errors of the abstract model of the system or to prove their absence. Primarily, Model Checking is applied to the verification of the real-time systems and it is successfully used in practice to verify real industry designs [8]. The usage of the Model Checking tools increases the efficiency of the validation of the real-time systems and helps to generate systems with higher reliability.

The two main representatives of Model Checking algorithm are *explicit Model Checking* and *symbolic Model Checking*. These methods will be defined and explained in the next subsections giving significant information about their features.

### 2.1.1. Explicit Model Checking

The first well established Model Checking approach is *explicit Model Checking* [10]. In some papers and books it is also known as *on-the-fly Model Checking* [3, 7]. By explicit Model Checking the global state transition graph of the verified system will be explicitly constructed in the memory. Thus, every state and every transition of the graph is checked individually for the compliance of the specification requirement given by the user. The search of the state space requires the storage of every single system state in efficient way. The storage of visited nodes of the graph is also necessary for recognizing those already inspected. Hereby, visited nodes can be stored in the already inspected table. If the data set entry is found in this table, the node will be well-defined at a particular time as visited one. The stored states can be efficiently recognized using hash techniques.

From the explicit inspection of all system-states according to the size of the system results problems about required computing time and therefore required memory capacity. Different attempts are meanwhile made to extend the explicit Model Checking. The amount of algorithms are introduced to decrease respectively minimize the required memory capacity. All well known Model Checkers, which are based on the explicit Model Checking technique, employ search algorithms to traversal the state transition graph and therefore do not combine well with BDD-based methods [7, 3]. These Model Checkers usually include following algorithms:

1) Depth-First Search (DFS)
2) Breadth-First Search (BFS)
3) Depth-First Iterative Search (DFIS)

Explicit Model Checking should be applied for verification of small and practice-orientated models due to its' on-the-fly structure.

The advantage of this approach is particularly obvious. If the model does not fulfill the specified properties, the violation may be faster detected. For instance, if violated property is at very beginning in the hierarchy of the generated states' space in the cache then the technique of the explicit Model Checking must be clever and efficient used.

### 2.1.2. Symbolic Model Checking

The main disadvantage in Model Checking is that this approach suffers from the widely known problem of *state explosion*, i.e. an exponential increase of the verified system in the number of global states with an increasing number of components. This usually occurs in systems with many interacting components or systems that use data structure, which can assume many different values. Taking this fact into consideration, some efforts are made by researchers in order to alleviate the problem of state explosion. Some of these attempts rely upon variations in the logic and methodology, other use effective techniques such as symbolic manipulation in order to explore the state space implicitly [19].

Among these alternatives the primary Model Checking algorithm dealing with state explosion problem is *symbolic Model Checking*. Symbolic Model Checking [10] is an efficient approach that is based on the manipulation of Boolean formulas. Boolean expressions describe the verified system and its behavior symbolically. Symbolic Model Checker avoids building of a state transition graph by using these Boolean formulas and uses OBDD (Ordered Binary Decision Diagram, data structure for representing Boolean functions) [10] representation to perform Model Checking process and to explore the state space implicitly. The behavior of a system is preserved in OBDD while the size of memory requirement can be reduced. Compared to a global state transition graph the Ordered Binary Decision Diagrams permit an efficient handling of the complex systems.

Most Model Checkers like the symbolic Model Checking attempt to reduce memory capacity through the applicable usage of encoding for efficient storage of the systems states.

## 2.2. Model Checking Tools

There are many Model Checking tools available. These Model Checking tools can be used for the verification of different types of systems, such as hybrid, real-time and probabilistic systems. In addition, they can also verify system models described in different programming languages. For instance, SMV Model Checker can verify real-time systems described in its own specification language SV. VCEGAR Model Checker can be used also for verifying real-time systems given in the Verilog code, Java PathFinder is a convenient Model Checker tool for verification system models implemented in Java programming language.

This section will give an introduction to some existing Model Checking tools that can be applied in order to verify DEVS systems and which are based on temporal logic to describe the required specifications on the system. Temporal logic is classified into Computational Tree Logic (CTL) and Linear Tree Logic (LTL) according to the fact whether the time is assumed to have a branching or a linear structure [10]. Thus, the most well-known Model Checkers re defined in the next subsections in order to their practical feasibility by the verification of the real-time systems.

### 2.2.1. SMV

The Symbolic Model Verifier (SMV) [31] is a verification system for hardware design. SMV Model Checker was developed in 1992 by McMillan as part of his doctoral dissertation. The SMV system uses a symbolic Model Checking technique to verify automatically if a hardware design satisfies

specification for a given system [30]. A specification for SMV is a collection of properties that are described in temporal logic: CTL and LTL [42].

The input language of SMV is Synchronous Verilog (SV) and syntactically only a slight variation of the Verilog language. It was actually developed to describe sequential circuits and protocols at various levels of abstraction. Synchronous Verilog is designed to allow the user description of finite state systems that range from synchronous to asynchronous and from detailed to the abstract. It supports interleaving processes with shared variables and allows nondeterminism. SV programming language also provides possibilities for modular hierarchical description and for definition of reusable components.

The SMV Model Checker implements a variety of techniques for compositional methods, allowing the user verification of large and complex systems by reducing the verification problem in small finite state problems that can be solved automatically by the Model Checker. SMV Model Checker provides refinement verification, symmetry reduction, data type reduction and temporal case splitting.

This Symbolic Model Checking tool includes an easy-to-use graphical user interface and source level debugging capabilities.

### 2.2.2. RAVEN

RAVEN [36, 37] is a real-time verification tool, which is extended by the analysis algorithm. The RAVEN model checker was developed by Jürgen Ruf at the University of Tübingen in 1999 to verify timed systems on various levels of abstraction. The RAVEN system models are described as an I/O-interval structure [38] (time-extended finite state machine). The requirement specifications which have to be checked over a given interval structure are formulated in a temporal logic CCTL (clocked CTL, a time extended version of CTL). Defined properties may be verified automatically by RAVEN Model Checking tool.

I/O-interval structures and CCTL formulas are specified in an input language RIL (RAVEN Input Language). The RIL specification [24] includes:

1) Global definitions, for example some fixed time constants and frequently used formulas.
2) I/O-interval specification in parallel executed modules.
3) CCTL formula that describes certain model properties.

To enable the execution of the Model Checking processes in RAVEN the I/O-interval structures are represented as MTBDD (Multi Terminal Binary Decision Diagrams) [36].

This Model Checker provides some pre-defined algorithms [35] for analyzing the specified system: for dead- and live-lock detection, for event occurrences, for inspecting data values, for analyzing critical delay times and for quantitative analysis of the system. By the way, the analyzer of RAVEN permits to find out for example the minimal and maximal assignment of permitted variables without violation of the property.

Simulation and verification results of the Model Checking process are represented in an integrated wave-form browser. If the given systems do not hold on the specified properties, the counterexample generator provides helpful information for error recovering by printing an execution trace of a failed specification in the wave-form browser.

### 2.2.3. VCEGAR

VCEGAR (Verilog Counter Example Guided Abstraction and Refinement) [22] is a Model Checking tool for hardware designs. VCEGAR realizes verification at the RTL (register transfer language) level directly using software verification techniques such as word level predicate abstraction and a refinement loop [11, 23]. The predicate abstraction is employed in the VCEGAR tool in order to reduce the state space explosion during verification process. This is a key technique of the SLAM software verification project [2]. The VCEGAR Model Checker also provides various options for balancing the precision of abstraction and the time required for abstraction computation.

VCEGAR checks safety properties of the given hardware design. The input for this Model Checking tool is a Verilog program and it requires specifications of the given system. If the property of given program is violated a counterexample will be automatically generated showing the execution path, which leads to the properly violation and the right variable assignment.

### 2.2.4. UPPAAL

UPPAAL [26] is a tool for modeling, simulation and verification of real-time systems. It is based on the constraint-solving and explicit verification techniques. The UPPAAL Model Checker is suitable for the verification of systems that can be designed as a collection of nondeterministic processes with finite control structures and real-valued clocks (i.e. Timed Automata). It supports communication of the designed processes through the channels and shared data structures.

The first prototype of the UPPAAL Model Checker, TAB was developed at the University of Uppsala in Sweden in 1993. This tool allowed the verification of simple system properties such as safety and bounded liveness properties. These properties can easily be represented through the state-reachability analysis in a network of Timed Automata. The requirement specification language in UPPAAL is a simplified subset of computational tree logic TCTL (Timed CTL).

In 1995 the Department of Information Technology of the University Uppsala in Sweden in cooperation with the Department of Computer Science and Mathematics of the University Aalborg in Denmark developed a new tool UPPAAL that was extended with a Simulation and Modeling environment. Thus, the UPPAAL Model Checker consists of a *model checking engine* and *graphical user interface* [5], the screenshot of the graphical user interface of the UPPAAL Model Checker is given below in Figure 2.

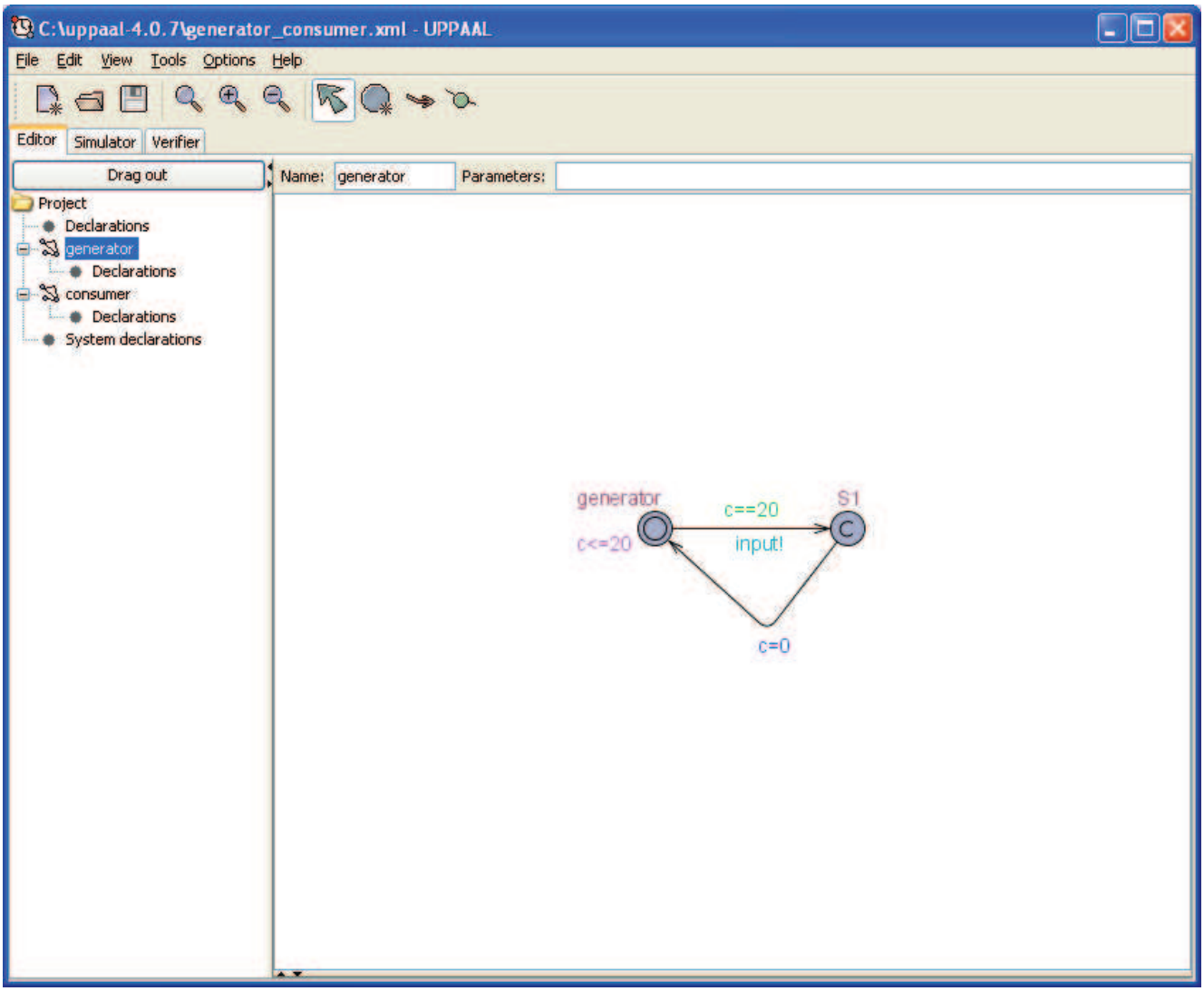


**Figure 2:** Overview of UPPAAL

This extension allows the user to design abstract models of real-time systems (this approach will be discussed in Chapter 3.2) in a graphical interface and to validate the generated models in the simulator immediately. The simulator shows a graphic representation of all of the automata that compose the system model, their control nodes and enabled transitions. It also gives full information about the values and intervals of possible values of all global and local variables and clocks. The simulator allows the user to examine the dynamic behavior of a system model during early design stages from any intermediate point in an interactive fashion, i.e. the simulator explores only a particular execution trace.

In opposite to the simulator in the UPPAAL tool, the verifier checks the whole reachable state space of a system, i.e. it covers the exhaustive dynamic behavior of the system to be verified.

Even more, the UPPAAL Model Checker offers the user to verify arbitrary user defined properties of the system. This Model Checking tool generates automatically a diagnostic trace explaining why a given property is satisfied or why this property is violated from the system description, when verification

fails. The generated diagnostic trace is visualized in the simulator window and it facilitates modeling and debugging processes allowing user to detect easily and to correct founded errors.

Since 1995 the UPPAAL Model Checking tool has been developed permanently. Multiple improvements have been achieved including partial order reduction, symmetry reduction, state space reduction, state space representation, user defined functions, search order and extrapolation. Up the version UPPAAL 4.0 this Model Checker supports XML format for modeling. All features of the UPPAAL Model Checker are perfectly displayed in this format, such elements like templates, locations, edges and labels are described in XML format using tags. The current version of the tool is 4.0.7.

The UPPAAL Model Checker has been successfully used in a number of industrial case studies [6]: Audio/Video Protocol developed by Bang and Olufsen, Collision Avoidance Protocol, Gear-Box Controller, Mutual Exclusion Protocol and some others.

### 2.2.5. Summary

After time-consuming testing and careful examination it has been observed that SMV Model Checker, RAVEN Model Checker and VCEGAR Model Checker are less convenient for the verification of DEVS models than UPPAAL Model Checker. Primary reason is that the previously mentioned Model Checking tools have a bad availability of the documentation and programs. Another reason is that the modeling process is not well supported in this Model Checking tools and the text presentation of automata templates is less intuitive.

All these aspects and facts were taken into account and concerning individual decision, the UPPAAL Model Checking tool is going to be used for the verification of DEVS models for reconfigurable systems, because of its' practical feasibility, efficiency and ease of usage. The next section represents UPPAAL Model Checker more detailed describing specific features and essential prospects for the implementation in this Model Checking tool.

## 2.3. UPPAAL in details

With the respect to the selection criteria given in Chapter 1.2, UPPAAL Model Checker is the most convenient tool for verification of DEVS models. The later investigation resolving the problem of the verification of DEVS models has been done completely with the UPPAAL tool. This attempt is based on the theory of the formal verification performed with Timed Automata and DEVS models as well as the possibility of transformation of Timed Automata into DEVS models and DEVS models into Timed Automata respectively [16, 17].

The complete functional presentation of the UPPAAL Model Checking tool is needed to review the DEVS models in the proposed Model Checker.

### 2.3.1. Structure

The real-time systems are described in the UPPAAL tool as a network of nondeterministic sequential processes [32], which will be explained in Section 2.3.2 more in detail. The processes themselves are

specified as Timed Automata. A Timed Automaton developed by Alur and Dill [1] is an extension of the finite state machines with clocks and data variables. Clocks are represented as real numbers which progress synchronously. The Timed Automaton in UPPAAL consists of a set of *clocks, invariants, variables, guards, synchronization channels, transitions, urgent and committed locations*. In other words Timed Automaton is considered to a finite directed graph annotated with conditions over transitions and locations and resets of non-negative real valued clocks. Locations are represented as vertices in this graph. Exactly one location in the graph is marked as initial node which is pictured by a double circle as it is shown in Figure 3. Locations in the graph may be additionally labeled with invariants. These invariants express constraints on the clock values in order to remain in a particular node. Hence, an invariant is a conjunction of the form $x \leq const$ or $x < const$ , where $x$ is a clock and $const$ is an integer number. An automaton example with three possible dispositions of the invariants expressions is demonstrated in Figure 3. The state *S0* and *S1* are annotated with invariants of the form mentioned above and the state *S2* has no invariant expression. An invariant condition must hold whenever the node belongs to the current state and this must be true throughout the execution. The BNF form of constraints represented by the expressions is given in Figure 26, see Appendix.

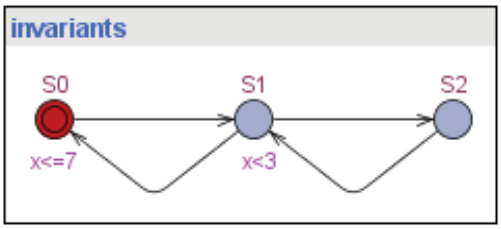


**Figure 3:** Invariants template

### 2.3.2. Types of locations

There are three different types of locations in UPPAAL: *normal* (with or without invariants), *urgent (u)* and *committed (c)* locations without invariants. To illustrate the differences we depict two examples in Figure 4 where *x* denotes the local clocks and *x* in *P0* is different from *x* in *P1*. A normal location *P0.S0* (state *S0* in process *P0*) with an invariant can be left as long as this invariant expression is true and there are possible transitions that can proceed. If multiple transitions can proceed at the same point of time, then the UPPAAL Model Checking tool completes the execution nondeterministically. Unlike to the locations with invariants, the locations without time expressions can be left only if the expression labeled on the outgoing edge becomes true. If both, invariant and transition expression are present, both have to be true.

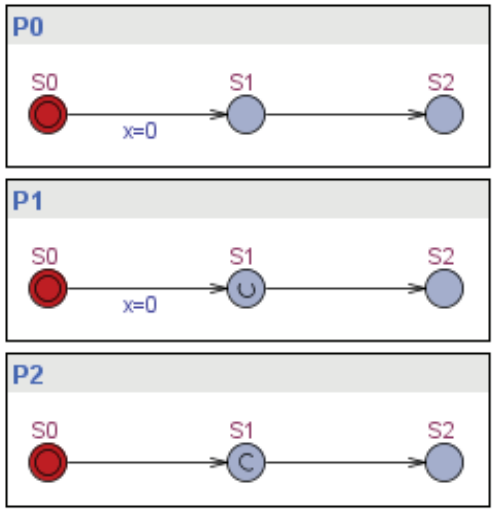


**Figure 4:** Automata with normal, urgent and committed locations

Time passing is not allowed when a process is in *urgent location* (e.g. *P1.S1*), i.e. the location must be immediately left as soon as one probable outgoing transition can execute. In other words transitions that are available to this moment can fire, i.e. an interleaving with normal states is allowed. An example in Figure 4 above shows that being in the state *P1.S1* all active transitions can proceed.

A *committed location* is more restrictive than the urgent location. If any process is in a control node labeled as being committed this node must be left immediately by invoking an outgoing edge from this location, i.e. no delay is allowed to occur. For example, if the state *P2.S1* is active the only possible transition that can be fired is the outgoing edge to its successor state *P2.S2*.

Control nodes in the automata template are connected by edges. Each edge in the automaton is annotated with four types of labels: *selections, guards, synchronization* and *updates*. All these types of labels are optional [5]. In the following a short description of the most important and useful UPPAAL elements for DEVS representation is given in the next subsections.

### 2.3.3. Guard expression

A *guard* is similar to an invariant, i.e. a guard is a particular expression referring to clocks and integer variables that must be satisfied in order for the transition to be taken. It is not obligatory that the transitions must be fired. Assuming that clocks of the system start at 0 and the time proceed at the same rate and at any time an invariant condition of some automata template is true (if available) as well as the guard expression on the value of particular clock must be true then the outgoing edge can be fired. But if at the same time the guard expression on the integer variable in any other or just in the same automata template is also becoming true, then the first one can be canceled. An example of automata using guard expressions is given in Figure 5. Transitions without synchronization channel can proceed, when the guard expression is true and as long as none of the invariants of the successor nodes are violated. The abstract syntax of guard expressions in BNF form is given in Figure 26, see Appendix.

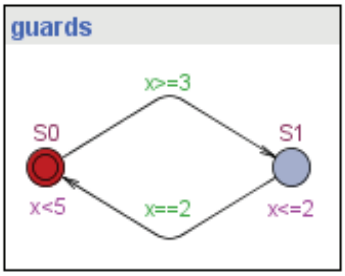


**Figure 5:** Template with guards' expressions

### 2.3.4. Synchronization channels

A *synchronization label* is either of the form *channel!* or *channel?*, where *channel!* is on the emitting side and *channel?* is on the receiving side. Actually, a channel fires when its transition labeled with *channel!* is taken. In both cases *channel!* and *channel?* represents an arbitrary name. The expression on the transitions labeled with synchronization channel must be side-effect-free, evaluate to a channel and refers to integer numbers, constants and the channel itself. Synchronization channels are used to enable the communication between the processes in the system composed of them. Transition *t1* labeled with synchronization channel *channel!* can be fired if there exists another transition *t2* with the complementary labeled *channel?* and the guards of both are true as well as successor invariants are not violated after execution. Thus, the transition from *channel1.S1* to *channel1.S2* is fired as shown in Figure 6, because there exists a complementary labeled transition from *S0* to *S1* in the process **channel2**.

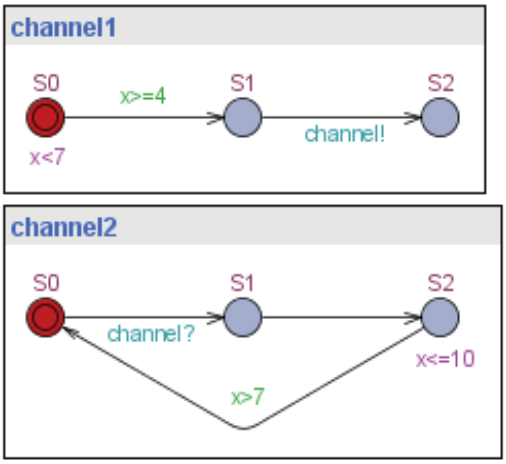


**Figure 6:** Two automata with a synchronization channel

### 2.3.5. Update labels

An *update label* in UPPAAL is a comma separated list of expressions on the transitions, e.g. a reset of clocks $x := 0$ or an assignment of the form *i:=expr*, where *i* is an integer variable or an element of array and the right side of assignment is an integer expression. A very important detail of the assignments is that they do not evaluate concurrently, i.e. on the synchronization edges the assignment on the sending side (*channel!*) evaluates before the receiving side (*channel?*). The example in Figure 7 shows

possible usage of the update labels in the automata template **update1** and **update2** respectively. Being in the state *S0* in the process **update1** it proceeds to its successor state *S1* if the guard expression on the taken transition and invariant expression of the state *S1* are true. If the transition from *update1.S0* to *update1.S1* is taken the clock value will be updated and reset to 0: $x=0$. Further, the state *S1* in the process **update1** transits to the state *S2* taking a transition labeled with synchronization channel *increment!*, at the same point of time the integer variable *n* is updated and increased. After taking this transition the synchronization channel *increment* fires and the complementary edge annotated with *increment?* is activated. Consequently, the state *S0* in the process **update2** changes to the state *S1* receiving updated integer variable *n* with the current assigned value. If *n* is less than 3 ($n<3$) the guard condition on the transition from *S1* to *S0* is true and the state *update2.S1* proceeds to *update2.S2*. If integer variable *n* is greater or equal than 3 the state *update2.S1* transits to *update2.S2*. When the state *S2* in the process **update2** is left, *n* is reset to 0: $n=0$.

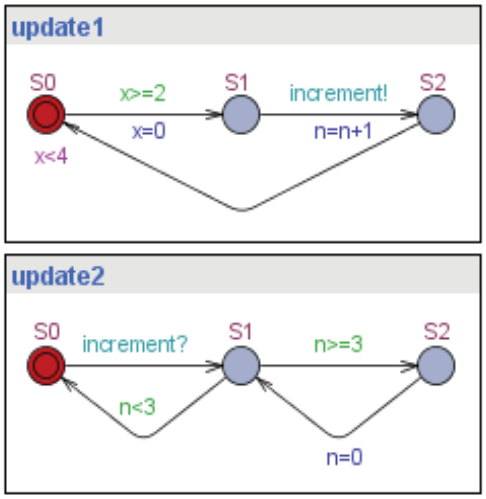


**Figure 7:** Automata with update labels

### 2.3.6. Multiple Structure

As it was mentioned above, UPPAAL enables the user to model reconfigurable hardware systems as a network of timed finite state automata with global or local clocks and variables. Practically it is also possible to declare global synchronization channels. Global variables can be modified by any instance of any automaton in the model. Global clocks can be reset or assigned to any natural number by any automaton. In contrast to global clocks, local clocks can be reset or assigned to any natural number by related automaton. In general, these clocks are aligned to each other and progress synchronously, i.e. all clocks of a system start at the same instant from their initial value and they proceed at the same rate. The value of the clocks can be compared to natural numbers during the system execution.

Figure 8 depicts timing behavior of a system defined by two processes **A** and **B** with control nodes {*A0, A1, A2, A3*} and {*B0, B1, B2, B3*} respectively. The system model contains two clocks *a* and *b*, a global integer variable *n* and one synchronization channel *ab*. Automaton **A** has local a clock *a* and emits on the synchronization channel via *ab!*. The initial node *A0* of the automaton **A** is annotated with the invariant: $a<=6$. The outgoing edge is marked with the guard expression: $a>=3$. In consequence the

control node *A0* can remain as long as the time $a$ is not greater than 6 and the edge from *A0* to *A1* can be fired if $a$ is greater or equal 3. This means that this state will be left somewhere in the time range $3 \le a \le 6$. Automaton **B** has local a clock $b$ and receives the synchronization channel *ab?*. The initial node *B0* in the automaton **B** is annotated with the invariant: *b<=4*. The outgoing edge is marked with the guard expression: *b>=2*. Thus, **B** can remain in *B0* as long as the value of $b$ is not more than 4 and the edge from *B0* to *B1* can be fired if $b$ is greater or equal 2. As long as none of the invariants of the control nodes in the current state are violated, time may progress without affecting the control nodes vector and with all clock values incremented with the elapsed duration of time. In Figure 8, from the initial state ((*A0*, *B0*), *a=0*, *b=0*, *n=0*) time may elapse for a minimum 3 time units leading to the state ((*A0*, *B0*), *a=3*, *b=3*, *n=0*). However, time cannot elapse 5 time units as this would violate the invariant of *B0*.

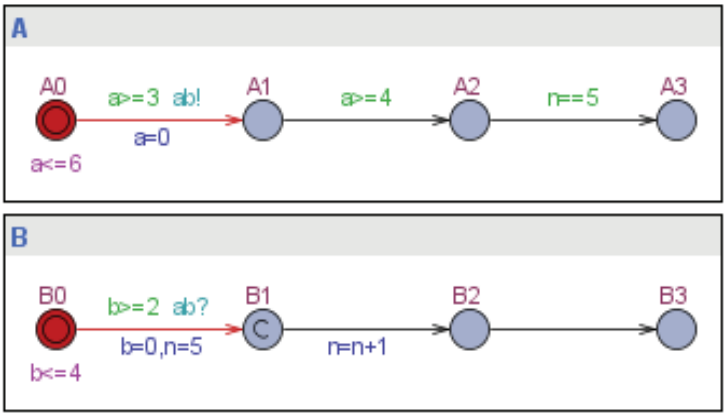


**Figure 8:** System model composed from two automata templates

The process **A** is synchronized with process **B** via the synchronization channel *ab*. This means that two edges of two different components, which are enabled in a state can synchronize. Here the process **A** is synchronized with the process **B** via synchronization channel *ab* and the edge from *A0* to *A1* in the process **A** is complementary to the edge between *B0* and *B1* in the template **B**. It follows that in the state ((*A0*, *B0*), *a=3*, *b=3*, *n=0*) the processes synchronize through channel *ab* leading to the new state ((*A1, B1*), *a=0*, *b=0*, *n=5*), clocks and variable $n$ have been appropriately updated: clocks are reset to 0 after taking appropriate edges and integer variable $n$ is assigned with 5, see Figure 8. The process **B** has an internal edge from *B1* to *B2* enabled, the edge can be taken without any synchronization, notice that the state *B1* is marked as a committed location. Thus, in the state ((*A1*, *B1*), *a=0*, *b=0*, *n=5*) the component **B** leads to the state ((*A1*, *B2*), *a=0*, *b=0*, *n=6*) without any time delay. The integer variable $n$ is accordingly incremented when the edge between *B1* and *B2* is taken.

# 3. DEVS Models and UPPAAL tool

Formal representation of the specifications and models is needed to facilitate the verification process. The DEVS formalism [42] was first introduced by Zeigler as a foundation for a high-technology systems design methodology. This formalism is well established and represents behavior of real-time systems. Reconfigurable systems are supported with the RecDEVS extensions [28].

This chapter starts with recalling the basic notions of the DEVS formalism. After detailed definition of parallel DEVS this chapter continues with describing the transformation of DEVS models into UPPAAL representation.

## 3.1. Parallel DEVS

Parallel Discrete Event System Specification (DEVS) [42] is one of the discrete types of the system specification formalism that is used by engineers to build an abstract simulation model of the system. The formalism of parallel DEVS is the most convenient one for a possible extension towards reconfigurable systems employing reconfigurable devices such as FPGAs, which basic idea is reusing the computational components for independent computations and using multiplexers to control the routing between these components [14].

The parallel DEVS formalism is an extension of the classical DEVS. This extension allows to describe hierarchical composition of the system defying it through multiple parallel modules and it also permits the concurrent execution of these modules. Basic parallel DEVS models unlike classical DEVS have a bag of inputs on the external transition function. $X^b$ is a set of bag over elements in $X$ with their multiple occurrence, e.g. $\{a, b, c, a, b\}$ is a bag. In the following a short review of the most important elements of DEVS that are relevant for design of reconfigurable systems is given.

Thus, a basic parallel Discrete Event System Specification is a structure of DEVS system specification $N$ and a set of model hierarchy $M$.

$$M = (X_M, Y_M, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta)$$

*with*

$X$ : *set of inputs*

$Y$ : *set of outputs*

$S$ : *set of sequetial states*

$s_0 = initial\ node$

$\delta_{int} : S \to S$ *is the internal transition function*

$\delta_{ext} : Q \times X_M^b \to S$ *is the external state transition function, with*

$Q = \{(s,e) \mid s \in S, 0 \le e \le ta(s)\}$ *is the set of total states*

$e$ *is elapsed time since the last state transition*

$\delta_{con} : S \times X^b \to S$ *is the confluent transition function, subject to* $\delta_{con}(s,\varphi) = \delta_{int}(s)$

$\lambda : S \to Y$ *is the output function, specifies the output values due to internal transition function*

$ta : S \to \mathbb{R}^+_{0,\infty}$ *is the time advance function*

$X^b$ : *bag of* $X$

$M = set\ of\ Models\ hierarchy$

*Elems* is a set of hierarchical DEVS system specifications and/or atomic DEVS components. This DEVS components can be linked together by the port coupling *Conns*, which is a set of tuples $(x, y)$. It connects ports from the set of input ports $x \in X_M$ with corresponding output ports $y \in Y_M$, respectively with the global ports $X_{ext}$ and $Y_{ext}$.

Three types of the state transition functions are described in DEVS formalism: internal transition function $\delta_{int}(s)$, external transition function $\delta_{ext}(s,x,e)$ and confluent transition function $\delta_{con}(s,x,e)$.

*Internal transition*: if the timeout $\tau(s)$ is elapsed, i.e. the time has arranged up to the timeout defined by $\tau(s)$, the internal transition $\delta_{int}(s)$ will proceeded, i.e. the state $s$ transit to $s'$ at the time $\tau(s)$. The state $s'$ denotes the next state after execution of an internal transition function, i.e. $\delta_{int}(s) := s'$. The elapsed time will be reset to 0 and an output event $\lambda(s)$ is generated.

*External transition*: iff the timeout $\tau(s)$ of the state does not occur and the component receives an input event, the external state transition $\delta_{ext}(s,x,e)$ will be executed. Here $\boldsymbol{e}$ is an amount of time that has been elapsed since the last input event, where $0 \le e < \tau(s)$ and the state will transit from $s$ to $s''$. The next state when an external transition is executed is denoted by $s''$, i.e. $\delta_{ext}(s,x,e) := s''$. The output function $\lambda(s)$ is not generated by an external transition.

*Confluent transition*: proceeds iff the state receives an external event at the time of its internal transition (the timeout $\tau(s)$ is occurred), i.e. $\delta_{con}(s,x,e)$ evaluate to some next state $s'''$. Where $s'''$ denotes the next state when the confluent transition function proceeds, i.e. $\delta_{con}(s,x,e) := s'''$. Confluent transition resolves the collision problem of the two transition functions: $\delta_{int}$ and $\delta_{ext}$ that

are possible at the same point of time, i.e. confluent transition function decides which of the possible transition functions will execute if they occur at the same time.

## 3.2. Design of parallel DEVS in UPPAAL

This section presents the exhaustive transformation and design of parallel DEVS models defined in Chapter 3.1 above into UPPAAL tool.

Each Model $M$ from the formal DEVS definition is mapped to one UPPAAL model. The set of states $S$ of DEVS is graphically reflected in UPPAAL as a set of states of the finite directed graphs, where $s_0$ is the initial node. Each node can have an optional name, i.e. this can be a consecutive enumeration or explicit name due to the performing task. Consider the UPPAAL model of Figure 8 introduced in Section 2.3.2, where {*A0, A1, A2, A3*} is a set of states and *A0* is an initial node in the finite directed graph **A**.

### 3.2.1. Timeout representation

For the representation of three main types of DEVS functions: $\delta_{int}$, $\delta_{ext}$ and $\delta_{con}$, primarily the representation and explanation of the timeout execution in UPPAAL is of note. Before starting with the representation of these main types of DEVS functions, foremost, the timeout function in UPPAAL must be represented and explained. Figure 9 illustrates the representation of the timeout function $\tau(s)$ in the UPPAAL Model Checking tool. Assuming that $\tau(S1) = ta$ time units, it works as follows: the state *S1* is annotated with the invariant $x \leq ta$, where $x$ is local clock and $ta$ is an upper bound (at least after $ta$ time units the state *S1* must be left); the transition from *S1* to *S2* has a guard expression $x == ta$, when the guard condition is true the edge is taken and the timeout $\tau(S1)$ is occurred, meaning that the timeout occurs not before and not after when the guard condition is true and the edge is taken. The syntax of the invariant expression in UPPAAL is very strict; only two possibilities of the time expressions are allowed to be used to influence the behavior of the system, see Section 2.3.1. Thus, an invariant of the form $x = ta$ is disallowed to be used. That's why the execution of the timeout function is represented as combination of the invariant and guard expressions.

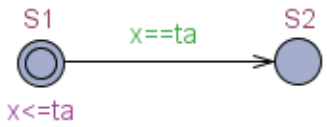


**Figure 9:** Timeout representation

The UPPAAL tool facilitates the implementation of the indeterminism of equal timeouts in different models. This proceeds automatically. The UPPAAL selects one of the possible transitions in the simulation phase, either the transition in the process **P0** from *S1* to *S2* or the transition in the process **P1** from *S1* to *S2*, see Figure 10.

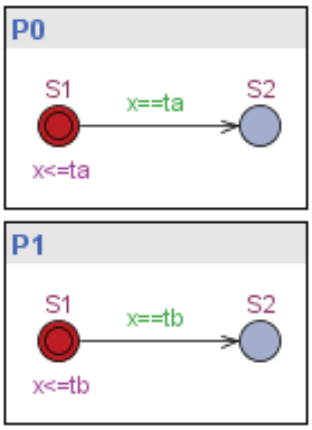


**Figure 10:** Indeterminism of equal timeouts in different automata templates

At this point it might be helpful to analyze two special variants of the timeout: what happens if the lifetime of a state is 0? And what happens when the timeout is equal infinity? If $\tau(s) = 0$ then the location must be explicitly marked as committed one symbolizing that no time delay is allowed. In this case an internal transition is immediately taken when entering this state. In Figure 11 the state *S2* is marked as committed location and its timeout is equal 0, whereas the timeout of the state *S1* is: $\tau(s) = ta$ .

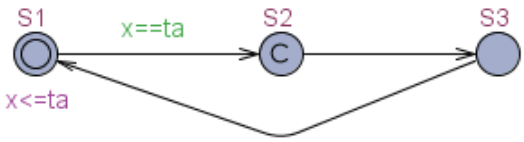


**Figure 11:** Timeout of *S2* is equal 0: *τ(S2)=0*

The initial node can be also marked as committed one, see Figure 12. The number of committed locations in an UPPAAL model is not restricted, but it has to be proved if it is worthwhile in the investigated implementation.

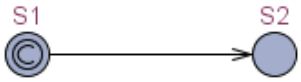


**Figure 12:** Committed initial node

If the lifetime of a state is ∞ the model stays in its current state until an external event occurs. This example is shown in Figure 13 below. The process **P0** stays in the state *S1* as long as the synchronization channel *input?* is not activated through a related edge between the states *S5* and *S6* in the process **P1**.

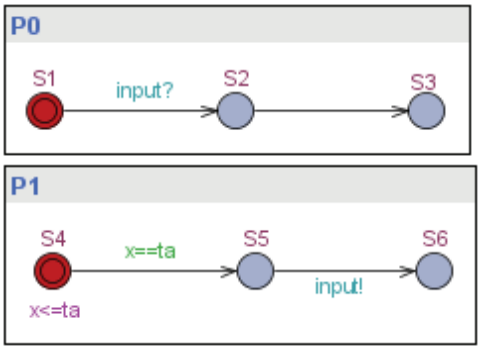


**Figure 13:** Timeout is equal infinity ( $\tau(S1)=\infty$)

Knowing how to achieve the execution of the timeout function in the UPPAAL Model Checking tool, the representation of major DEVS functions can be further introduced.

### 3.2.2. Representation of the internal transition function

Figure 14 illustrates the transformation of the DEVS internal transition $\delta_{int}$ with $\tau(s)=10$ defined in Chapter 3.1 into the UPPAAL tool. According to the DEVS definition the internal transition proceeds when the timeout $\tau(s)$ occurs, so the execution of an internal transition function is identical to the timeout execution. If the guard condition $x==10$ is true and the upper bound of the invariant of the state $s$ is 10, then timeout $\tau(s)=10$ is occurred and the state $s$ transits to the next state $s'$, i.e. $\delta_{int}(s):=s'$.

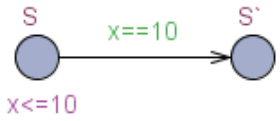


**Figure 14:** Internal transition function

A design sample of the internal transition with an output function $\lambda(s)$ is given in Figure 15. When the edge from $s$ to $s'$ labeled with *input!* is fired, the output function $\lambda(s)$ transmits some information to another automata, e.g. $msg=5$ - transmitted information.

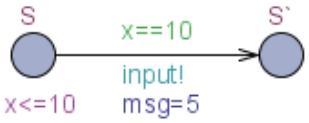


**Figure 15:** Internal transition function with output function λ(S)

### 3.2.3. Representation of the external transition function

The design of the external transition function in the UPPAAL Model Checking tool is coherent with the usage of synchronization channels, see Figure 13. The timeout of the states remains the same, $\tau(s)=10$. If the timeout is not elapsed, i.e. $\tau(s)<10$ and the state $s$ receives an external event, then

the external transition triggers, i.e. $\delta_{ext}(s,x,e) := s"$. It is evident that the guard condition must be less than the timeout $x<10$, more precisely less than the state invariant: $x=10$, as an internal transition has to proceed for $x \geq 10$ see Section 3.2.2. Figure 16 represents the design of the external transition function in UPPAAL. When the guard is true and the state $s$ becomes an external event through synchronization channel *input?*, which is activated by the transition with a complementary label *input!*, then the external transition is occurred and the state $s$ transits to $s"$, where it will remain either to the next external event or $\tau(s")$. Disadvantage of this construction is that it is impossible to determine in UPPAAL the elapsed time $e$ from the last input event.

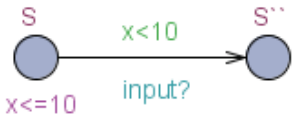


**Figure 16:** External transition function

### 3.2.4. Representation of the confluent transition function

With the knowledge of the implementation of the execution construction of internal and external transitions it is not difficult to construct the confluent transition function. The confluent transition occurs iff the state $s$ receives an external event signalized by *input?* at the time of its internal transition. So, if the timeout $\tau(s)=10$ in the state $s$, as shown in the Figure 17 below, elapsed and the guard conditions on both outgoing edges from $s$ becomes true $(x==10)$, it seems that the transition from $s$ to $s'$ can actually be fired as well as the transition from $s$ to $s"'$. But the edge from $s$ to $s"'$ labeled with the synchronization channel *input?* has priority over the edge from $s$ to $s'$ without synchronization channel. Thus, the external event is activated by the synchronization channel *input?* and then the state $s$ changes to $s"'$, see Figure 17. This implementation of the confluent function in the UPPAAL tool allows to check the local time of the system model by means of guard conditions on the appropriate edges. According to this time it can differ whether a timeout is elapsed or not.

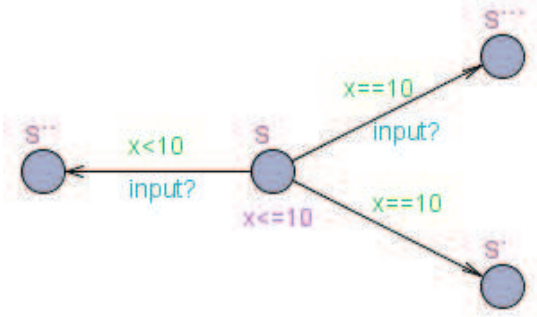


**Figure 17:** Confluent transition function

### 3.2.5. Representation of the multiple message transition

Another significant construction is visualized in Figure 18 that explains how to transmit multiple messages in the network of multiple automata composing the system model. Assume for the moment

that a certain model component is going to concurrently communicate with two other automata components, so it sends some piece of information to both components. To enable the implementation of this transmission additional transitive states are required, because UPPAAL supports only one synchronization channel on each transition. These transitive states should be marked as committed locations. Consider the given pattern state $s$ changes to $s'''$, while sending the message information to another model components via the synchronization channels *input1!* and *input2!* respectively. Started in the node $s$ one of the possible two edges can be fired, this proceeds nondeterministically in the UPPAAL Model Checker. If the edge labeled with *input1!* is taken first, then $s$ proceeds to $s'$. The state $s'$ is marked as committed location where no time delay is allowed to occur, i.e. the state $s'$ will be left immediately taking the outgoing transition labeled with *input2!*. If taking the edge labeled with *input2!* first, state $s$ transit to $s''$ and leaves it immediately by taking the outgoing edge with the synchronization channel *input1!* without any time delay. This construction can be applied into any model with multiple automata components using an appropriate number of intermediate states and synchronization channels.

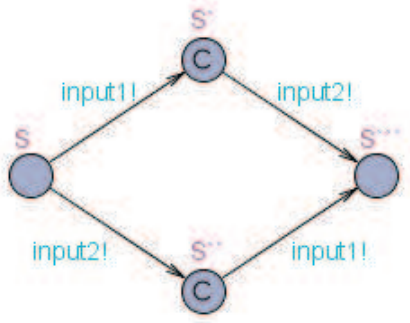


**Figure 18:** Multiple message transmission

So far, the most essential and fundamental features of parallel DEVS models were introduced as well as the exemplified implementation of these features in the UPPAAL Model Checking tool. Relying on the proposed features a complex application based on the AutoVision example, which is going to be used for the verification, is presented in the next section.

## 4. Example

For a better comprehension and demonstration of the verification capabilities of DEVS models within UPPAAL, a complex application was implemented, which is based on the AutoVision example [13]. The model of the AutoVision system consists of several distinct components for vision enhancement and for automated recognition aimed to a driving assistance scenario. Figure 19 gives an overview over the following components and their interaction: *Sensor*, *Shape*, *Contrast* and *Taillight*.

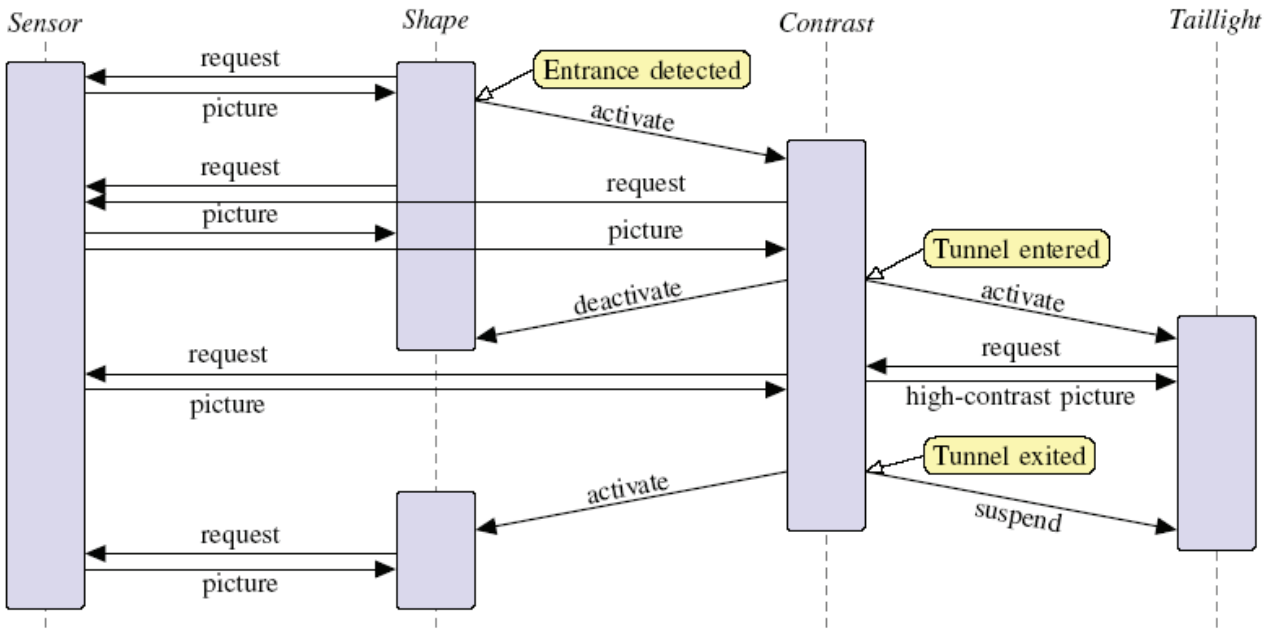


**Figure 19:** Object Diagram of the AutoVision Example

*ROI*: provides pictures to all requesting components: *Shape* and *Contrast*.

*Shape*: performs picture requests and scans the result for important shapes. If the shape of a tunnel entrance is found, then the *Contrast* component is invoked.

*Contrast*: enhances the *Sensor* picture and recognizes, when the car enters or leaves a tunnel, in which case it activates or suspends other components: *Shape* or *Taillight*. If the car is in the tunnel and the picture is dark, *Contrast* suspends the *Shape* component and activates *Taillight*. When the car is in tunnel and the picture is bright meaning the end of the tunnel, *Contrast* component resolves the *Taillight,* if this was active before and invokes the *Shape* component.

*Taillight*: provides object information to the driver based on taillight traces. It operates when the car is inside the tunnel where it is too dark for the *Shape* component to operate properly.

All four components run concurrently allowing the evaluation of the parallel execution behavior inside UPPAAL. By using different request intervals for *Shape* and *Contrast* components, the situation of multiple competing request arriving at the same time can be modeled, which is a very interesting aspect for verification.

Through the brightness of the picture the *Contrast* component recognizes if a tunnel is left or not. When the car has left the tunnel, *Contrast* deletes the *Taillight* component, activates a new *Shape* and finally deletes itself.

Another additional component, the *Testbench* was designed enabling the AutoVision simulation process in UPPAAL. In some sense, the *Testbench* simulates the behavior of the real-time environment for the designed model, so that manual control of the components falls into disuse. The *Testbench* is an independent and self-consistent entity, whose execution is not addicted on other components. The *Testbench* component replicates car driving inside and outside the tunnel and passes information to the *ROI* component about external environment. Configuration and performance of the *Testbench* component are represented in Figure 20.

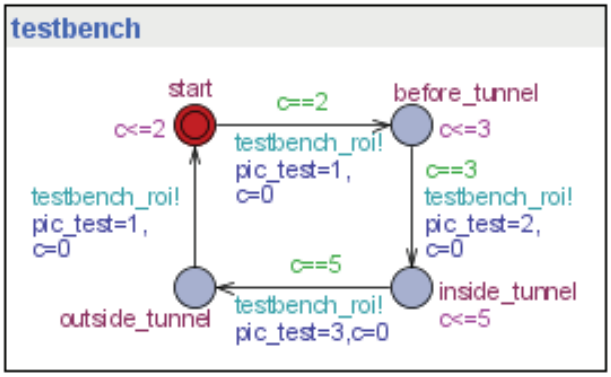


**Figure 20:** Testbench component in the AutoVision Example

All four components from the AutoVision system model are represented as single DEVS structure. The following subsections give a formal definition of these components and afterword their graphical representation in the UPPAAL tool.

### 4.1. ROI Component

$$ROI = (X, Y, \delta_{ext}, \delta_{int}, \delta_{con}, ta)$$
$$S = (\text{"}Waiting\text{"}, \text{"}SendPicture\text{"}, \text{"}RoiShape\text{"}, \text{"}RoiContrast\text{"}) \times source$$
$$X^b = X_{Shape} \times X_{Contrast} = \{request\} \times \{request\}$$
$$Y^b = Y_{Shape} \times Y_{Contrast} = \{picture\} \times \{picture\}$$
$$s_0 = (\text{"}Waiting\text{"}, \varnothing)$$

$$\delta_{int}(\text{"}SendPicture\text{"}, shape) = (\text{"}ROIShape\text{"}, \varnothing)$$
$$\delta_{int}(\text{"}SendPicture\text{"}, contrast) = (\text{"}ROIContrast\text{"}, \varnothing)$$
$$\delta_{int}(\text{"}ROIContrast\text{"}, \varnothing) = (\text{"}Waiting\text{"}, \varnothing)$$
$$\delta_{int}(\text{"}ROIShape\text{"}, \varnothing) = (\text{"}Waiting\text{"}, \varnothing)$$

$$\delta_{ext}((\text{"}Waiting\text{"}, i), e), msg) = \begin{cases} (\text{"}SendPicture\text{"}, shape) & if\ \ msg = (request, \Diamond) \\ (\text{"}SendPicture\text{"}, contrast) & else \end{cases}$$
$$\delta_{con} = \delta_{ext}$$

$$ta(s', i) = \begin{cases} \infty & if\ \ s' = \text{"}Waiting\text{"} \\ T_{refresh} & if\ \ s' = \text{"}SendPicture\text{"} \\ 0 & else \end{cases}$$

$$\lambda(\text{"}ROIShape\text{"}, \varnothing) = (picture, \Diamond)$$
$$\lambda(\text{"}ROIContrast\text{"}, \varnothing) = (\Diamond, picture)$$

The graphical UPPAAL representation of the *ROI* component from the AutoVision example is depicted in Figure 21.

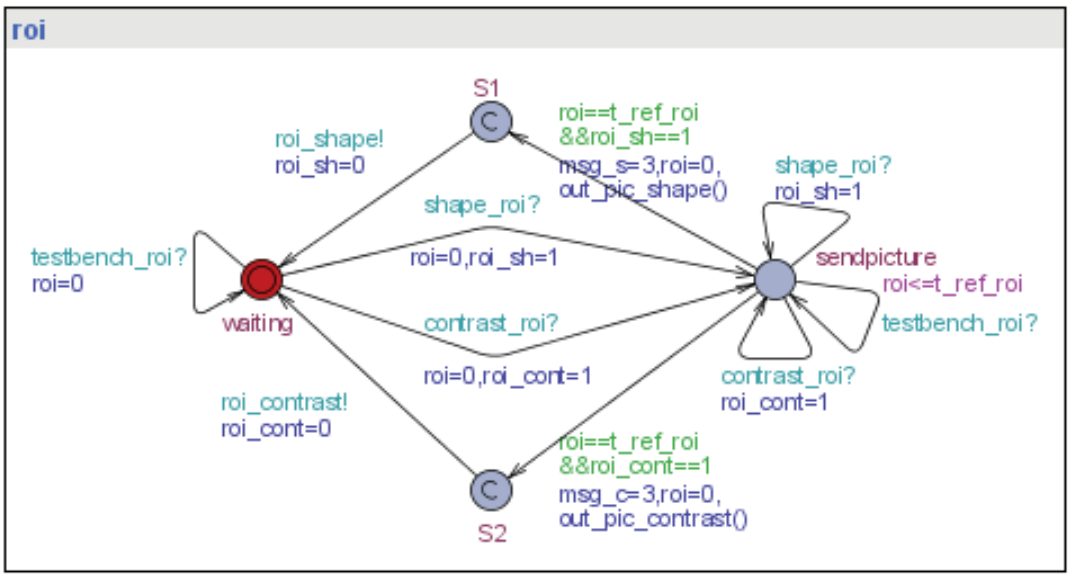


**Figure 21:** UPPAAL representation of the *ROI* component

### 4.2. Contrast Component

$$Contrast = (X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

$$S = (\text{"}Idle\text{"}, \text{"}Action\text{"}, \text{"}Entrance\text{"}, \text{"}TaillightRequest\text{"}, \text{"}Exit\text{"})$$

$$X = X_{ROI} \times X_{Taillight} = \{\text{"}picture\text{"}\} \times \{request\}$$

$$Y = Y_{Taillight} \times Y_{Shape} \times Y_{ROI} = \{activate, shutdown, picture\} \times \{activate, shutdown\} \times \{request\}$$

$$s_0 = (\text{"}Idle\text{"}, 0, 0, 0)$$

$$\delta_{int}(s') = \begin{cases} (\text{"}Entrance\text{"}, picture) & if\ s' = \text{"}Action\text{"} \text{ and entered tunnel} \\ (\text{"}Exit\text{"}, picture) & if\ s' = \text{"}Action\text{"} \text{ and if leaving tunnel} \\ (\text{"}Idle\text{"}, picture) & if\ s' = \text{"}Action\text{"} \text{ and if no changes} \\ (\text{"}Idle\text{"}, picture) & else \end{cases}$$

$$\delta_{ext}((Idle, oldpicture), e), msg) = \begin{cases} (\text{"}Action\text{"}, picture) & if\ msg = (picture, \Diamond) \\ (\text{"}TaillightRequest\text{"}, oldpicture) & if\ msg = (\Diamond, request) \end{cases}$$

$$\delta_{con} = ((s, e), x) = \delta_{ext}((\delta_{int}(s), e), x)$$

$$ta(s) = \begin{cases} T_{refresh} & if\ s = \text{"}Idle\text{"} \\ 0 & else \end{cases}$$

$$\lambda(s, picture) = \begin{cases} (picture, \Diamond, \Diamond) & if\ s = TaillightRequest \\ (\Diamond, \Diamond, request) & if\ s = Idle \\ (activate, shutdown, \Diamond) & if\ s = Entrance \\ (shutdown, shutdown, \Diamond) & if\ s = Exit \end{cases}$$

Figure 22 shows the design of the *Contrast* component in UPPAAL.

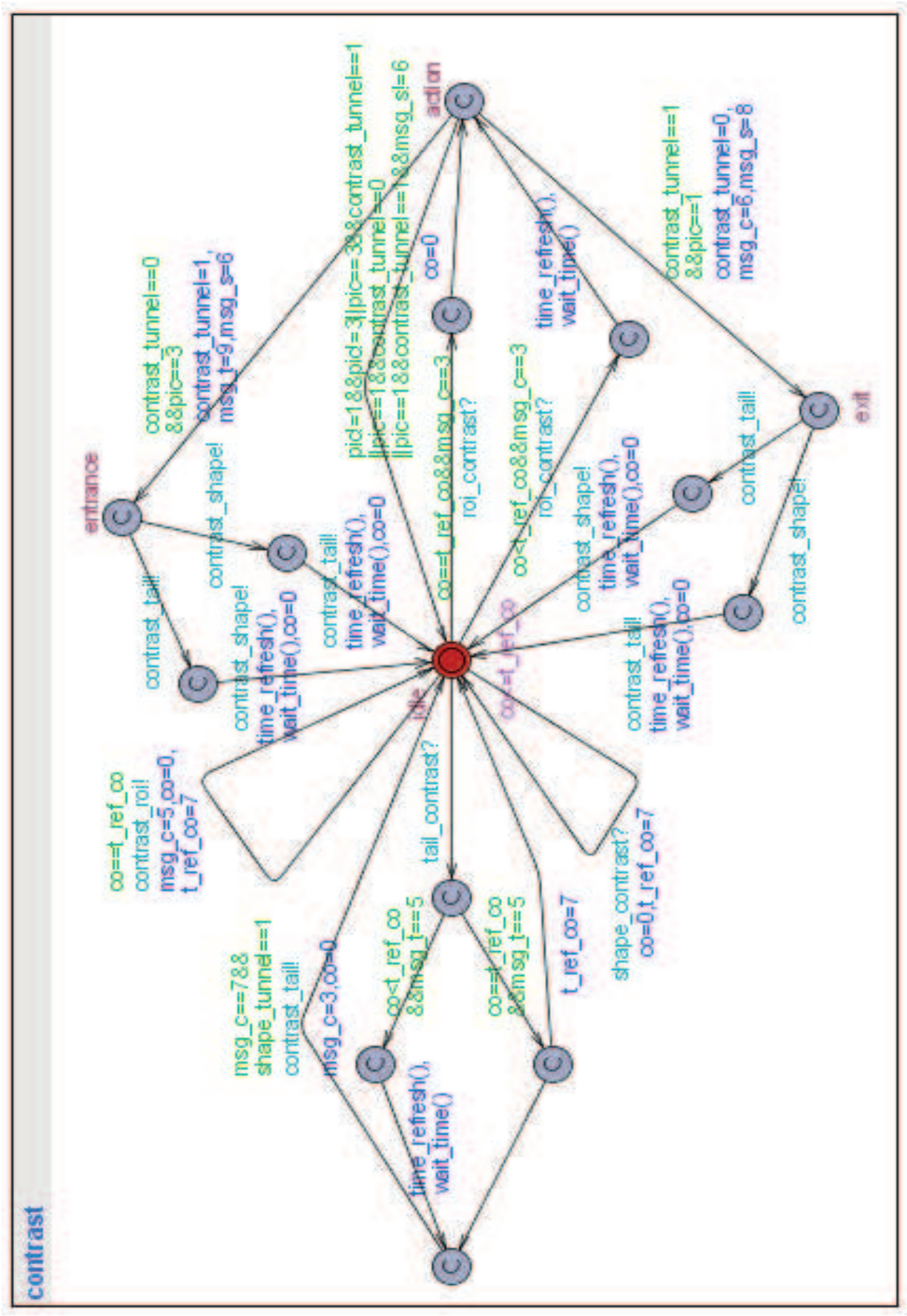


**Figure 22:** UPPAAL representation of the *Contrast* component

### 4.3. Taillight Component

$$Taillight = (X, Y, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

$$S = ("Deleted","Idle","Action","Send")$$

$$X = X_{contrast} = \{activate, shutdown, picture\}$$

$$Y = Y_{contrast} = \{request\}$$

$$s_0 = "Deleted"$$

$$\delta_{ext}((Idle, e), msg) = \begin{cases} "Action" & if\ \ msg = picture \\ "Deleted" & if\ \ msg = shutdown \end{cases}$$

$$\delta_{ext}((Deleted, e), activate) = Idle$$

$$\delta_{int}(Idle) = Send$$

$$\delta_{int}(Send) = Idle$$

$$\delta_{con} = \delta_{ext}$$

$$ta(s) = \begin{cases} T_{refresh} & if\ \ s = "Idle" \\ \infty & if\ \ s = "Deleted" \\ 0 & else \end{cases}$$

$$\lambda("Send") = request$$

The UPPAAL representation of the *Taillight* component is given in Figure 23.

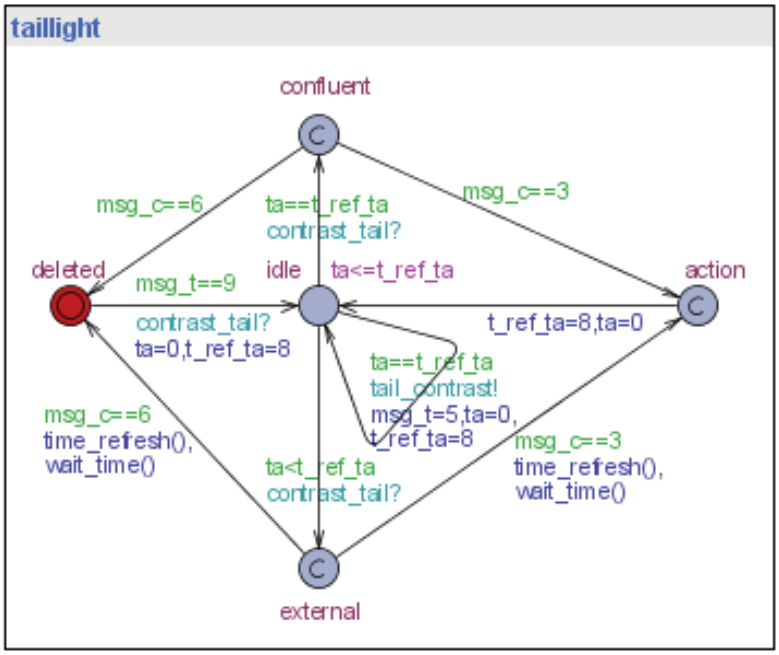


**Figure 23:** UPPAAL representation of the *Taillight* component

### 4.4. Shape Component

$$Shape = (X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$
$$S = (\text{"}Idle\text{"}, \text{"}Action\text{"}, \text{"}Entrance\text{"}, \text{"}SendRequest\text{"}, \text{"}Deleted\text{"})$$
$$X = X_{ROI} \times X_{Contrast} = \{picture\} \times \{activate, shutdown\}$$
$$Y = Y_{ROI} \times Y_{Contrast} = \{request\} \times \{activate\}$$
$$s_0 = (\text{"}Idle\text{"})$$
$$\delta_{int}(s) = \begin{cases} \text{"}SendRequest\text{"} & if\ s = \text{"}Idle\text{"} \\ \text{"}Idle\text{"} & if\ s = \text{"}Action\text{"}\ and\ no\ tunnel\ detected \\ \text{"}Entrance\text{"} & if\ s = \text{"}Action\text{"}\ and\ tunnel\ detected \\ \text{"}Idle\text{"} & if\ s = \text{"}Entrance\text{"} \end{cases}$$

$$\delta_{ext}(((\text{"}Idle\text{"}), e), msg) = \begin{cases} \text{"}Deleted\text{"} & if\ msg = (\Diamond, shutdown) \\ \text{"}Action\text{"} & elsif\ msg = (picture, \Diamond) \end{cases}$$
$$\delta_{ext}(((\text{"}Deleted\text{"}), e), (\Diamond, activate)) = \text{"}Idle\text{"}$$

$$\delta_{con}((s, e), x) = \delta_{ext}((\delta_{int}(s), e), x)$$

$$ta = \begin{cases} T_{refresh} & if\ s = (\text{"}Idle\text{"}) \\ \infty & if\ s = (\text{"}Deleted\text{"}) \\ 0 & else \end{cases}$$

$$\lambda(s) = \begin{cases} (request, \Diamond) & if\ s = \text{"}SendRequest\text{"} \\ (\Diamond, activate) & if\ s = \text{"}Entrance\text{"} \end{cases}$$

The graphical representation of the Shape component in the UPPAAL Model Checker is shown in Figure 24.

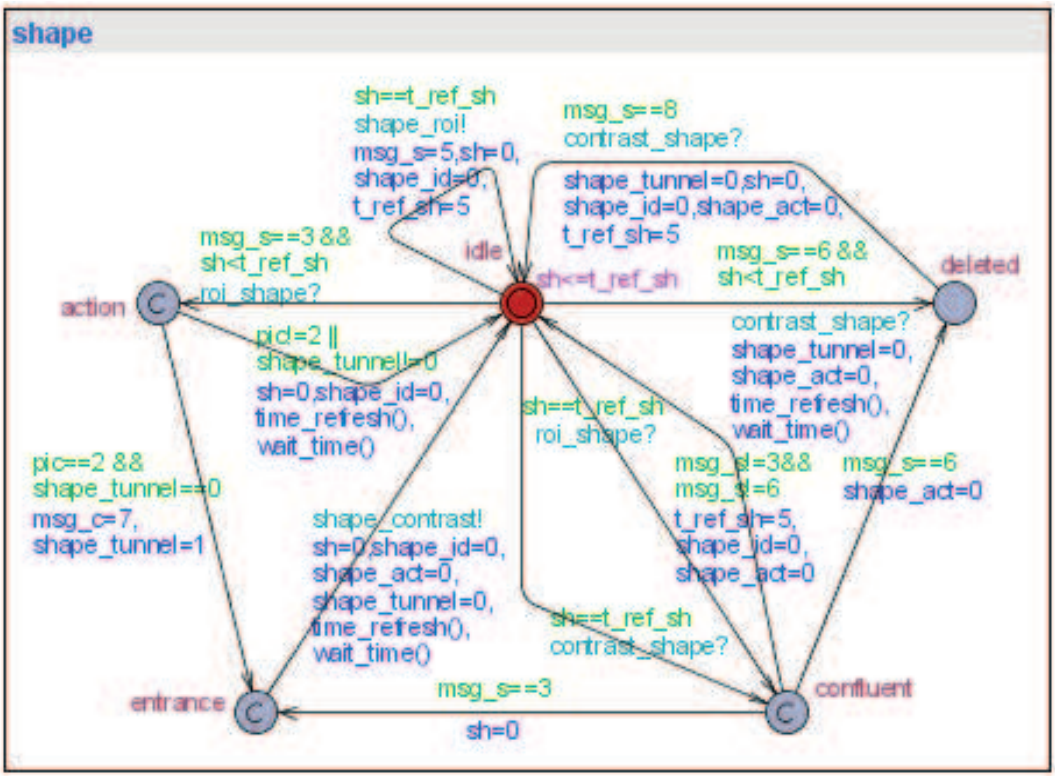


**Figure 24:** UPPAAL representation of the *Shape* component

## 4.5. Timed Behavior

The time-dependent behavior is of great concern in the model implementation of the AutoVision system as well as in DEVS models in general. This has been achieved as a consequence of feasible types of the states contributed in the UPPAAL Model Checking tool in Chapter 3.2. The states marked in the model as committed stand for the life time of a state when it is equal 0. An internal transition is immediately taken when entering, if the life time of the state is 0. States without time invariants denote a performance of the model states if their life time is $\infty$. In this case the model stays in the same state until an external event occurs. Finally, the states annotated with time invariants represent the execution process of the internal, external and confluent transition functions respectively, see also Chapter 3.2.

Being in the state *shape.confluent* as shown in the Figure 24, the *Shape* component will immediately proceed into *shape.deleted*, *shape.entrance* or *shape.idle*, executing the transition according to the contained message information. If the *msg_s==6* (meaning that the current Shape component must be deleted), then the *Shape* transits *shape.deleted*; if *msg_s==3* (meaning that the current environment information must be updated), then *shape.confluent* changes to *shape.entrance* and otherwise moves *shape.confluent* to *shape.idle*. Thereby, the elapsed time *e* is evaluated using two implemented functions. As an example, the code of the implementation of evaluation methods for the *Shape* component is given in the Figure 25.

```
int t_ref_sh=5;
int x;

void time_refresh(){
x=t_ref_sh;
x=x-1;
}

void wait_time (){
if (x>1)
t_ref_sh=x;
else
t_ref_sh=5;
}
```

**Figure 25:** Time evaluation methods in the *Shape* component

The methods implementation for the evaluation of the expired time for other components is just the same applying appropriate variables and is given in Appendix. These functions help to determine the remained time for the node to stay in after fulfillment of $\delta_{int}$, $\delta_{ext}$ or $\delta_{con}$. This effort was done due to the lack of automatic time passing functions, for example such as *wait_time* that is used in C++.

## 4.6. Request detection and information transmission

A DEVS Model is represented in the UPPAAL Model Checking tool as a network of processes. Each process is defined as a single automaton with locations, edges and local declarations respectively. These automata can communicate with each other via synchronization channels and transmit information in both directions. This is enabled by annotating edges with synchronization labels of the form *channel!* and *channel?*, see Chapter 2.3. In such a manner the two processes are synchronized when two edges from different processes are fired at the same time, i.e. the guards of both edges are satisfied and the edges have synchronization labels that evaluate to the same channel.

Requests from the components about an external environment and information transmission in the given DEVS model of the AutoVision example are enabled by using synchronization channels. The names of these synchronization channels are defined as a direction from one component to another, e.g. *roi_shape* (from *ROI* to *Shape* component), *roi_contrast* (from *ROI* to *Contrast* component), *contrast_roi* (from *Contrast* to *ROI*), *taillight_contrast* (from *Taillight* to *Contrast*) or *shape_contrast* (from *Shape* to *Contrast*).

For example, if the *Shape* component in the Figure 24 wants to get current information about the external environment to act according to the situation, it sends a request to *ROI* component and the edge with *shape_roi* channel (in this case Shape is an initiator and *ROI* is a receiver) is fired. In its turn, the *ROI* component establishes the picture of the external environment by using the synchronization channel *testbench_roi* between *Testbench* and itself (*Testbench* is a sender and *ROI,* the other way around, is receiver). Thereafter, *ROI* replies with appropriate information to the *Shape* via *roi_shape* channel, where *ROI* component is the sender and *Shape* component is a receiver. Information

transmission between the components in the model and the request's detection about the external environment for the *Contrast* component executes in exact the same manner sending received data through synchronization channels in both directions.

## 4.7. Activation and deleting process (of model components)

The activation and termination of the components in the model is also achieved by means of synchronization channels. Figure 24 depicts an example, where the shape of the tunnel entrance is found: *pic=2* by the *Shape* component, then *Shape* moves from the state "*action*" to "e*ntrance*" updating the following content: *msg_c=7,* meaning that a new *Contrast* component can be generated and *shape_tunnel=1,* denoting that the car is before the tunnel. After this the *Shape* sends the determined information through the *shape_contrast* synchronization channel to the *Contrast* unit. *Contrast* compares on the synchronized edge if the message information about activation of a new *Contrast* component is true, i.e. *msg_c==7* and *shape_tunnel==1*. If the guard expression is satisfied, a new component will be initiated, if not then this will be recognized by the "a*ction*" state of the *Shape* component. Another outgoing edge from this state is annotated with the guard expression which checks whether the picture information is wrong *pic!=2* or there is no tunnel *shape_tunnel!=0* found, in this case *Shape* changes to the state "*idle*" again.

The execution order of activation and deleting processes is the main point of the implementation. In this particular case the user can independently decide, which sequence he desires: activation of the new components as first or deleting of unusable processes. Taking the AutoVision example into consideration, activation of the *Contrast* component occurs before the *Shape* process is deleted. The reason is that both components can be enabled at the same time due to the given description above sending request to the *ROI* element.

Another illustrative example is when the *Contrast* process is in the state "*entrance*", which can proceed with two alternatives: creating a new *Taillight* process or deleting the *Shape* component simultaneously. The choice will be taken nondeterministically by the Model Checking tool. This is realized by using committed locations and two outgoing edges annotated with corresponding synchronization channels. If the component ever enters the "*deleted*" state once, it remains in this location until an external event activates it again, see Figure 22 and Chapter 3.2 for general information.

## 4.8. Internal, external and confluent transition functions

To distinguish the basic DEVS relation functions in the model is not difficult. According to the transformation rules given in Chapter 3.2 the distinctive features of these constructions are clock expressions on the guards and synchronization labels.

Figure 24 illustrates all three DEVS relation transitions in the *Shape* component of the given system. Starting from the state *shape.idle* internal, external or confluent transitions can occur retrieving the time of the system. If the timeout of the state *shape.idle* is not expired two different edges can be taken depending on the activated synchronization labels *roi_shape?* or *contrast_shape?*. If the timeout is

occurred, *shape.idle* can transit to *shape.idle* taking a self-loop or to *shape.confluent* taking the edge annotated with synchronization labels *contrast_shape?* or *roi_shape?*. To make the construction of the confluent function more distinctive a special state "*confluent*" is created, handling further differentiations and actions. For this reason, additional states are implemented in the *Taillight* component too, which are named respectively to their executing functions.

According to the given system specification it may happen that some of DEVS relation functions are equal to each other. The specification of the AutoVision example is not an exception thereof. To find out the conformity points is left to the reader.

## 4.9. Restrictions

For additional modification of the synchronization channels it is to mention that urgent and broadcast channels are not used in the DEVS to UPPAAL mapping since clock guards are not allowed on the edges synchronizing over these channels. Since the timed behavior of DEVS models is an inalienable part, this makes no appreciation to use urgent and broadcast synchronization channels without time retrieval. The resulting limitation concerning synchronization channels is that the edges can be annotated only with one synchronization channel. A multiple annotation with more than one synchronization label is not allowed in the UPPAAL Model Checking tool.

Another restriction in UPPAAL is the integer type, meaning that it is only possible to use integer variables for defining time units and real numbers are excluded.

An explicit time evaluation of the expired time after executing one of the DEVS relation transitions is not permitted in the UPPAAL Model Checker. This problem can be avoided by implementing distinct functions as it was shown in Section 4.5. Comprehensive data on declaration of components of the AutoVision example are given in the Appendix.

# 5. Verification of DEVS Models

The verification phase in the design process is a variant of logical analysis of the system implementation. This phase verifies usually the correctness properties of the system. A variety of correctness properties that are often used in the verification process are classified as follows [19]:

1) *functional correctness properties*
2) *safety (invariant) and liveness properties*
3) *timing properties*

According to the classification of the correctness properties given by A. Gupta, it may be concluded that the UPPAAL specification formalism based on the temporal logic fulfils all specification requirements. Foremost, temporal logic reasoning explicitly about time and thus, is more suitable than a logic that does not provide any special facilities for doing so.

The verification process also checks the operational correctness by means of checking lower-level operational specifications against higher-level assertions. The formalism employed for these specifications may be the same or different. The UPPAAL verification process is based on a dual specification approach where the operational specification expresses the DEVS formalism and is checked against assertions described in the temporal logic. The single specification approach is based on extended DEVS formalism (C-DEVS) [41].

## 5.1. Requirement specification language

The main purpose of the UPPAAL Model Checker is to verify the model with respect to a given requirements specification, which are nothing else than the description of intended behavior of a hardware design. The query language of UPPAAL consists of *path formula* and *state formula*.

A *state formula* is a side-effect free expression describing an individual state, i.e. a superset of its guards where the use of disjunctions is not restricted. In addition it is possible to test whether a process is in a particular location using an expression on the form *P.l*, where *P* is a process and *l* is a location.

A *path formula* explores the overall path and traces of the model. UPPAAL supports the following path formula: *reachability*, *safety* and *liveness*.

*Reachability* properties test if some state satisfying $\varphi$ should be reachable using a path form. In the UPPAAL Model Checker this property is defined using the syntax $E <> \varphi$, meaning there exists a path where $\varphi$ eventually holds.

*Safety* properties are on the form “something bad will never happen”, i.e. φ should be true in all reachable states with the path formula $A[]\,\varphi$. Whereas $E[]\,\varphi$ means that there should exists a maximal path such that $\varphi$ is always true.

*Liveness* properties mean that something will eventually happen. They are expressed in UPPAAL as $A <> \varphi$ and $\varphi \rightarrow \phi$. The first one signifies that $\varphi$ is eventually satisfied and the last one says that whenever $\varphi$ is satisfied, and then eventually $\phi$ will be satisfied.

Beyond these properties it is possible to check the absence of deadlocks in the model. Deadlock is expressed in the UPPAA tool using a special formula: $A[]$ *not deadlock* .

Questions and requirements upon DEVS models for reconfigurable systems being of the great importance and relevance for the verification process in the UPPAAL Model Checking tool are given in the Table 1 in the next page.

| *Description* | *UPPAAL notation* |
|---|---|
| ***Is the state x reachable?*** | **E<> P1.S1** (is the state S1 in the process P1 reachable?) |
| ***Is an edge x used?*** | **E<> P1.S1** ***imply*** **P1.S2** (the automata P1 is in the state S1 and it transits directly to state S2 taking the outgoing edge S1S2. In other words, is the state P1.S2 reachable from the state P1.S1?) |
| ***Is a synchronization channel x used?*** | **E<> P1.S1** ***imply*** **P3.S2** (the process P1 is in the state S1 and proceeds to state S3 in process P3 firing transition with the appropriate synchronization channel)<br><br>**E<> P1.S1** ***imply*** **(P2.S2** ***or*** **P3.S3) (**inspects if a synchronization channel fired in more than one transition. In this form it is possible to prompt if only one enabled/possible transition with synchronization label is fired or all of them through conjunction declaration) |
| ***Is at least one component always deleted?*** | **E[] (P1.S1** ***or*** **P2.S2** ***or*** **P3.S3)** (S1, S2 and S3 denote "deleted" or "exit" locations. The number of "deleted" locations is not restricted) |
| ***Is the state in the form of (x, y, *, *, *) reachable? (where x and y are definite states in the process)*** | **E[] (P1.S1** ***and*** **P2.S2** ***and*** **P*.S*** ***and*** **P*.S*** ***and*** **P*.S*)** (is the state in the form *(x, y, ∗, ∗, ∗)* always reachable? x and y are fixed states in the automata template, states marked with ∗ are nonattached)<br><br>**E<> (P1.S1** ***and*** **P2.S2** ***and*** **P*.S*** ***and*** **P*.S*** ***and*** **P*.S*)** (is the state in the form *(x, y, ∗, ∗, ∗)* actually reachable? x and y are fixed states in the automata template, states marked with ∗ are nonattached) |
| ***If a confluent function is executed?*** | **E<> P1.S1** ***imply*** **P3.S2 and P3.p3==n** (P3 is the automata template of interest and S2 is the location symbolizing "confluent state", p3 – local clock in P3 and *n* has an integer value, whereas time request is optional )<br><br>**E<> P2.S3** (is the state S3 in the process P2 reachable, S3 is defined as "confluent")<br><br>**E<> P1.S1** ***imply*** **(P1.S2 and P1.S3 and P1.p1==n)** (process P1 is in the "confluent" state S1, states S2 and S3 are possibilities where to S1 can proceed to S2 and S3, p1 – local clock in P1, n – integer number. Time specifications is also optional) |
| ***Execution time of some transitions through certain nodes?*** | **E[] (P1.S1 imply P1.S3) and P1.p1<=n** (is it always possible to execute a transition through a certain node, e.g. P1.S3 in n time units or even less then n time units? n is an integer upper time bound)<br><br>**E<> (P1.S1 imply P1.S3) and P1.p1<=n** (is it eventually possible to execute a transition through a certain node, e.g. P1.S3 in n time units or even less then n time units? n is an integer upper time bound) |
| ***Is the implemented model deadlock free?*** | **A[]** ***not deadlock*** (This specification requirement checks whether implemented model is deadlock free) |

**Table 1:** Specification requirements

## 5.2. Verification results

In this section the most relevant verification results of the AutoVision example will be presented. This includes a definition of the specification requirements on the given systems as well as their essential description.

Foremost, the deadlock specification was checked to get information, if the implemented model is deadlock free:

***A[] not deadlock*** – Property is not satisfied

The deadlock situation may be caused, for instance, when the *Contrast* component sends a "*delete*" message to the *Shape* and at the same point of time *Shape* receives a reply from the *ROI* process to his request sent before.

To find out if a particular state in the automaton of interest is reachable, for a example if the state "*exit*" is ever in the *Contrast* reachable, describes the following specification:

***E<> contrast.exit*** - Property is satisfied

It is not less interesting is to find out, if a certain transition is taken. This can be the case if during the whole simulation phase this does not stick out. For instance, if the *Shape* component will be inspected being in the state "*confluent*" to verify if the edge from state "*confluent*" to "*idle*" will be ever taken results in the specification:

***E<> shape.confluent imply shape.idle*** - Property is satisfied

It seems that the synchronization channel *taillight_contrast*, which connects two automata: *Taillight* and *Contrast* is absolutely not in use, see Figure 23. This case is determined using the following specification:

***E<> taillight.idle imply contrast.S1*** - Property is satisfied

or retrieving time explicitly to be sure that the timeout has occurred:

***E<> taillight.idle imply contrast.S1 and taillight.ta==taillight.t_ref_ta*** - Property is satisfied

For the convenience of reconfigurable hardware systems it should be proved if at least one model component is always deleted:

***E[] (shape.deleted or contrast3.exit or taillight4.deleted)*** - Property is satisfied

By forming subsets from the set of "*deleted*" locations it can be tested, which of these model components could not be deleted at the same time.

The execution of the confluent function is verified again on the instance of the *Shape* component, which it closely and tricky connected with automata template of the *Contrast* component:

***E<> contrast.entrance imply shape.confluent and shape.sh==shape.t_ref_sh*** - Property is satisfied

An alternative specification would be if the execution of the confluent function in the *Shape* process can be checked as follows:

> ***E<> shape.confluent imply (shape.entrance and shape.deleted)*** - Property is satisfied

In this case the time request is omitted because the state "*confluent*" is marked as committed location.

Some implementation restrictions were given in Section 3.3.7. The major one is impossibility to evaluate the elapsed time *e* of a state. Therefore, to resolve this limitation an effort was made to retrieve the time of execution inside UPPAAL. The condition is that additional local clocks must be defined to evaluate the time of execution. These clocks are named matching the function, i.e. *shape_act* (see Figure 24) and after a transition they are set to 0 again. For instance, how much time is needed to detect a tunnel and to perform corresponded transition in the *Shape* component? The following specification allows to implement this time evaluation:

> ***E<> (shape.idle imply shape.action imply shape.entrance imply shape.idle) and shape.shape_act<=6*** - Property is satisfied

In this case the complete execution path is given through discrete states: *shape.idle* imply *shape.action* imply *shape.entrance*. Notice that some intermediate states can be omitted if it is rather obvious that there is no any other execution path for this transition:

> ***E<> (shape.idle imply shape.entrance imply shape.idle) and shape.shape_act<=6*** - Property is satisfied

During simulation it was observed that tunnel detection lasts about 6 time units; let us suppose that there exists eventually a path where this specification is true. The next specification proves whether this transition always lasts 6 time units or not:

> ***E[] (shape.idle imply shape.entrance imply shape.idle) and shape.shape_act<=6*** - Property is not satisfied

The result is obvious: the property is not satisfied. Even so, the simulation process in the UPPAAL Model Checking tool permits time evaluation through observing the execution trace. Afterwards, it is still possible to verify assumptions depending on the observations.

The demonstrated results do not cover all verification capabilities of the UPPAAL Model Checking tool. Specification requirements can become more complex in the structure depending on the desired goals of the investigation and the interest for non-functional behavior of the system, but it still depends on the individual interest of the researchers, the user's way to express desired features and the formulated problem.

# 6. Conclusions

The technique of formal verification described in this diploma thesis is already used in industry to find nontrivial errors in implementations [26], e.g. circuits and protocols and by a number of companies such as IBM, Intel and Motorola which benefits from applying Model Checking as a part of the design process due to relatively easy usage. Model checking alleviates the process of verification in virtue of its automation level. Additionally it provides a counterexample trace of the violated specification requirements and avoidance of the complicated proofs constructions which in its turn facilitate the modeling and verification processes for users.

Taking these aspects of utmost importance it was decided to show the reader the connection between the verification and DEVS models as well as relation of DEVS models and its' use in reconfigurable systems.

A variety of different available Model Checking tools was examined as a part of this research giving a short introduction to those that were found as applicable tools for modeling and verification of DEVS Models. Being guided by the selection criteria given in the introduction, it must be concluded that the UPPAAL Model Checker is the most convenient formal verification tool.

Furthermore, the overall structure and the main features of the UPPAAL tool were represented in this diploma thesis, along with the design process of DEVS models and RecDEVS extension for reconfigurable systems. Because of various and multiple attempts to represent DEVS models in the UPPAAL tool the implementation possibilities were exemplified. Many conclusions were made, powerful and forcible limitations were found. However, they have not complicated the transformation and verification process of the reconfigurable systems based on DEVS models, e.g. AutoVision example.

The collected verification results and gained experience during the investigation period proved that the UPPAAL Model Checker provides the ease of use for simulation and verification necessary for system verification. It is quite simple to prove the supposed system behavior and to check the absence of deadlocks in the model. Verification results recognize that in spite of some time restrictions in the UPPAAL tool, it is still possible to verify timed behavior of DEVS models in a certain way by extending the model component with additional local clocks and defining appropriate specification requirements.

In the future, this research is going to continue expecting further developments in this field. There exists a graphical editor for DEVS models developed at the Department of Integrated Circuits and Systems that also enables the design process of the systems and stores the model design in XML based format (SCXML). The next approach would be the development of an UPPAAL generator, which automatically provides generation of verified UPPAAL Models from DEVS models designed in the XML format.

$$
\begin{aligned}
\mathit{Expression} \rightarrow\ & \mathrm{ID} \mid \mathrm{NAT} \\
\mid\ & \mathit{Expression}\ \texttt{'['}\ \mathit{Expression}\ \texttt{']'} \\
\mid\ & \texttt{'('}\ \mathit{Expression}\ \texttt{')'} \\
\mid\ & \mathit{Expression}\ \texttt{'++'} \mid \texttt{'++'}\ \mathit{Expression} \\
\mid\ & \mathit{Expression}\ \texttt{'--'} \mid \texttt{'--'}\ \mathit{Expression} \\
\mid\ & \mathit{Expression}\ \mathit{AssignOp}\ \mathit{Expression} \\
\mid\ & \mathit{UnaryOp}\ \mathit{Expression} \\
\mid\ & \mathit{Expression}\ \mathit{BinaryOp}\ \mathit{Expression} \\
\mid\ & \mathit{Expression}\ \texttt{'?'}\ \mathit{Expression}\ \texttt{':'}\ \mathit{Expression} \\
\mid\ & \mathit{Expression}\ \texttt{'.'}\ \mathrm{ID} \\
\mathit{UnaryOp} \rightarrow\ & \texttt{'-'} \mid \texttt{'!'} \mid \texttt{'not'} \\
\mathit{BinaryOp} \rightarrow\ & \texttt{'<'} \mid \texttt{'<='} \mid \texttt{'=='} \mid \texttt{'!='} \mid \texttt{'>='} \mid \texttt{'>'} \\
\mid\ & \texttt{'+'} \mid \texttt{'-'} \mid \texttt{'*'} \mid \texttt{'/'} \mid \texttt{'\%'} \mid \texttt{'\&'} \\
\mid\ & \texttt{'|'} \mid \texttt{'\^{}'} \mid \texttt{'<<'} \mid \texttt{'>>'} \mid \texttt{'\&\&'} \mid \texttt{'||'} \\
\mid\ & \texttt{'<?'} \mid \texttt{'>?'} \mid \texttt{'and'} \mid \texttt{'or'} \mid \texttt{'imply'} \\
\mathit{AssignOp} \rightarrow\ & \texttt{':='} \mid \texttt{'+='} \mid \texttt{'-='} \mid \texttt{'*='} \mid \texttt{'/='} \mid \texttt{'\%='} \\
\mid\ & \texttt{'|='} \mid \texttt{'\&='} \mid \texttt{'\^{}='} \mid \texttt{'<<='} \mid \texttt{'>>='}
\end{aligned}
$$

**Figure 26:** Syntax of expressions in BNF

```
int t_ref_co=7;
int y;

void time_refresh(){
y=t_ref_co;
y=y-1;
}

void wait_time (){
if (y>1)
t_ref_co=y;
else
t_ref_co=7;
}
```

**Figure 27:** Time evaluation methods in the *Contrast* component

```
int t_ref_ta=8;
int q;

void time_refresh(){
q=t_ref_ta;
q=q-1;
}

void wait_time (){
if (q>1)
t_ref_ta=q;
else
t_ref_ta=8
}
```

**Figure 28:** Time evaluation methods in the *Taillight* component

## List of Illustrations



## Index of Tables

## List of Literature


**[1] R. Alur and D. Dill.** "A theory of timed automata," *Theoretical Computer Science*, Vol. 126, No. 2, pages 283-235, 1994.

**[2] T. Ball and S. Rajamani.** "Boolean programs: A model and process for software analysis," Technical Report 2000-14, Microsoft Research (2000).

**[3] I. Beer, S. Ben-David, and A. Landver.** "On-The-Fly Model Checking of RCTL Formulas," in *Proceedings of the 10th International Conference on Computer Aided Verification*, LNCS 818, pages 184–194. Springer-Verlag, June-July 1998.

**[4] K. Bondalapati and V. K. Prasanna.** "Reconfigurable computing systems," in *Proceedings of the IEEE*, Vol. 90, No. 7, pages 1201-1217, 2002.

**[5] G. Behrmann, A. David, and K. G. Larsen.** "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems (SFM-RT 2004), Bertinoro, Italy, September 13-18*, LNCS 3185, pages 200–236. Springer, 2004.

**[6] G. Behrmann and K. G. Larsen.** "UPPAAL – Present and future," in *Proc. of 40th IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

**[7] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster.** "Scalable distributed on-the-fly symbolic model checking," in the *International Journal on Software Tools for Technology Transfer (STTT)*, Vol.4, No. 4, pages 496-504, 2003.

**[8] S. V. Campos and E. M. Clarke.** "Real-Time Symbolic Model Checking for Discrete Time Models," *Theories and Experiences for Real-Time System Development*, Vol. 2, AMAST Series in Computing, World Scientific, Singapore, pages. 129–145, 1995.

**[9] E. M. Clarke and E. A. Emerson.** "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981, LNCS* 131. Springer 1981.

**[10] E. M. Clarke, O. Grumberg, and D. A. Peled.** "Model Checking," MIT Press, 1999.

**[11] E. M. Clarke, H. Jain, and D. Kroening.** "Predicate Abstraction and Refinement Techniques for Verifying Verilog," Technical Report CMU-CS-04-139(2004).

**[12] E. M. Clarke and J. M. Wing.** "Formal methods: State of the art and future directions," Technical Report CMU-CS-96-178, Carnegie Mellon University, School of Computer Science, September 1996.

**[13] C. Claus, W. Stechele, and A. Herersdorf.** "Autovision – A Runtime Reconfigurable MPSoC Architecture for Future Driver Assistance Systems," *It – Information Technology*, Vol. 49, No. 3, pages 181-186, 2007.

**[14] K. Compton and S. Hauck.** "Reconfigurable Computing: A survey of Systems and Software," ACM Computing Surveys, Vol. 34, No. 2, pages 171–210, 2002.

**[15] H. P. Dacharry and N. Giambiasi**. "A Formal Verification Approach for DEVS," in *Proceedings on the 2007 Summer Computer Simulation Conference*, 2007.

**[16] H. P. Dacharry and N. Giambiasi.** "Formal verification with timed automata and DEVS models: a case study," in *Proceedings of ASSE'05*. Argentine Society for Computer Science and Operational Research, 2005.

**[17] H. P. Dacharry and N. Giambiasi.** "From Timed Automata to DEVS Models: Formal verification," in *Proceedings of SpringSim'05*. SCS – The Society for Modeling and Simulation International, 2005.

**[18] H. Giese and S. Burmester.** "Real-Time Statechart Semantics,". Technical Report tr-ri-03-239, Universität Paderborn, Deutschland, Institut für Informatik, 2003.

**[19] A. Gupta.** “Formal hardware verification methods:  A Survey,” Formal Methods in System Design, 1: 151- 238, Kluwer Academic Publishers, 1992.

**[20] D. Harel.** “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming*, Vol. 3, No. 8, pages 231–274, 1987.

**[21] S. Hauck and A. Dehon.** “Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation,” Elsevier, 2008.

**[22] H. Jain, D. Kroening, and N. Sharygina.** “VCEGAR: Verilog counterexample guided abstraction refinement,” In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, 2007.

**[23] H. Jain, D. Kroening, N. Sharygina and E. Clarke.** “Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog,” in *Proceedings of the 42nd annual conference on Design automation*, June 13-17, 2005.

**[24] A. Krupp and W. Mueller.** “Modelchecking von Klassifikationsbaum-Testsequenzen,” GI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", 2005.

**[25] W. K. Lam.** “Hardware Design Verification: Simulation and Formal Method-Based Approaches,” Prentice Hall PTR, March 2005.

**[26] K. G. Larsen, P. Pettersson, and W. Yi.** “UPPAAL in a Nutshell,” in *Springer International Journal of Software Tools for Technology Transfer 1*(1+2).

**[27] K. G. Larsen, P. Pettersson, and W. Yi.** “Model Checking for Real-Time Systems,” in *Proceedings of the 10th International Symposium on Fundamentals of Computation Theory, Lecture Notes In Computer Science,* Vol. 965, pages 62-88, 1995.

**[28] F. Madlener, A. S. Huss and A. Bidermann.** “RecDEVS: A Comprehensive Model of Computation for Dynamically Reconfigurable Hardware Systems,” in *4th Workshop on Discrete-Event System Design*, 2009.

**[29] F. Madlener, H. G. Molter and A. S. Huss.** “SC-DEVS: An efficient SystemC extension for the DEVS Model of computation,” in *Proceedings – Design Automation and Test in Europe (DATE 2009)*. ACM/IEEE.

**[30] K. L. McMillan.** “A methodology for hardware verification using compositional model checking,” *Science of Computer Programming*, Vol. 37, No.1-3, pages 279-309, 2000.

**[31] K. L. McMillan.** “Symbolic Model Checking: An Approach to the State Explosion Problem,” Doktorarbeit, Carnegie Mellon University, Pittsburgh, Mai 1992.

**[32] E.-R. Olderog and H. Dierks.** “Real-Time Systems. Formal Specification and Automatic Verification,” Cambridge University Press, 2008

**[33] J. P. Quielle and J. Sifakis.** “Specification and verification of concurrent systems in CESAR,” in *Proceedings of the 5th International Symposium on Programming*, pages 337-350.

**[34] W. Reif, G. Schellhorn, T. Vollmer and J. Ruf.** “Correctness of Efficient Real-Time Model Checking,” *Journal of Universal Computer Science*, Vol. 7, No. 2, pages 194-209, 2001.

**[35] J. Ruf.** “RAVEN: Real-Time Analyzing and Verification Environment,” *Journal on Universal Computer Science (J.UCS)*, (1):89–104, February 2001.

**[36] J. Ruf.** “Techniken zur Modellierung und Verifikation von Echtzeitsystemen, ” *Dissertation*, Universität Karlsruhe, Logosverlag Berlin, March 2000.

**[37] J. Ruf and T. Kropf.** “Formal Verification of Discrete Real-Time Systems,” *Informationstechnik und Technische Informatik*, Vol. 43, No.1, pages 39-46, 2001.

**[38] J. Ruf and T. Kropf.** "Modeling Real-Time Systems with I/O-Interval Structures," In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 91-100. GI/ITG/GMM Workshop, Braunschweig, Germany, Shaker Verlag, March 1999.

**[39] G. Wainer, L. Morihama and V. Passuello.** "Atomatic verification of DEVS models," in *Proceedings of the SISO Spring Interoperability Workshop,* 2002.

**[40] J. M. Wing.** "A specifier's introduction to formal methods," *IEEE Computer*, Vol. 23, No. 9, pages 8-24, 1990.

**[41] B. P. Zeigler.** "Verification and validation of DEVS Models: Applying the theory of modeling and simulation to the needs of simulation based acquisition," *Summer Computer Conference Simulation Conference*, 2000.

**[42] B. P. Zeigler, H. Praehofer, and T. G. Kim.** "Theory of modeling and simulation," *2d edition*. Academic Press, London, 2000.