# A Model of Computation for Reconfigurable Systems

Ein Berechnungsmodell für rekonfigurierbare Architekturen
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Dipl.-Inform. Felix Madlener aus Frankfurt am Main
2013 — Darmstadt — D 17

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Institut für Integrierte Schaltungen und Systeme

# A Model of Computation for Reconfigurable Systems
Ein Berechnungsmodell für rekonfigurierbare Architekturen

Genehmigte Dissertation von Dipl.-Inform. Felix Madlener aus Frankfurt am Main

1. Gutachten: Prof. Dr.-Ing. Sorin A. Huss
2. Gutachten: Prof. Dr.-Ing. Hans Eveking

Tag der Einreichung: 25.11.2012
Tag der Prüfung: 9.4.2013

Darmstadt — D 17

## Erklärung zur Dissertation

Hiermit versichere ich die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 25. November 2012

(Felix Madlener)

# Contents

# 1 Introduction

## 1.1 Motivation

More than 40 years ago Moore [40] stated his famous law about an exponentially increasing number of logical elements in electronic circuits that still does hold. Beside all its technological implications, this prediction has had a severe impact on hardware design methodology. The complexity of hardware systems is also exponentially increasing together with the size. Design methodologies, which have been feasible in the 1970s are not able to handle current complex systems any more.

Conventional hardware development has addressed the problem of increasing complexity by levering the abstraction levels in the design phase. In general, this leads to the introduction of abstract *Models of Computation (MoC)* which allows a system designer to concentrate on functional behavior. The lower abstraction layers are hidden from the designer and are processed by advanced automatic processing steps, such as logic synthesis or automated routing tools.

### 1.1.1 Reconfiguration

In the last decade, a new kind of hardware, the reconfigurable hardware architecture, has emerged as a promising technique to increase the flexibility of hardware design. While this new technology is still in its infancy, especially the support of dynamic reconfiguration will clearly expand the present limitations of existing hardware implementations.

The general idea of such systems is to reconfigure the actual logical behavior of logical design elements. For example, a logical AND gate can be reconfigured to work as a logical OR gate. Beside some direct advantages, such as a serving as a fast development platform dynamic reconfiguration gives even more benefits: The reconfiguration of existing hardware resources enables their reutilization. This leads to a better resource exploitation without losing hardware-specific advantages such as high performance. At the same time, it is possible to apply reconfiguration methodologies to improve the flexibility and fault tolerance of hardware implementations. A hardware system has no longer to remain static after deployment but can be changed in-field to update the functionality or correct faulty behavior.

The introduction of reconfigurable hardware architectures does not only have great impact on the hardware implementations, but also on the design methodologies required for the development of these systems. The problem of system design flow complexity is even more present for reconfigurable systems, where the additional reconfiguration aspects increase the design space even more. While the number of logical elements is roughly the same, the possibility of reconfiguration introduces a whole new design dimension that has to be taken care of.

One main disadvantage of existing reconfigurable systems is the lack of an established proper design methodology that would enable a rise in the abstraction level to a point including reconfiguration. An important reason for this is the lack of an underlying formal MoC for reconfigurable hardware systems. Instead, the design methodologies for such systems are based on low-level design tools and hardware description languages that do not provide a proper formal semantic. Without such a semantic it is not possible to define formal design activities as required by a high-level reconfigurable systems design flow. By incorporating reconfiguration on such a low-level only, the formal models feature deep problems in modeling the reconfiguration specific properties.

An introduction of high-level MoCs does not only ease the system implementation of complex system, but also enables other design methodologies such as verification by the utilization of Model Transformation processes. A key benefit of formally defined models is the possibility to apply formal verification methods.

This work addresses the design flow complexity by first investigating the requirements for a reconfigurable design methodology. The insights on the development of formal reconfigurable hardware systems are then used to develop a formal MoC and a corresponding design flow for reconfigurable hardware systems. The *Reconfigurable Discrete Event Specified System (RecDEVS)* model first published by the author in Madlener et al. [34] is this high-level MoC.

### 1.1.2 Outline of this Thesis

The structure of this thesis is as follows. In the first half of Chapter 2 the essential features of a reconfigurable hardware systems are analyzed. In its second half a high-level design methodology for a MoC-based formal specification is developed. This design methodology is based on the concept of model transformation. It distinguishes between horizontal transformations that keep the same abstraction level and vertical transformations, lowering the abstraction.

The *RecDEVS* formalism, other related formalisms, and the design considerations that led to the development of *RecDEVS* are presented in Chapter 3. We will see that this event-based formalism with an integrated reconfigurable and timed behavior is highly applicable for reconfigurable hardware and real-time software models.

The design methodology of Chapter 2 is then applied to *RecDEVS*. The necessity and usage of horizontal transformations is demonstrated on base of an automated model transformation between *RecDEVS* and the formal representation of the UPPAAL model checker. A vertical model transformation from the *RecDEVS* MoC into hardware description languages is being presented. For this, the existing hardware description language SystemC has been extended towards reconfiguration. Details on both, the horizontal model transformation into UPPAAL models and the vertical model mapping in a SystemC-based implementation language, are given in Chapter 4.

Several different test scenarios have been exercised to illustrate the strength of various aspects of this work. The implementations and a discussion of their results are given in Chapter 5.

Finally Chapter 6 concludes this work and provides an outlook on further research topics that will benefit from this work.

## 1.2 Previous Own and Related Works

Parts of this thesis have been published previously in cooperation with other authors in various publications. A concept for resource management of reconfigurable hardware systems has been developed in the author's diploma thesis and published in Kühn et al. [30]. In his diploma thesis, Geisse [22] has developed a framework for the integration of different domain specific languages under supervision of the author of this thesis. In Molter et al. [39] the general concept of the presented high-level design methodology based on model transformation is presented. One possible vertical mapping from DEVS into SystemC is outlined in Madlener et al. [35] while the advantages of vertical transformations and the resulting possibilities for the verification of reconfigurable systems have been worked out in Madlener et al. [37]. The *RecDEVS* MoC that will be presented here in detail, has been published by Madlener et al. in [34].

There are also various other publications in which the author was involved during the creation of this thesis. These works of Anikeev et al. [4], Madlener et al. [36], Stöttinger et al. [51], and Anikeev et al. [5] focus on the area of security and cryptography. The original hope was that such implementations will turn out as good examples for the presented design methodology. However, as will be motivated in Chapter 5, other examples are much more promising and have been preferred later on.

The area of related work can roughly be separated into distinct research fields. Only few works focus directly on the research topic of this thesis. Thus, a short overview on the state of different relevant research fields will be given first. These fields are high-level design flows based on model transformation, design methods for reconfigurable systems and system verification.

### 1.2.1 Model-Driven Design Flows

There is a variety of publications which emphasize the advantages of MoC-based design flows. For this work approaches on model transformations in the hardware domain are of interest. Most of this research is done on base of SystemC as it directly supports arbitrary abstraction levels. Patel and Shukla [42] describe the integration of synchronous data flow models (SDF) in SystemC and Patel et al. [43] address the integration of a rule-based MoC (Bluespec). There is a lot of work available, which is aimed to overcome the lack of a formal SystemC specification and to extend it to a formally specified MoC. This would allow to reduce the need for other MoCs in the area of hardware development. Vardi [55] summarizes formal methods which can applied to SystemC itself, and Man [38] develops a formal MoC for a subset of the SystemC language.

An integration of Petri Nets is described by Rust et al. [46]. Each Petri Net is directly implemented as a SystemC module without any supporting framework, which makes this approach questionable for complex system specifications. The integration of an analog MoC to support mixed-signal simulation is described by Aljunaid and Kazmierski [2]. As it is focused on the analog simulation environment, it lacks a methodical description of the integration into SystemC execution environments. However, none of these works address reconfigurable systems.

HetSC, originating from Herrera and Villar [25] specifies an additional SystemC layer to ease the integration of different MoCs. It features the so-called Border Process and Border Channel to simplify the SW implementation of a heterogeneous MoC design.

Ptolemy II from Brooks et al. [12] is a genuine design environment for the combination of arbitrary MoCs. Its focus is the research of model interaction in heterogeneous systems instead of model transformations. To support HW synthesis in terms of VHDL generation it was extended by Filiba et al. [20]. In contrast to our proposed system level design flow, they transform models from the synchronous reactive domain only. Their extension neither supports heterogeneous models nor discrete event models.

For this work we have taken the methodology from Molter et al. [39] as reference. It distinguishes horizontal and vertical transformations aimed to denote either the transformation of single implementations or the transformation of complete MoCs.

As already stated we strongly believe that a more formal approach will yield better results than extending existing description languages like SystemC. In the area of formally defined MoCs the Discrete Event Specified System (DEVS) from Zeigler et al. [58] seems most promising. It is highly flexible and several MoCs can be easily transformed into a DEVS model description. This fact has already been exploited in previous works, e.g., Risco-Martín et al. [45] describe an UML statecharts transformation into DEVS and Bobeanu et al. [9] present a Petri Net conversion.

### 1.2.2 Reconfiguration Methodologies

There are various approaches which deal with modeling methods for reconfigurable hardware architectures. The first approach to be mentioned addresses the development of dedicated frameworks around existing design languages. These frameworks add certain reconfiguration mechanisms and reuse existing design flows for conventional hardware development. SyCERS by Santambragio [48] extends VHDL, the Perfecto framework by Hsiung et al. [26] extends SystemC and the work by Craven and Athanas [17] is based on ImpulseC. While these works can benefit from existing design tools, the underlying design languages lack a formally specified MoC. It is unclear, whether and how formal design activities such as verification are realizable here.

The Molen processor by Vassiliadis et al. [56] and Morpheus by Thoma et al. [53] utilize reconfigurable architectures as flexible coprocessors. They do not put a specific focus on a formal computational model and are less suited for hardware solutions.

A slightly different approach is used in the Hybrid System Architecture Model (HySAM) by Bondalapati and Prasanna [10]. This proposal does provide a formal model for the reconfiguration process. However, it separates the reconfiguration from the execution functionality. To create a complete reconfigurable system the HySAM model has to be combined with some other model. This will clearly limit a comprehensive formal analysis of component-induced reconfiguration activities.

In the area of dynamic hardware-aware MoCs the proposal OSSS+R of Schallenberg et al. [49] is to be mentioned. This approach utilizes the concept of code polymorphism from the object-oriented software development domain. We argue that this approach is somehow limited for dynamic hardware reconfiguration due to the underlying complexity of the polymorphism paradigm.

By the Dynamic Structure Discrete Event Specified System (DSDEVS), Barros [6] has already extended DEVS with a dynamic structure. Though not originally targeted to reconfigurable hardware architectures, many of DSDEVS concepts were considered and extended for the development of the more general *RecDEVS* model.

### 1.2.3 Verification

This thesis shows that one great benefit of horizontal model transformations stems from the application of formal verification techniques. There are different verification techniques such as logical inference based on theorem proving or the Model Checking approach. Clarke [14] gives a comprehensive overview over these techniques. In this work an approach based on the Model Checking tool UPPAAL presented by Larsen et al. [32] was selected.

Regarding the verification of *RecDEVS* models, there is some preliminary work on the formal verification of the underlying DEVS formalism. The work of Morihama et al. [41] implements an own theorem prover, while Weingart [57] as well as Dacharry and Giambiasi [18] benefit to some extend from the established UPPAAL model checking environment.

## 1.3 Impact

This work is intended as a fundamental step towards a scalable and persistent design methodology for reconfigurable systems. By using the formally specified *RecDEVS* MoC and the proposed high-level design flow it is likely to integrate all the new features reconfigurable hardware architectures can offer into a real operational system.

Currently, various approaches exist that try to integrate reconfigurable features into existing hardware design flows in a rather pragmatic way. As they are able to reuse existing technologies they may be more effective in short term. However, it is questionable whether such approaches without a formal base have the same long term potential.

Although this work proposes the *RecDEVS* formalism as the foundation of its high-level model transformation flow and verification capabilities, it is not limited to this formalism. This work defines the requirements of a model for reconfigurable systems as a set of formal lemmas. As long as any model can also fulfill these requirements (or be modified in such a way), it will be possible to adapt the presented design methodology for it as well.

# 2 System Development of Formally Specified Reconfigurable Systems

In this chapter, an overview over design methodologies for reconfigurable hardware systems will be given.

In the first half of this chapter, the concepts and the advantages of model-based system development are presented and explained. It is detailed how a formally defined MoC such as the *RecDEVS* formalism can be utilized for the development of reconfigurable systems. For this, the concept of model transformation and two targeted design flows for reconfigurable systems are presented. A special focus will be set on formal verification, which is an important aspect of formal design methodologies in general and of reconfigurable systems in special.

In the second half, reconfigurable hardware itself is introduced in more detail. This work sets a special focus on the most established reconfigurable architecture, the *Field Programmable Gate Array (FGPA)*. As a conclusion of this chapter, the most important aspects and features are summarized in a collection of lemmas. As we will see in Chapter 3, the development of an approriate MoC allows to select between some different design variants. These lemmas embody a major design criteria when it comes to the selection of the best design variant for a MoC in the context of reconfigurable systems.

## 2.1 Design Flow Methodologies

In the recent past the abstraction in system-level design has been significantly raised to handle the complexity of modern computing systems. Conventional design flow methodologies have been enriched with formal *Models of Computation (MoC)*. In software-centric design flow methodologies this process is reflected by the introduction of model-driven design tools like MOFLON (Amelunxen et al. [3]) or Ptolmy II (Brooks et al. [11]). Hardware-centric design methodologies and circuit-level design follow the same path, but are still lacking the same amount of established model-driven design tools.

At the same time reconfigurable hardware architectures have emerged as a new computing platform and formal verification has been established as an design alternative to testing and simulation environments. The possibility for reconfiguration introduces additional design flow aspects for which no well-established design methodologies exist yet.

### 2.1.1 MoC-based High-Level Design Flow

Most existing design flows for reconfigurable systems are directly based on low-level implementation languages like VHDL or Verilog. While it is possible to completely implement a design on these lower abstraction levels it is highly appropriate to raise the abstraction level and to represent a system exploiting several Models of Computation. These models provide a more abstract system view and give multiple benefits as follows.

They allow to step back from the actual platform-dependent implementation and to obtain a more abstract view of the system and its behavior. A high-level design-flow based on an abstract MoCs can ease the system development and can account system requirements. Another great benefit of a design-flow with MoCs is the verification ability. High-level specifications allow for the formal verification of system-level constraints without having to respect language specific properties of low level hardware description language (HDL) based methods. Therefore verification is one of the strengths of MoCs over low-level design-flows. Reusability and modeling comfort are additional advantages.

Distinct MoCs provide different specific properties and different advantages for system-level design. A high-level design flow should therefore not be composed of a single MoC, but it should allow the integration of arbitrary and possibly new MoCs.

In the sequel, a systematic approach to integrate different MoCs into an integrated system-level design flow for reconfigurable systems will be presented. The key feature to achieve this goal is the process of model transformation or model mapping. With this process the functional description given in one model can be transformed into the formal specification within another MoC.

Note that the exploitation of just a single generic MoC instead of a plain implementation language will already provide a remarkable benefit due to the higher abstraction level. Utilization of one MoC would allow to implement most of the system using, e.g., SysML with its design methodology and tools. By supporting multiple MoCs previously existing hardware components, e.g. described as FSMs, can also be integrated and reused. Dedicated parts of the inter-module communication may then be implemented by Petri-Nets, which allow to verify their liveness (i.e., the absence of potential deadlocks). To gain the full benefit of different MoCs, the system-level design flow requires to support an easy way to integrate new MoCs.

If new features like reconfiguration are to be supported, a corresponding new design flow is required. In general it is not reasonable to create completely new design flows, but to exploit existing technologies and expertise. This goal can either be achieved by defining a transformation of a new MoC into an equivalent system model specified within another MoC, which is already part of the design flow, or by implementing the MoC in a suited specification language like SystemC.

The consideration of existing MoCs as a means for design entry allows to reuse expertise and knowledge, which has already been aggregated for these specific models. It is then possible to partition the design into smaller units and to use domain-specific MoCs instead of a generic MoC or a design language. As a substantial part of the domain-specific features will already be included in the MoC definition, this will considerably ease a system development. This leads to our first requirement for developing a design flow methodology for reconfigurable hardware systems, which is given in Lemma 1.

**Lemma 1 (Model Driven)** *A design flow for reconfigurable hardware systems should be based on an appropriate Model of Computation (MoC).*

The integration of existing MoCs into the system design flow allows the reuse of all the expertise which already exists for this model. For example, it might be possible to model a part of the system with Petri Nets, allowing the analysis of deadlocks and liveness. Such a formal analysis is not possible with low level implementation languages as they do not contain a sufficiently formal specification or this specification is too complex to allow proving of interesting system properties. There is a wide variety of available MoCs focusing on different problem domains, examples are UML, Petri Nets, or FSM. However, a problem arises on how to integrate such models into a consistent design flow.

Furthermore, the raised abstraction layer no longer utilizes a specific hardware or software specific description language like VHDL or C, but a computational model that can be mapped into both implementation domains (HW or SW) in the ongoing design flow.

Thus, there are two possible orthogonal mappings for a design flow described by multiple MoCs: Vertical transformations, which transform a high-level MoC into lower abstraction levels and horizontal transformations which map between different high level MoCs. These two options have a considerable impact on the design flow consistency and will be discussed in more detail in the following.

## Vertical Transformation

First, the abstract models denoted in different MoCs can be transformed vertically into underlying programming languages. This process is also known as *Implementation-step* when the target model of the transformation is a programming language.

Inside the vertical transformation process two fundamental concepts can be distinguished: Mapping of the whole MoC in a generic way and a specialized mapping of a single model instance. The first approach, the mapping of the whole MoC, can operate on arbitrary models. It reflects each of the MoC specific computational rules in a generic way in the underlying programming language. Each MoC model instance will be based upon these computational rules. Once the complete MoC formalism is mapped into a programming language, we can derive a transformation for every single model. The model derivation can then be done automatically.

The latter approach, the mapping of a single model, may result in a more fine-grained implementation. It does not transform the generic computational rules of the MoC but rather the specific behavior of a specific model. As the mapping process is mostly handcrafted, we can cope with model specific characteristics and thus optimize the code. Modeling expertise about the concrete realization of the model is then used to create a sufficiently simple low-level description. The transformation into the programming language cannot be done automatically. Each time the model is modified, its changes must be transformed, too. This approach does also work when not all original MoC properties can be represented in the target language, as long as they are not a substantial part of the actual model to be mapped. In Chapter 4 it is detailed, how the *RecDEVS* formalism can be vertically transformed into the lower level programming language SystemC.

Vertical transformations are *platform specific*. They integrate low-level implementation constraints into the system description. So, if we change these constraints, e.g., change the platform architecture of the implementation, we may need to adapt all vertical transformation engines in our design-flow methodology to meet the new requirements. Vertical transformations are also *poorly scalable*. Each supported combination of MoC and programming language requires an own specialized transformation engine.

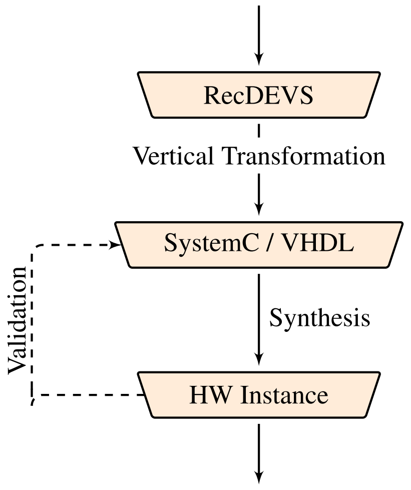


Figure 2.1: Vertical Design Flow

In contrast to the vertical transformations, horizontal ones between high level MoCs are platform independent and scale better, as it will be explained the following. Thus, it is appropriate to utilize horizontal mappings when possible and to reduce the number of vertical transformations.

Please note that it is generally not sufficient to provide just one vertical transformation engine. In general, MoCs build different disjoint classes, where the MoCs within each class can only be transformed into elements of the same class, e.g., continuous time MoCs cannot directly be transformed into time discrete ones. Thus, we need a vertical transformation engine for at least one representative MoC of each class. The other class members can be transformed vertically after successive horizontal transformations within the same MoC class.

In Fig. 2.1 a sample vertical design flow is depicted. A system may be described in *RecDEVS*, it will then be mapped into SystemC or VHDL descriptions that are implemented on reconfigurable hardware instances. It is also possible to utilize the reverse mapping direction as it is the case when validation results from lower abstraction levels influence the design in the higher abstraction levels.

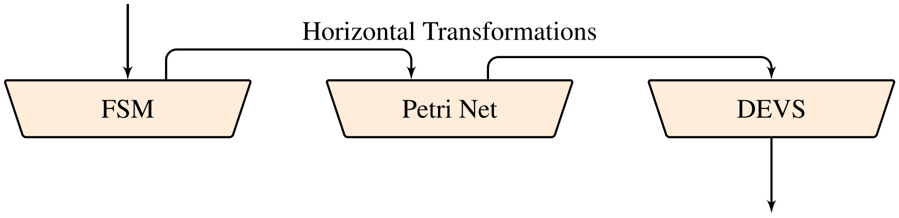


Figure 2.2: Horizontal Design Flow

### Horizontal Transformation

In the second design flow variant, MoC-based system descriptions can be transformed horizontally into other MoCs, e.g., Petri Nets to DEVS, and then taken down to SystemC code. In this case only one vertical transformation, i.e., one “synthesizable” MoC, is required.

Horizontal transformations can ensure the consistency of the different models. They stay on the same abstraction level and do not introduce additional platform-specific properties into the model description until a vertical transformation step is applied. As the implementation requirements and constraints change over time, we have to modify the used models. The consistency of the unchanged models therefore is ensured as they are logically connected on the same abstraction level and not at a lower level. If we choose a flexible target MoC in terms of timing notation, state notation, state transitions, and hierarchical design, just to name a few, it is quite easy to create those transformation engines. Thus, horizontal transformations are *platform independent*.

In contrast to vertical transformations it is not required to provide model transformations for each utilized combination of source and target model. Instead, it is possible create transitive mappings. This is illustrated in Fig. 2.2, where an FSM system description is first transformed into an Petri Net representation which is then mapped into an DEVS model. The DEVS formalism of Zeigler et al. [58] is a very good target MoC for horizontal transformations: It is highly flexible and several MoCs can easily be transformed into a DEVS model description. This fact has already been exploited in previous work, e.g., Risco-Martín et al. [45] describe an UML statecharts transformation into DEVS and Bobeanu et al. [9] present a Petri Net conversion.

### 2.1.2 Design Flow Methodologies for Reconfigurable Systems

Most existing design flows for reconfigurable hardware systems are directly based on low-level implementation languages like C or hardware description languages like VHDL and Verilog. They build upon the bottom abstraction layer. It is necessary to manually partition the design into the reconfigurable components and to schedule the reconfiguration of these components. Afterwards, a classical design process is performed for each single reconfigurable solution and each component that has been scheduled and partitioned. In addition, it is necessary to create and to implement the on-chip infrastructure to support the required reconfigurable functionality.

The low abstraction level is an even bigger problem when complexity is further increased by the introduction of the reconfigurable features. As a first approach to raise the abstraction level, the programming language SystemC as high-level modeling language has been introduced. SystemC aims towards the combined development of HW and SW. It is widely known and has become a de facto standard for system-level design.

While it is possible to implement system models directly in SystemC, it is highly appropriate to increase the abstraction level even further and to represent a system design on top of several MoCs. These models provide a more abstract system view, because SystemC has to integrate model specific features in a single, C-compatible programming language. In contrast, MoCs can be defined more formally without the limitations of a specific programming language.

A formal MoC for reconfigurable hardware systems has to address two different problem domains:

- First, it has to support the flexibility in system behavior that stems from dynamic reconfiguration. This objective is well-covered by most MoCs for software development. Most hardware-centric MoCs, however, do not support such behavior as the structure and functionality of conventional hardware remains static all the time.

- Secondly, the concurrency and time annotated behavior of hardware has to be modeled as well. This aspect is handled very well by most existing hardware models, while software models rarely consider such properties.

Thus, we have to address both domains in one single MoC to obtain a fully operational *RecDEVS* model. It is reasonable to extend an existing MoC with the approriate features instead of creating a new model from scratch. Therefore, we have

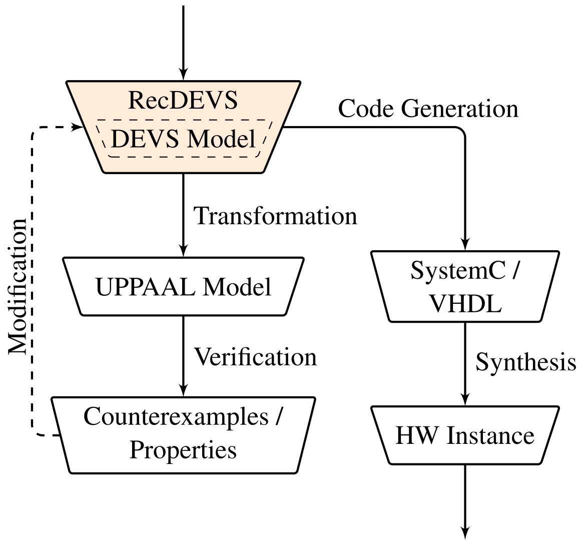


Figure 2.3: Proposed Design Flow for Verifiable Reconfigurable Systems

chosen DEVS (respectively its extension Dynamic Structure DEVS) as the fundamental MoC. The main reason for this decision lies in the clear and formal specification that makes DEVS well-suitable for further extensions. Furthermore, it provides support for relevant hardware properties such as concurrency and timed behavior. The DEVS formalism and the developed *RecDEVS* extension are presented in detail in Chapter 3.

### 2.1.3 Verification-based Design Flow

An essential aspect of a MoC-based high-level design methodology is the consideration of formal verification methods for system design. Such methods include mathematically-based languages, techniques, and tools for modeling, specification, and verification. Their purpose is to ease the development of complex systems where unintended system behavior may otherwise not be detectable due to the overall complexity of the system. Compared to testing techniques, a formal verification technique is an exhaustive process that can cover the whole possible system behavior whereas testing techniques can only explore a limited set of user-specified test cases. As Gupta [24] and Lam [31] summarized, the main goal of formal verification is revealing design errors by proving a relationship between an implemented model and an user-specified system behavior. Hence, with the help of formal verification, it is possible to prove whether the implementation satisfies a desired specification or not.

Formal methods require an exact specification, mostly in form of a formally defined Model of Computation that is used for system description. The verification of informally defined systems (e.g., by using a low-level implementation language like VHDL or C++) is possible as well, but only by retroactively annotating a formal behavior. However, if the original system specification may be ambiguous or too complex for a formal verification approach, the retroactive annotation is not always feasible. Thus, it is highly appropriate to start system-level entry with formally defined MoCs. This fact is established as next requirement for the subsequent parts of this work as Lemma 2.

**Lemma 2 (Formal Specification)** *The Model of Computation for a reconfiguration specific design flow must be formally specified to enable verification.*

There are various verification techniques, ranging from automated higher-order theorem provers to model checking tools. The model checking approach is suitable for the verification of systems, which can be represented by nondeterministic processes with finite control structures and real-valued clocks (i.e., Timed Automata). Compared to theorem proving approaches it does not require any user-interaction. The model checker can also produce counterexamples when the implemented model does not satisfy the required specifications. The counterexample generation is another important feature that often motivates the utilization of model checking tools over theorem provers which cannot produce counterexamples in the same manner. For the verification of reconfigurable hardware systems the automated model checking approach seems more promising and will be carried forward.

In the design flow presented throughout this work we combine two specialized MoCs. The *RecDEVS* MoC captures the functionality of reconfigurable hardware systems. It focuses on reconfiguration and provides specific properties for the description of such features. The UPPAAL Model of Computation is especially designed for the UPPAAL model checking environment and includes valuable verification expertise and knowledge. This combined approach was chosen because it turned out to be easier to exploit two specialized MoCs instead of including all required features into a single MoC.

To enable the verification of reconfigurable models described by *RecDEVS* a model transformation between *RecDEVS* and UPPAAL is necessary. If the transformation preserves all important *RecDEVS* model properties, then any verification statement about the transformed UPPAAL model will also hold for the equivalent *RecDEVS* system.

A transformation from a *RecDEVS* specification into an equivalent UPPAAL model representation can thus obtain verification results at an early stage of the design process. They can then be used to further refine the implementation until all desired verification properties are met. The quality and exactness of the model transformation is of essential significance for UPPAAL verification results representing the original *RecDEVS* model. If there is not only a transformation from *RecDEVS* into UPPAAL, but also a back transformation the other way around, then it is possible to change the model in UPPAAL (e.g., after finding a bug during verification phase) and directly back-annotate the changes into *RecDEVS*. If no such transformation is available, then the flawed system properties have to be re-identified in the original *RecDEVS* model and changed there manually.

If the transformation itself is not completely accurate and not all model properties are transformed correctly the results of the UPPAAL verification do not necessarily hold for the equivalent *RecDEVS* model as well. However, even then the verification process can serve as an indicator for potential problems. In this case the developer has to reassure that the verification problem really exists in the original *RecDEVS* model or has been introduced by an incorrect model transformation step. Also, a correctly verified property in the UPPAAL model provides no guarantee that the corresponding property in the original model is also correct if the transformation step is incorrect. But even then, the verification process provides huge benefits for the overall system development of complex systems. The more exact the model transformation is, the more exact will the verification results represent the behavior of the *RecDEVS* system. It is furthermore possible to improve the model transformation each time a mismatch between the behavior of both models is detected without changing anything else in the overall design flow.

In this work, an automated horizontal transformation from arbitrary *RecDEVS* models into UPPAAL has been developed and utilized. The model transformation will be explained in detail in Chapter 4. A proposal for a designed flow to enable the verification of reconfigurable systems is illustrated in Fig. 2.3. There is no formal proof available that the presented model transformation is always correct, thus the verification results can only be used as indicators for the correctness of the *RecDEVS* behavior as described above. Until now, no mismatch between the UPPAAL and the *RecDEVS* model has ever occurred throughout the development of this work and the implementation of the examples presented in Chapter 5. There is also currently no back transformation from UPPAAL to *RecDEVS* available, but the re-identification of system flaws in *RecDEVS* was never a problem due to the close relationship between these models.

## 2.2 Reconfigurable Architectures

In the following, the concept of reconfigurable hardware and some of the underlying theoretical ideas are explained. This is important as the terms and definitions are not consistent in the scientific community and are sometimes ambiguously used.

In general, reconfiguration can be regarded as a special kind of dynamic behavior of previously static hardware structures. It is possible to distinguish three different kinds of dynamic behavior in a computational model:

**Dynamic Values:** The most general example for dynamically changing values inside a computational model is the system state. Only reactive, stateless systems can be modeled without the need to save the current system state over time. All other computational models provide mechanisms to change the system state and other stored values during execution.

**Dynamic Communication Structure:** One common approach to ease system design is the creation of more coarse granular modules that can then be described independently. Interaction between such modules uses a clearly specified communication interface and a fixed communication structure. Dynamically changing communication structures can thus change the communication topology during runtime.

**Dynamic Functionality:** The functional behavior of a system description can also be adopted dynamically during runtime. For module-based computational models this is mainly realized by the creation (or deletion) of functional modules. In this case a functional change comes together with a modification of the communication structure that reflects the new functionality. The functionality inside the individual module remains static, however.

While *Dynamic Values* are part of most computational systems, this is not the case for the other two behaviors. In software-centric enviromnments the concept of *Dynamic Communication Structures* is realized by address manipulation. For more abstract models like object-oriented software *Dynamic Functionality* can be modeled by the instantiation of new software modules. In that case object pointers are modified, transmitted, or deleted to change the communication topology.

However, in hardware-centric environments (namely integrated circuits) the functional description and the communication structure is part of the fixed circuit. They can not be modified without the production of a new chip. To keep the hardware-centric system descriptions compact, *Dynamic Communication Structures* and *Dynamic Functionality* have been left out from the corresponding computational models. As we will realize reconfigurable hardware overcomes this limitation and enables all three kinds of dynamic behavior for hardware.

While this overview might suggest that communication structure and dynamic functionality may be merged, we will see in the following that the separation of both is helpful for the description of reconfigurable hardware systems.

### 2.2.1 Behavior and Structure of Reconfigurable Hardware

A conventional hardware architecture is characterized by a completely static structure. While the contents of memory elements like RAM and flipflops may change, the layout of the chip and the connections between the logic elements always remain static. In contrast, reconfigurable hardware introduces a dedicated reconfiguration process that may change functionality of all hardware resources within a chip. Thus, reconfiguration allows the reuse of hardware resources by different operations.

A reconfigurable hardware system can be defined by a set of functional components. Each functional component utilizes a certain amount of available resources, may it be computational resources, routing resources, or memory resources available in the reconfigurable hardware system. The process of reconfiguration changes the instantiated functional components. This process can create additional functional components if there are enough systems resources available, it can replace or remove existing components to release utilized system resources. The granularity of the functional components may reach from a fine granular architecture, where the functionality of single flip flops (e.g., it's reset values) may change, up to coarse granular architectures where the computational elements are DSP cores and reconfiguration changes the programs executed by these cores.

Reconfiguration consists of two different basic operations on which this work focuses. All other operations can be build upon these two operations as given by Lemma 3.

**Lemma 3 (Reconfiguration Operations)** *A reconfigurable system requires a New-operation and a Delete-operation to perform reconfiguration activities. Both operations require associated information on the target component that should be created or removed from the system resources.*

There may be additional operations to ease the description of reconfigurable systems. This work focuses on the two essential operations from Lemma 3 as other operations can be constructed by them. A *move*-operation, for example, which moves an existing functional component to other system resources and may ease potential fragmentation issues for the resources of reconfigurable systems, can be modeled by a direct sequence of a delete-operation followed by a new-operation.

It is one essential goal of this work to develop a design flow methodology for reconfigurable systems that supports verification. To address reconfiguration features in such a design flow the following lemma will be required in the following chapters:

**Lemma 4 (Self-Contained Model)** *Specification of reconfigurable behavior, represented by the New-operation and the Delete-operation has to be a part of the functional specification model. This allows a verification process to reason about the impact of functional elements on reconfigurable systems.*

This lemma addresses another approach to formal specification discussed in Bondalapati and Prasanna [10], where reconfiguration and functional behavior have been split into two disjunct MoCs. In such a case it is not easily possible to reason about the effect of a functional result on a subsequent reconfiguration step.

We pick up some important definitions of that work and define several classes of reconfiguration, namely *statically*, *partially*, and *dynamically reconfigurable* architectures.

**Statically reconfigurable** architectures allow the reshaping of a hardware system as a whole. This situation is depicted in Fig. 2.4. As a consequence, static reconfiguration does not allow any data persistence across reconfiguration as this would require to keep back some areas where the persistent data can be stored. The design flow for statically reconfigurable architectures is performed by repeating conventional design flows multiple times. Possible examples for this kind of application is the in-field deployment of hardware patches etc.

**Partial reconfiguration** denotes the situation where only a smaller part of the whole chip is being reconfigured. In this case the remaining part of the chip preserves both its functionality and data. This kind of reconfiguration adds a new dimension to the hardware design space. System design is no longer limited to the space domain (routing and placement of modules), but is extended to time domain (by reusing resources over time).

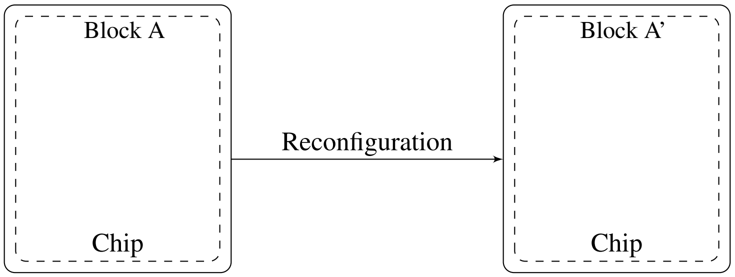


Figure 2.4: Static Reconfiguration

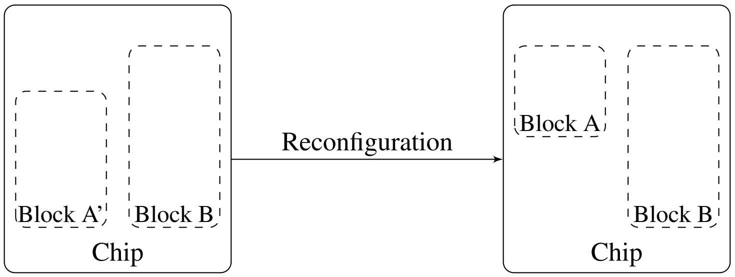


Figure 2.5: Partial and Dynamic Reconfiguration

**Dynamic reconfiguration** is a special case of the partial reconfiguration. While some part of the chip is being reconfigured, the other components continue their execution. This situation is depicted in fig. 2.5.

### 2.2.2 Towards a Model of Computation for Reconfigurable Hardware

The creation of a new Model of Computation is a complex and time consuming process. It seems natural to reuse existing solutions and research results. As already mentioned, a formal MoC for reconfigurable systems has to integrate two different concepts found in other Models of Computation too:

- The flexible and dynamic system behavior that previously may only be found in software systems (here stemming from from dynamic reconfiguration).

- Hardware-centric features like concurrency and cycle accurate time-annotated behavior.

As a special class of hardware-centric systems, reconfigurable hardware systems share some basic modeling principles with all other hardware-centric systems. One key benefit compared to software-centric systems is the ability to improve the system performance by parallel computations. This requires the possibility to describe concurrency inside the system description. Practical experiences with various hardware description languages and models have shown that it is normally desirable to also offer the possibility to model small sequential blocks, which are then executed in parallel.

**Lemma 5 (Concurrency)** *The components of a reconfigurable system run concurrently to each other and sequentially inside themselves.*

Another special property of hardware centric systems is the ability to support cycle-accurate system models. In the more general case this concept can be extended towards a requirement for time-annotated models. These models can be interpreted as cycle accurate if the time granularity is chosen appropriately small. While it may be possible to imagine hardware systems, that work without any time-annotation the availability has no direct disadvantages. Thus, the intended model for reconfigurable hardware systems should support it as well.

**Lemma 6 (Time Annotation)** *A formalism for reconfigurable hardware systems should be able to model timed behavior.*

At the beginning of this work is was not completely clear if the development of a MoC for reconfigurable systems should start with a highly-dynamic software-centric model that will then be extended towards timed-behavior and concurrency or vice-versa with a hardware-centric model that has to be extended towards dynamically changing behavior. The second approach has been chosen and started with a hardware-centric model for two reasons. First, the retroactive extension of an MoC with timing annotation seemed more difficult than the other way around. Secondly, the internal behavior of reconfigurable components is, in most cases, a straight-forward hardware implementation and can thus be easily modeled within a hardware-centric MoC.

### 2.2.3 Field Programmable Gate Arrays

The most established platform for reconfigurable hardware systems are the *Field Programmable Gate Arrays (FPGA)*. While the general concept of this work is applicable to every reconfigurable system, certain design considerations for developing the *RecDEVS* MoC take in account platform features.

This work focuses on reconfigurable hardware systems with a fine granularity. To motivate certain design decisions in the development of *RecDEVS*, we first give a short overview over such systems.

An FPGA consists of an large array of small reconfigurable elements, the so-called *Configurable Logic Blocks (CLB)*. A logic block is build up by a logical computational element, a memory element and additional resources speed-up dedicated operations (e.g., the carry-chain of an addition). Depending on the architecture, the computational element can be realized in form of an Look-Up-Table (LUT) or by a set of multiplexers. All logic cells are connected via a flexible switch matrix, which is able to connect different logic cells together and thus build up arbitrary complex functions. The actual functionality of an FPGA is defined during the configuration. In this process the computational elements are set up to perform the approriate functions, the switch matrix enables the required connections between the utilized logic cells and all additional logic resources are configured according to the requirements as well. The overall structure of a typical FPGA is depicted in Fig. 2.6.

The configuration data inside the FPGA can be inserted on top of an SRAM-based technology or an Anti-Fuse-based technology. This work focuses on the SRAM-based technology as it provides an substantial benefit compared to Anti-Fuse. The functionality can easily be replaced by a new one, simply be changing the content of the reconfiguration SRAM, thus enabling reconfiguration.

It must be noted that the reconfiguration process of existing hardware architectures is multiple times slower than the actual computation inside the computational elements. Thus, it is reasonable for a MoC to distinguish between reconfiguration and normal system execution. This allows to handle both processes separately in a design methodology for reconfigurable systems. For faster architectures, it is easily possible to merge both processes if no distinction is required.

**Lemma 7 (Network Topology)** *Reconfiguration should distinguish between a change in functionality (Reconfiguration), a value change (Storage) and routing changes (Communication).*

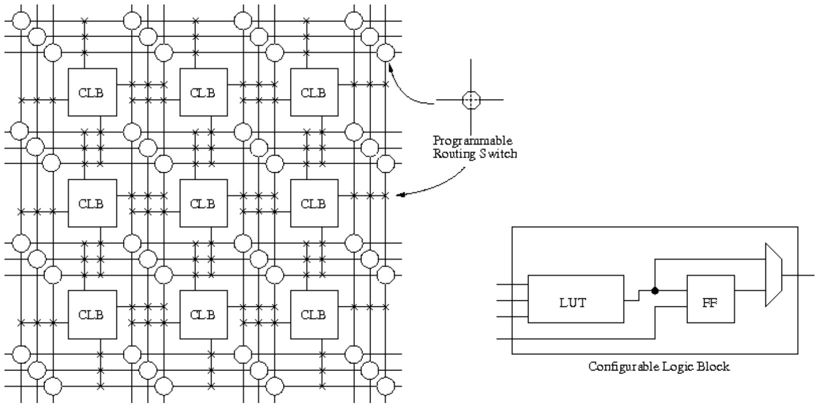


Figure 2.6: Basic Structure of Field Programmable Gate Arrays

Reconfiguration is limited by present hardware architectures. The reconfigurable hardware resources are formed into blocks of a certain minimal size. This is justified by the various design rule checks required for reconfigurable hardware. Reducing the granularity of these reconfigurable blocks too far would increase the complexity of design rule checks beyond feasibility. Reconfiguration can then be summed up into the *creation* and the *deletion* of those reconfigurable blocks. This may also be applied to multiple components at the same point in time.

Together with the runtime overhead to perform an actual reconfiguration this leads to component sizes that must not be too fine granular. This is specified by lemma Lemma 8.

**Lemma 8 (Granularity)** *Due to the reconfiguration times, reconfiguration requires a certain coarser granularity. Thus, reconfiguration should be performed on a module-based level.*

Communication between different resources is another critical limitation of present dynamically reconfigurable architectures. In order to assure signal integrity, the connections between different resources are deactivated during reconfiguration. Typically, this task is performed automatically by the design tools. However, this requires that the interface of the reconfigurable blocks is reasonable small and simple.

**Lemma 9 (Encapsulation)** *The reconfigurable modules should be self-contained. Interaction with the external enviroment should be realized with a limited and explicitly specified interface.*

In most systems, reconfiguration is triggered by an external component which holds the different reconfiguration informations and performs the required interface protocols with the FPGA. A new approach is the *In-System-Configuration* (ISC) (Jacobson [28]) that enables an FPGA to reconfigure certain parts of it itself. This allows for the realization of an FPGA as a stand-alone reconfigurable hardware platform.

FPGAs do not provide inherent management for the reconfigurable resources. This means that different configuration data is required depending on the exact resources a component should utilize on the FPGA. The resources have to be specified explicitly and it is not possible to just demand to place the component onto free resources.

Such a resource management, together with the management of configuration data for the different functional components, is generally handled by an user-implemented *Arbiter* component. Such a component will also manage the reconfiguration interface of an ISC architecture. Technical realizations of existing FPGAs provide only one single ISC interface. Consequently, the implementation of a central Arbiter component is reasonable.

This results in a certain problem for a generic MoC for reconfigurable systems. To be platform-independent such a MoC should not require a special arbiter realization. Furthermore, it is desirable to be able to trigger the reconfiguration process from each component.

**Lemma 10 (Trigger Reconfiguration)** *All components should be able to trigger a reconfiguration. Because of Lemma 9, this operation should be part of the components interface.*

From an user-based point of view this lemma allows to abstract all arbiter components that might be required by an actual reconfigurable platform. Furthermore, the local reconfiguration process allows a modular based design-flow. A system designer needs to implement his own components independently. This would not be possible if all components incorporate their own arbiter implementation. In that case potentially colliding requests from multiple reconfiguring components would have to be considered. In the MoC such parallel requests will be handled by the envisioned MoC formalism and not by the system designer.

# 3 RecDEVS: Model of Computation for Reconfigurable Systems

As already stated in the previous chapter a MoC based design flow provides many benefits. Consequently, Lemma 1 demands that a design flow methodology for reconfigurable systems will also be based on MoCs. However, in the area of reconfigurable hardware systems there is a certain lack of appropriate MoCs. Section 3.3.1 explains why the already existing models are not sufficient for modeling reconfigurable hardware systems.

Therefore, in this chapter a novel Model of Computation for reconfigurable systems, *RecDEVS*, is developed. It is based the DEVS formalism from Zeigler et al. [58] which does already provide a lot of the desired features. It is highly flexible and already provides a complete formally specified behavior that eases the creation of model transformations. Thus, other MoCs can be easily transformed into a DEVS model description.

We will see that DEVS already covers many of the non-reconfiguration specific requirements, which have been formulated as lemmas in the previous chapter. It is cleanly and formally specified and therefore suitable for verification as required by Lemma 2. It is also time-annotated and supports concurrency as required by Lemma 5 and 6, respectively.

The rest of this chapter is structured as follows. In Section 3.1 the existing DEVS formalism is introduced together with various extensions. In addition, a graphical representation of DEVS models is developed that will be used subsequently. In Section 3.2 the different DEVS extensions that are important for the development of *RecDEVS* are combined in a single *Reference DEVS* that works as a foundation for the following development of the *RecDEVS* Model of Computation for Reconfigurable Systems detailed in Section 3.3. *Reference DEVS* will not only combine all important extensions, but it will also cover certain ambiguities that exist in the original specification.

## 3.1 DEVS Model of Computation

The DEVS formalism is a powerful mathematical foundation for specifying hierarchically, concurrently executed formal models. It is organized in a layered structure which starts with an atomic model that exists as a stand alone system and is extended towards the concurrent execution of multiple hierarchically coupled components. We will follow this definition scheme and first explain the DEVS formalism basic components and then its various important extensions.

The DEVS formalism is created as an event-based, timed Model of Computation. Each component consists of a local time $t$ that contains the elapsed time since the actual state has been entered. The local times of all components in a complex DEVS system advance simultaneously. The DEVS formalism uses the time to define certain timeouts, that force state transitions.

### 3.1.1 Classical DEVS

A classical atomic DEVS component $\mathrm{C}^{classic}$ is described by the tuple

$$\begin{aligned}
\mathrm{C}^{classic}_{[name]} &= (X_{[name]}, Y_{[name]}, S, \delta_{\mathrm{int}}, \delta_{\mathrm{ext}}, \lambda, \tau).\\
X_{[name]} &: \text{Input Type}\\
Y_{[name]} &: \text{Output Type}\\
S &: \text{Set of States}\\
\delta_{\mathrm{int}} &: S \to S\\
\delta_{\mathrm{ext}} &: Q \times X \to S \text{ where } Q = \{(s,e)|s \in S, 0 \le e \le \tau(s)\}\\
\lambda &: S \to Y\\
\tau &: S \to \mathbb{R}^+
\end{aligned}$$

This formalism is similar to usual finite state machines. The state set $S$ is a non-empty set of states where all elements must be distinct. A component C has an external interface defined by its input set $X$ and output set $Y$. Input and output events occur upon state transitions. Every state has an associated timeout $\tau : S \to \mathbb{R}^+$. The optional *[name]* allows to differ between multiple components. It can be omitted if the correct component can be derived from the context of the given formula. Such an optional naming tag exists for all subsequent DEVS extensions, however, it will not be defined separately for each definition in the sequel.

Functional behavior in DEVS is realized via state transitions. The DEVS formalism differs between different transition types for various system scenarios. For the classical DEVS component there are two state transitions defined.

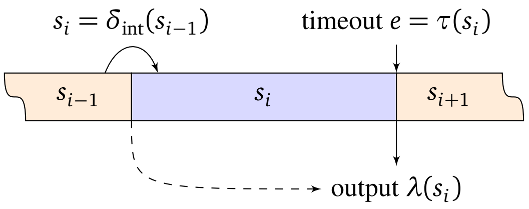


Figure 3.1: State Transition and Output

**Internal Transition** $\delta_{\text{int}} : S \rightarrow S$**:** After the timeout $\tau(s)$ of a certain state occurred, i.e., the local time of the component has been advanced to this point in time, the component will do an internal state transition. After the transition a new timeout is set by the new state.

**External Transition** $\delta_{\text{ext}} : Q \times X \rightarrow S$**:** Iff the component receives an input event on the input $X$ it will do an external state transition $\delta_{\text{ext}}((s, e), x)$. $\delta_{\text{ext}}$ has knowledge about the elapsed time $e < \tau(s)$ since the last state transition occurred. After the transition the local time is set to 0 and the new timeout is derived from $\tau(s')$ for the new state $s'$.

Whenever a timeout $\tau(s)$ is hit, the component will emit the output $\lambda(s)$. This means that the output is bound to the end of an internal transition. An external transition can only be activated before the timeout occurs, because at each timeout the internal transition will switch the system state and set a new timeout. Consequently, an external transition can not directly generate an output. Section 3.2.3 will explain why this is no limitation of the DEVS formalism.

As DEVS is an event-based model, its output is both, an output value and a corresponding event at the same time. The output may also be the absent event $\diamond$. As depicted in Fig. 3.1 an output occurs upon leaving a state $s_i$ to the next state $s_{i+1}$, although its value is determined upon entering $s_i$ from the previous state $s_{i-1}$.

If no explicit transition is given, DEVS defines implicit transitions $\delta_{\text{ext}}((s, e), x) = s$ and $\delta_{\text{int}}(s) = s$, i.e. the system remains in its current state.

## DEVS$^{classic}$ Example

In the following an example for a classical DEVS component is given. This component comprises a variable $n$ that is emitted and incremented every 10 time units. In addition, it is possible to update the variable $n$ with an external input. In this case an output of the new value $n$ is generated immediately.

$$
\begin{aligned}
C_{ex1}^{classic} &=< X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \tau > \\
X &= \mathbb{N} \\
Y &= \mathbb{N} \\
S &= \underbrace{\{\text{“wait”}, \text{“update”}\}}_{phase} \times \underbrace{\mathbb{R}^+}_{\text{timeout } t} \times \underbrace{\mathbb{N}}_{\text{variable } n} \\
\delta_{\text{int}}(phase, t, n) &= \begin{cases} (\text{“wait”}, 10, n+1) & \text{if phase=“wait”} \\ (\text{“wait”}, t, n) & \text{if phase=“update”} \end{cases} \\
\delta_{\text{ext}}(((*, t, n), e), x) &= (\text{“update”}, t-e, x) \\
\lambda(*, *, n) &= n \\
\tau(\text{“wait”}, t, \ldots) &= \begin{cases} t & \text{if phase=“wait”} \\ 0 & \text{else} \end{cases}
\end{aligned}
$$

$*$ is used as a wildcard parameter. This parameter can have any value without influencing the result of the function.

This example demonstrate a typical behavior of all DEVS models. The output value of a DEVS model does only depend on the current state $s$ and it is only generated if a timeout occurs.

The output does not depend on the input set $X$. Therefore, it is necessary to include the “update”-state in its definition. Here, we have the updated state that includes the new external value $x$ that can now be written to the output.

Another typical aspect are external transitions with a timeout of the kind $t - e$. With this pattern the timeout of the originating state (“wait” in this example) can be preserved. For this, the timeout has to be saved as part of the state. Without such an approach all external events would reset the timeout as we can not directly modify $\tau$ depending on $e$.

In this case the context of the functions as elements of $C_{ex1}^{classic}$ is clear and thus the optional *[name]* is omitted.

## DEVS$^{classic}$ Graphical Notation

The DEVS models can also be denoted by an equivalent graphical representation.

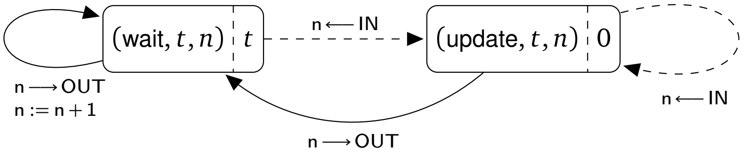


Figure 3.2: Graphical Representation of DEVS$^{classic}$

Each box denotes a system state with an associated timeout that is given in the right side of the box. The dashed arrows depict the external transitions and the normal arrows internal transition. All internal transitions may emit an output event that is denoted as “x → OUT” in the graphical representation. All external transitions receive an input event that is received and saved in an element of the system state with “$n \leftarrow IN$”.

In this example it would not be possible to draw all possible system states explicitly, as $t$ and $n$ are elements of an infinite set. The graphical representation allows to denote such elements as variables instead of explicit values as shown in the example. To change these state elements all transition may have an additional update label like $n := n + 1$ in the example.

## 3.1.2 The DEVS Extension for Ports

In the next step the classical DEVS formalism is extended towards a distinction between different inputs and outputs. This is performed by the introduction of *ports*.

A classical component $\mathrm{C}^{classic}$ is extended towards $\mathrm{C}^{ports}$ by

$$\begin{aligned}
\mathrm{C}^{ports} &= \mathrm{C}^{classic} \cup \mathit{\{InPorts,\ OutPorts\}} \text{ with} \\
\mathit{InPorts} &: \text{Enumeration of Input Port Names} \\
\mathit{OutPorts} &: \text{Enumeration of Output Port Names} \\
X^{ports} &= \{(p, v) | p \in \text{InPorts}, v \in X^{classic}\} \\
Y^{ports} &= \{(p, v) | p \in \text{OutPorts}, v \in Y^{classic}\}
\end{aligned}$$

Here, the input and output sets $X$ and $Y$ of $\mathrm{C}^{classic}$ are denoted as $X^{classic}$ and $Y^{classic}$ for better readability. This means that the input and output variables are extended into tuples of the form $(port, data)$. These port names ease communication and model description. They allow to differentiate between the communication structure and communication data. This is of special importance in the subsequent definitions with multiple interacting DEVS components.

By the given construction of ports, all ports share one common data type. All input ports have the data type $X^{classic}$ and the output ports $Y^{classic}$. However, this is no requirement of the DEVS formalism and it would be easily possible to extend the formalism for distinct data types for all ports if necessary.

### DEVS$^{ports}$ Example

To illustrate the $DEVS^{ports}$ the example of the classical DEVS formalism is picked up. Two input ports are introduced, however the system should only react on the port “in2”. Only one output port “out1” is defined.

$$
\begin{aligned}
C^{ports}_{ex1} &= < X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \tau > \\
InPorts &= \{\text{“in1”}, \text{“in2”}\} \\
OutPorts &= \{\text{“out1”}\} \\
X &= \{(p_{in}, v_{in}) | p_{in} \in InPorts, v_{in} \in \mathbb{N}\} \\
Y &= (p_{out}, v_{out}) | p_{out} \in OutPorts, v_{out} \in \mathbb{N} \\
S &= \underbrace{\{\text{“wait”}, \text{“update”}\}}_{phase} \times \underbrace{\mathbb{R}^{+}}_{\text{timeout } t} \times \underbrace{\mathbb{N}}_{\text{variable } n} \\
\delta_{\text{int}}(phase, t, n) &= \begin{cases} (\text{“wait”}, 10, n+1) & \text{if phase=“wait”} \\ (\text{“wait”}, t, n) & \text{if phase=“update”} \end{cases} \\
\delta_{\text{ext}}(((*, t, n), e), (p_{in}, v_{in})) &= (\text{“update”}, t - e, v_{in}) \text{ if } p_{in} = \text{“in2”} \\
\lambda(*, *, n) &= (\text{“out1”}, n) \\
\tau(\text{phase}, t, \ldots) &= \begin{cases} t & \text{if phase=“wait”} \\ 0 & \text{else} \end{cases}
\end{aligned}
$$

## DEVS$^{ports}$ Graphical Notation

For the DEVS$^{ports}$ formalism the generic keywords “IN” and “OUT” are replaced with the corresponding port name $p$. Thus, an transition label $n \leftarrow in2$ refers to the value $n$ that is received via the port $in2$. It is also possible to generate multiple outputs with one transition, which would be noted by multiple output lines.

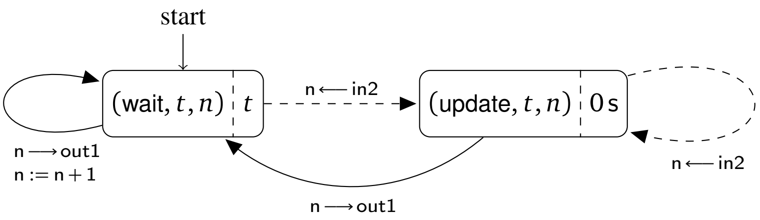


Figure 3.3: Graphical Representation of a DEVS$^{ports}$ model

### 3.1.3 Coupled DEVS

With help of the port formalism it is now possible to combine multiple DEVS components into a DEVS network system $\mathbb{S}$. The previously defined elementary components are denoted as *atomic* components.

Each network description includes a set $D$ of unique identifiers. These identifiers may be identical with the optional *[name]* of an atomic component. They are used to identify the components coupled together in $\mathbb{S}$. The *Coupled DEVS* formalism connects an arbitrary set $C_d \in D$ with each other by:

$$\begin{aligned}
\mathbb{S}^{coupled} =&< X^{Port}, Y^{Port}, D, \{C_d^{Port} \mid d \in D\}, EIC, EOC, IC, Select > \text{ with} \\
D &: \text{Set of Component Identifiers for } C_d^{Port} \\
EIC &\subseteq \{(x, (C_d, p_{in}) \mid x \in X^{Port}, d \in D, p_{in} \in InPorts_d\} \\
EOC &\subseteq \{(y, (C_d, p_{out}) \mid y \in Y^{Port}, d \in D, p_{in} \in OutPorts_d\} \\
IC &\subseteq \{((C_d, p_{in}), (C_{d'}, p_{out})) \mid d, d' \in D, p_{in} \in InPorts_d, p_{out} \in OutPorts_{d'}\} \\
Select &: 2^{|D|} - \{\} \to D
\end{aligned}$$

The distinction between the model identifier $d$ and the actual component $C_d$ allows to instantiate multiple components of the same kind. Thus $D$ can be interpreted as the *type* of the component.

The coupled DEVS network features one common set of input and output ports $X$ and $Y$, respectively. These external ports are linked to corresponding component ports $C_d$ by the *External Input Coupling (EIC)* and the *External Output Coupling (EOC)*. The *Internal Coupling (IC)* is used to link the ports of two internal components without any interaction with the global external interface of $\mathbb{S}$. Port couplings are statically defined and cannot be changed throughout the execution of the system. This means that all communication channels which may be used during runtime have to be defined initially.

The *Select* function provides an order of execution for all DEVS components. In the Coupled DEVS formalism only one transition can be active at a time. With multiple independent components in one system it might occur that multiple transitions are activated at the same time. In such cases the *Select* function works as a resolution function and defines the next active component. This can be compared to similar formal execution models like Petri Nets.

Note that the *Select* function is not part of any atomic DEVS component. Instead, it has to be provided as a global realization aspect of the complete network.

It is also possible to hierarchically instantiate other DEVS networks $\mathbb{S}$ in addition to atomic components. This extends the *Coupled DEVS* towards a hierarchal system of components. However, as this feature is not required for the development of *RecDEVS*, it will not be detailed in this work.

## DEVS$^{coupled}$ Example

To illustrate the coupled DEVS formalism two components $\mathrm{C}_{ex1}$ and $\mathrm{C}'_{ex1}$ are connected. The output "out" of the component $\mathrm{C}_{ex1}$ is used as an input port for $\mathrm{C}'_{ex1}$. All remaining ports of both components are tied to the global port interface.

$$\begin{aligned}
\mathbb{S}_{ex1} &=< X, Y, D, \{\mathrm{C}_{ex1}, \mathrm{C}'_{ex1}\}, EIC, EOC, IC, Select > \\
X &= \{(port, val) \mid port \in \{\text{“gIn1”}, \text{“gIn2”}, \text{“gIn3”}\}, val \in \mathbb{N}\} \\
Y &= \{(\text{“gOut”}, n) \mid n \in \mathbb{N}\} \\
D &= \{ex1\} \\
EIC &= \{(\text{“gIn2”}, (\mathrm{C}_{ex1}, \text{“in1”})), (\text{“gIn3”}, (\mathrm{C}_{ex1}, \text{“in2”})), \\
&\qquad (\text{“gIn1”}, (\mathrm{C}'_{ex1}, \text{“in1”}))\} \\
EOC &= \{(\text{“gOut”}, (\mathrm{C}'_{ex1}, \text{“out”}))\} \\
IC &= \{((\mathrm{C}_{ex1}, \text{“out”}), (\mathrm{C}'_{ex1}, \text{“in2”}))\} \\
Select(M) &= \begin{cases} \mathrm{C}_{ex1} & \text{if } \mathrm{C}_{ex1} \in M \\ \mathrm{C}'_{ex1} & \text{else} \end{cases}
\end{aligned}$$

## DEVS$^{coupled}$ Graphical Representation

For a graphical representation of *Coupled DEVS*, it is possible to use conventional block diagrams. For the given example the block diagram is depicted in Fig. 3.4.

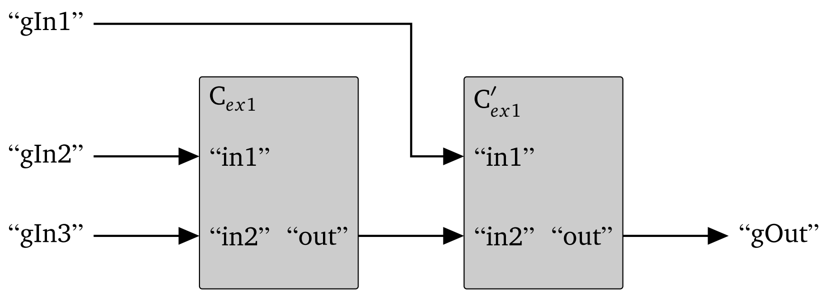


Figure 3.4: Block Diagram of DEVS$^{coupled}$

### 3.1.4 Parallel DEVS

The *Coupled DEVS* formalism does not support concurrent model execution. Whenever multiple concurrent transitions occur, they are serialized by the *Select* function. The *Parallel DEVS* formalism addresses this issue. It incorporates the component instantiation and port coupling formalisms of *Coupled DEVS* but it modifies the behavior of the transition functions. For this, the *Select* function is replaced by a third type of transition functions $\delta_{\text{con}}$. A *Parallel DEVS* system $\mathbb{S}$ is defined as

$$\begin{aligned}
&\mathbb{S}^{par} =< X^{Ports}, Y^{Ports}, D, \{\mathbb{C}_d^{par} \mid d \in D\}, EIC, EOC, IC > \text{ with}\\
&\mathbb{C}^{par} =< X^{Ports}, Y^{Ports}, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \tau > \text{ with}\\
&\delta_{\text{con}} : Q \times X^{Ports} \rightarrow S
\end{aligned}$$

#### The Confluent Relation $\delta_{\text{con}}$

In *Parallel DEVS* multiple transitions are executed in parallel when they are triggered at the same time. Thus, a *Select* function as it was utilized in *Coupled DEVS* is no longer required.

However, the concurrent execution of two different transition functions inside one atomic component is not possible. This is because the two transitions $\delta_{\text{int}}$ and $\delta_{\text{ext}}$

might calculate two contradictory new states leading to undefined system situations. In *Coupled DEVS* this is not been possible, because all transitions that ought to be executed at the same point in time are serialized by *Select*. For the resolution of this possible contradiction inside one atomic component, the additional *confluent relation* $\delta_{\text{con}}$ is introduced. It is triggered whenever two different transition functions would be called otherwise. These two transition functions are not activated in this case.

**Confluent Transition** $\delta_{\text{con}} : Q \times X^{Ports} \rightarrow S$**:** Iff the component receives an external event while an timeout occurs, the next state will be computed by $\delta_{\text{con}}((s,e),x)$ instead of $\delta_{\text{int}}$ or $\delta_{\text{ext}}$.

With the confluent function $\delta_{\text{con}}$ there is no longer a need for a centralized serialization system. All components can now run independently. The only interaction with the external environment is via the component interface, as demanded by Lemma 9.

Compared to *Coupled DEVS* the confluent transition is also more expressive. Considering a situation where $\delta_{\text{int}}$ and $\delta_{\text{ext}}$ should fire, it is only possible to select one of both functions. With the *Parallel DEVS* model it is possible to apply both functions by defining the confluent transition similar to $\delta_{\text{con}} = \delta_{\text{int}}(\delta_{\text{ext}}(q,x))$.

## $DEVS^{parallel}$ Example

Consider the example of $DEVS^{ports}$ again, it can be extended towards *Parallel DEVS* by defining the confluent transition function.

$$\begin{aligned}
\mathbb{S}^{par}_{ex1} &=< X, Y, D, \{\mathrm{C}_{ex1}, \mathrm{C}'_{ex1}\}, EIC, EOC, IC > \text{ with} \\
\mathrm{C}^{par}_{ex1} &= \mathrm{C}^{Ports}_{ex1} \cup \delta_{\text{con}} \\
\delta_{\text{con}}(((phase,t,n),e),x) &= \delta_{\text{ext}}(((phase,t,n),e),x)
\end{aligned}$$

## $DEVS^{parallel}$ Graphical Representation

In the graphical notation, confluent transitions are depicted by double-lined arrows. Their labels follow the conventions introduced by the internal and external transitions, Fig. 3.5 illustrates the usage of the component $\mathrm{C}^{par}_{ex1}$.

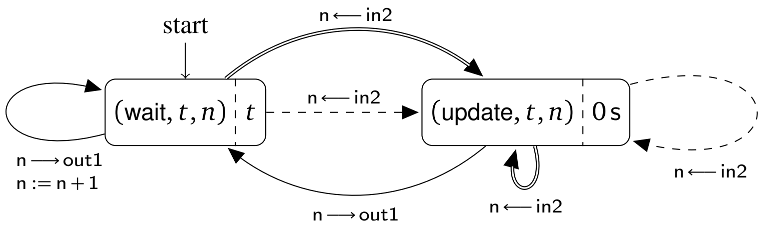


Figure 3.5: Graphical Representation of a DEVS$^{parallel}$ model

## 3.2 Reference DEVS Model of Computation

To have a consistent base for the development of a reconfigurable MoC the different DEVS variants have been unified towards a *Reference DEVS*. A Reference DEVS System $\mathbb{S}$ is given by the tuple:

$$\mathbb{S}^{ref} =< X^{ref}, Y^{ref}, D, \{\mathrm{C}_d^{par} \mid d \in D\}, EIC, EOC, IC > \text{ with}$$
$$\mathrm{C}^{ref} =< X^{ref}, Y^{ref}, S, s_o \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \tau >$$

This development covers three different goals:

- There is no standardized reference definition of a parallel DEVS formalism with a clear definition of ports. However, a clearly defined reference is of great benefit for defining research cooperations and public discussions.
- The original definitions contain some minor ambiguities. In some cases there are different possible interpretations of certain execution rules. While the differences are not crucial in many cases it has to be decided which interpretation is used throughout the following work.
- The original DEVS formalism realizes a *Timed Model of Computation*. However, the *RecDEVS* MoC that is to be developed has a clear focus on reconfigurable hardware systems that are clock synchronous. To achieve this goal, the *Reference DEVS* will be extended by an empty event $\diamond$ that is used for clock events without a corresponding DEVS event.

In the following, all functional elements are grouped together thematically and their functionality will be presented.

### 3.2.1 Component States

$$S : \text{Set of States}$$
$$s \in S : \text{Current State}$$
$$s_0 \in S : \text{Initial State}$$
$$\tau : S \to \mathbb{R}^+$$

In addition to the state set $S$ an additional dedicated *initial state* $s_o$ with $s_0 \in S \cap \{\emptyset\}$ is defined[1]. The other state specific functionality remains unchanged. The current state $s$ is left at latest after $\tau(s)$ via the transition functions $\delta_{\text{int}}$, $\delta_{\text{ext}}$, or $\delta_{\text{con}}$.

### 3.2.2 Ports and Port Coupling

$$
\begin{aligned}
X^{ref} &= \{(p,v) | p \in \text{InPorts}, v \in \{X^{classic} \cup \diamond\}\} \\
Y^{ref} &= \{(p,v) | p \in \text{OutPorts}, v \in \{Y^{classic} \cup \diamond\}\} \\
EIC &\subseteq \{(x, (\mathrm{C}_d, p_{in}) \mid x \in X^{Port}, d \in D, p_{in} \in InPorts_d\} \\
EOC &\subseteq \{(y, (\mathrm{C}_d, p_{out}) \mid y \in Y^{Port}, d \in D, p_{in} \in OutPorts_d\} \\
IC &\subseteq \{((\mathrm{C}_d, p_{in}), (\mathrm{C}_{d'}, p_{out})) \mid d, d' \in D, p_{in} \in InPorts_d, p_{out} \in OutPorts_{d'}\}
\end{aligned}
$$

As already mentioned this MoC introduces the absent event $\diamond$. The port definitions of the $DEVS^{ports}$ formalism have to be extended appropriately such that the values $v$ of each port tuple $(p, v)$ may now also contain $\diamond$.

### 3.2.3 Output Function

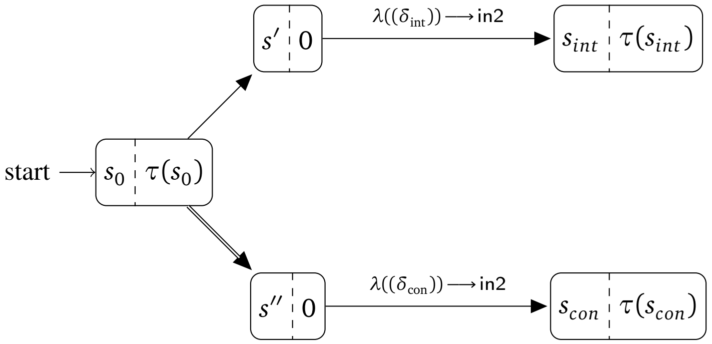
$$
\begin{aligned}
&\lambda : S \rightarrow Y^{ref} \\
&\Lambda : Q \rightarrow Y^{ref} \text{ with} \\
\Lambda(s,e) &= \begin{cases} \lambda(s) & \text{if } e = \tau(s) \\ \diamond & \text{if } e < \tau(s) \end{cases}
\end{aligned}
$$

The output function $\lambda$ generates output data on the different output ports defined by $Y^{ref}$. The output on each port can either be the empty event $\diamond$ or a real output

[1] The original DEVS defintion did not include the initial state, however the initial state has been used consistently throughout the literature on DEVS.

value. The output values on all different ports of $Y$ share one common output alphabet defined by $Y^{classic}$ of the port definition.

Non-empty output events may only be generated when the timeout $\tau(s) = e$ is reached. More precisely, the output only depends on the current state $s_0$ and is independent of the taken transition function $\delta_{\text{int}}$ or $\delta_{\text{con}}$, respectively. Consequently it is not directly possible to generate a transition dependent on the output values $\lambda(\delta_{\text{int}})$ or $\lambda(\delta_{\text{con}})$. This can however easily be modeled by the introduction of two supplemental output states for the timeout transitions as depicted in Fig. 3.6.



$\lambda((\delta_{\text{int}})) \longrightarrow$ in2
$s'$ 0
$s_{int}$ $\tau(s_{int})$
start
$s_0$ $\tau(s_0)$
$\lambda((\delta_{\text{con}})) \longrightarrow$ in2
$s''$ 0
$s_{con}$ $\tau(s_{con})$

Figure 3.6: Modeling of Transition Dependent Output Function

For the realization of a clocked synchronous model it is necessary to generate an absent event whenever no other event is explicitly generated. For this, a new output function $\Lambda$ is created. The previous output function $\lambda$ is embedded into $\Lambda$ as a partial function. It creates $\diamond$ as long as the timeout is not reached, i.e., $t \neq \tau(s)$. Please note that the definition of $\Lambda$ remains fixed and exists for a clear definition of hardware-near clocked synchronous behavior. It is not possible for a developer to modify this function in the system specification. Therefore, the definition of $\mathbb{S}^{ref}$ will only mention the user-modifiable output function $\lambda$.

### 3.2.4 State Transition

The definition of the three transition functions is consistent with their definition in *Parallel DEVS*. They are denoted as:

$$
\begin{aligned}
&\delta_{\text{int}} : S \rightarrow S \\
&\delta_{\text{ext}} : Q \times X^{ref} \rightarrow S \\
&\delta_{\text{con}} : Q \times X^{ref} \rightarrow S \\
&Q = \{(s,e) | s \in S, 0 \leq e \leq \tau(s)\}
\end{aligned}
$$

Because of the introduction of absent events the definition of the existing transition functions becomes somewhat unclear. Following the existing definitions strictly, these empty events would mean that the internal transition would never fire. Whenever an timeout occurs there will always be an external event. Whether this is an absent event or a real event is not important to the original distinction of confluent and internal transitions. It is very desirable for the comfortable modeling of DEVS model, that the original behavior of the transition functions is preserved. For this, we will first analyze the impact of empty events on the transition functions. In the following, a new meta-transition function $\Delta$ is proposed. This transition function is the only function that needs to be executed and will evaluate which other transition functions have to be executed.

The following transition cases may occur in DEVS models with absent events:

**No Event:** This happens when no timeout has occurred yet ($e < \tau(s)$) and no "real" external event is applied, i.e., all input ports contain the absent event ($\forall (p, v) \in X^{ref}.v = \diamond$).

**Internal Transition:** This happens when the $e = \tau(s)$ timeout is reached and the input ports contain only absent events ($\forall (p, v' \in X^{ref}.v = \diamond$).

**External Transition Variant 1:** The function $\delta_{\text{ext}}$ is called when the timeout is not reached $e < \tau(s)$ and *exactly one* "real" external event is applied, i.e., $\exists ! (p, v) \in X^{ref}.v \neq \diamond$. In this variant the handling of multiple external events will be covered by $\delta_{\text{con}}$.

**External Transition Variant 2:** In this variant it is sufficient if *at least* one external event instead of exactly one event is applied to the input port before the timeout $e < \tau(s)$. The related condition is given by $\exists (p, v) \in X^{ref}.v \neq \diamond$).

**Confluent Transition Variant 1:** The transition $\delta_{\text{con}}$ has to be taken when at least two different events occur. There are two possibilities for this. First, at least one external event ($\exists(p', v') \in X^{ref}.v' \neq \diamond$) happens together with the timeout event $e = \tau(s)$. And second, at least two different external events are applied to the component $\exists(p', v'), (p'', v'') \in X^{ref}.v' \neq \diamond \wedge v'' \neq \diamond$).

**Confluent Transition Variant 2:** : In accordance with the external transition variant 2, $\delta_{\text{con}}$ has to be executed only when both $\delta_{\text{int}}$ and $\delta_{\text{ext}}$ would be jointly called. Therefore $\delta_{\text{con}}$ can only occur when there is an external event in combination with the timeout $e = \tau(s) \wedge \exists(p', v') \in X^{ref}.v' \neq \diamond$.

The different variants for the confluent and external transitions are related to each other. The first variant utilizes the confluent transition for two different external events. The second variant utilizes the confluent transition only for a combination of internal and external transitions, i.e., when a timeout occurs. Two external events before the timeout are left to the external transition function in this variant.

For the *Reference DEVS* and the subsequent development of *RecDEVS* Variant 2 has been chosen. The decision has been taken with respect to potential hardware implementations of the MoC. In these systems the process to calculate the exact number of external events across all input ports is of significant complexity. For the calculation whether at least one event has been occurred a realization of an OR-tree is required. Such a computational structure can be implemented efficiently in hardware.

A combination of all transition functions in one global function $\Delta$ can now be given by:

$$\Delta : Q \times X^{ref} \rightarrow S$$

$$\Delta((s,e),x) = \begin{cases} \text{no Transition} & \text{if } \tau(s) < e \wedge (\forall(p,v) \in x.v = \diamond) \\ \delta_{\text{ext}}((s,e),x) & \text{if } \tau(s) < e \wedge (\exists(p,v) \in x.v \neq \diamond) \\ \delta_{\text{int}}(s) & \text{if } \tau(s) = e \wedge (\forall(p,v) \in x.v = \diamond) \\ \delta_{\text{con}}((s,e),x) & \text{if } \tau(s) = e \wedge (\exists(p,v) \in x.v \neq \diamond) \end{cases} \tag{3.1}$$

Like in previous DEVS definitions, the user specifiable transition functions $\delta_{\text{int}}$, $\delta_{\text{ext}}$, and $\delta_{\text{con}}$ implement a “default”-behavior. If no explicit state transition is given, then these functions remain in their current state. This has no impact on the behavior of the system model and is just to ease the modeling of system components.

**Definition 3.1 (Trace)** *A trace $t(s_1, s_2)$ denotes a possible path from the state $s_1$ to the state $s_2$. It is given by a set of transitions $\{\delta_1, \delta_2, ..., \delta_n\}$. They describe the order of execution to start at the state $s_1$ and reach the state $s_2 = \delta_n(...(\delta_2(\delta_1(s_1))))$.*

There can be multiple and different traces for each combination of beginning and end state. In general, each trace gives only one potential transition sequence of the DEVS model. It is not mandatory that a trace will always reach the final state. A trace might contain an external transition $\delta_{\text{ext}}$ and the required input event for this trace may not occur. It is also possible that another transition may be taken in one intermediate state.

## 3.3 RecDEVS Model of Computation for Reconfigurable Systems

The *Reference DEVS* model already provides many features that have been defined as requirements for the modeling of reconfigurable systems in Chapter 2.

It is specified as a formal model of computation as required by Lemma 1. This enables model-driven design approaches and formal verification as will be shown in Chapter 4. The *Reference DEVS* supports true concurrency and parallelism between the parallel components as required by Lemma 5. With the help of the timeout property $\tau$ all DEVS formalisms fulfill the time annotation requirements from Lemma 6.

In the following the *DEVS Formalism for Reconfigurable Systems RecDEVS* will be presented. It extends the DEVS formalism towards reconfiguration. There have been two important guidelines for the development of *RecDEVS*: Introduce minimal modifications on the existing DEVS formalism and to account for existing reconfigurable hardware architectures and their properties.

A formal model has to provide a formalism that allows the modeling of dynamically changing functionality. To comply to Lemma 7 the formalism should be able to distinguish between

**Dynamic Values** which are utilized during the normal execution of all behavioral models.

**Dynamic Communication** which changes the communication topology.

**Dynamic Behavior** which changes the functionality of the computing core elements that are interconnected by the communication topology and works on dynamic values.

In compliance with Lemma 4 these three aspects should all be part of one combined formal computational model. This will allow formal reasoning about the influence of e.g. the current state (a representative for the class of dynamic values) on certain configuration actions (which lie in the class of dynamic behavior).

These additional requirements have been met by a message-based communication scheme and the introduction of a special arbiter component, which is responsible for all reconfiguration actions.

### 3.3.1 Existing Reconfigurable Models

There are various approaches known, which deal with modeling methods for reconfigurable hardware architectures. Jówiak et al. [29] gives a comprehensive overview over the current state of this research area. Some of these works will be shortly discussed in the following to highlight the different approaches to reconfiguration and the special features which shape up *RecDEVS*.

The first approach to be mentioned addresses the development of dedicated frameworks around existing design languages like VHDL or SystemC. These frameworks add certain reconfiguration mechanisms and reuse existing design flows for conventional hardware development. SyCERS by Santambragio [48], the Perfecto framework by Hsiung et al. [26], and PaDReH by Carvalho et al. [13] are examples for this approach. While these works can benefit from existing design tools, they lack an underlying formal MoC. It is unclear, whether formal design activities such as verification are realizable here.

The Molen processor by Vassiliadis et al. [56] and Morpheus by Thoma et al. [53] utilize reconfigurable architectures as flexible coprocessors. They do not put a specific focus on a formal computational model and are less suited for hardware solutions. A runtime environment utilizing the Molen processor has been developed by Fazlali et al. [19].

A slightly different approach is used in the Hybrid System Architecture Model (HySAM) by Bondalapati and Prasanna [10]. This proposal does provide a formal model for the reconfiguration process. However, it separates the MoCs for modeling reconfiguration and modeling functional behavior. To create a complete reconfigurable system the HySAM model has to be combined with some other model. This will clearly limit a comprehensive analysis of component-induced reconfiguration activities.

In the area of dynamic hardware-aware MoCs the OSSS+R from Schallenberg et al. [49] is to be mentioned. This approach utilizes the concept of code polymorphism from the object-oriented software development domain. We think that this approach is somehow limited for dynamic hardware reconfiguration due to the underlying complexity of the polymorphism paradigm.

As already stated, we strongly believe that a more formal approach will yield better results. In the area of formally defined MoCs the Discrete Event Specified System (DEVS) from Zeigler et al. [58] seems most promising. With the Dynamic Structure Discrete Event Specified System (DSDEVS), Barros [6] has already extended DEVS by a dynamic structure. The work on DSDEVS has been continued by the same author [7] aiming at a heterogeneous flow system specification with the focus on continuous system simulation.

### 3.3.2 DSDEVS

In [6] the DEVS formalism has been extended to the *Dynamic Structure Discrete Event Specified System* (DSDEVS). While not originally targeted to reconfigurable hardware architectures, DSDEVS can be enhanced and thus adopted for hardware reconfiguration purposes. An efficient simulation algorithm for DSDEVS models was presented by Shang and Wainer [50].

DSDEVS introduces a special system component, the *network executive* $C_\chi$, which is part of every system $\mathbb{S}^{DSDEVS}$. $C_\chi$ is realized as a conventional component being enhanced by a structure function $\gamma$.

$$\begin{aligned}\mathbb{S}^{DSDEVS} &= \left\langle X^p, Y^p, elem, conn, C_\chi \right\rangle \\ C_\chi &= (X^p, Y^p, S, s_0, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \gamma, \lambda, \tau) \\ \gamma &: S \rightarrow conn \times elem\end{aligned}$$

The structure function $\gamma$ is executed at each state transition of $C_\chi$. It can modify the system structure which is defined by the port couplings $conn$ and instantiated elements $elem$. $\gamma$ does not directly depend on the input ports $X^p$, but only on the current state of the network executive.

This top-level network executive $C_\chi$ has some special properties and is defined as follows:

$M_\chi = (X_\chi, S_\chi, s_{0,\chi}, Y_\chi, \gamma_\chi, \Sigma^*, \delta_\chi, \lambda_\chi, \tau_\chi)$

$\Sigma = \gamma(s_\chi) = (D, M_i, I_i, Z_i)$ is the actual network topology

$\forall i \in D, M_i$ are conventional DEVS models

$I_i$ is the set of influencers of component $i$

$Z_i$ is the $i-$input function

E.g.: $I_B = (\chi, A) \Rightarrow Z_B : Y_\chi \times Y_A \rightarrow X_B$

In contrast to the classical DEVS specification DSDEVS is originally defined for just one state transition function $\delta$. A distinction between $\delta_{\text{ext}}$ and $\delta_{\text{int}}$ does not take place. Zeigler et al. [58] present a DSDEVS specification with such disjunct state transitions. However, this extended DSDEVS specification is modeled as sequential *Coupled DEVS* only, i.e., without a confluent transition $\delta_{\text{con}}$.

For a specification of reconfigurable hardware systems, which are the focus of this work, the main problem of DSDEVS lies in the specification of $\gamma$. By definition, it models dynamic changes of communication topology and functional behavior of components at the same time. This violates Lemma 7 in which we demand that we should be able to distinguish between these two kinds of dynamic behavior.

In addition, $\gamma$ is defined globally as part of the network executive and depends on its system state. This system state $S_\chi$ itself depends on all actual elements of the structure. Thus, a change in one component of the system may change the state of the network executive and so the structure function $\gamma$ and by this the communication topology $conn$. This behavior clearly violates Lemma 9, by which we demand that interaction between components should only be influenced by a small self-contained interface.

## 3.3.3 RecDEVS Definition

Based on the experiences with DSDEVS the DEVS extension for dynamically reconfigurable hardware systems, denoted as *RecDEVS* is defined in the sequel. *RecDEVS* accounts for the various special properties of reconfigurable hardware architectures. The fundamental concept of *RecDEVS* is the representation of reconfigurable hardware blocks as DEVS components. Thus, some kind of dynamic structure function is being introduced to model reconfiguration within *RecDEVS*.

Similar to DSDEVS, the dedicated network executive $C_\chi$ is responsible for the reconfiguration of the system. It must exist in each *RecDEVS* system at least once. As we will see in Section 3.3.5, in contrast to DSDEVS $C_\chi$ has no special structure function $\gamma$, but can be denoted just like any other *RecDEVS* component.

$$\begin{aligned}\mathbb{S}^{RecDEVS} &= \left\langle X^{par}, Y^{par}, D, \mathrm{C}_\chi \right\rangle. \\ D &= \text{Set of all available DEVS components}\end{aligned}$$

The list of instantiated components $\{\mathrm{C}_d^{par} \mid d \in D\}$ has been moved to the network executive $C_\chi$ and a list of available component names $D$ has been added to the global system description $\mathbb{S}^{RecDEVS}$. This list can be compared to a list of available component types. It is thus possible to add multiple DEVS components of the same type to one system.

*RecDEVS* has moved from a connection-based communication as in DSDEVS towards a message-based communication scheme. Each component instance is defined by its type $d \in D$ and an unique identifier $ID \in \mathbb{N}$. Thus, the set $I = \{C_d^{ID} \mid d \in D, ID \in N\}$ can be used to address all instantiated components within a configuration.

A communication between two components is performed by sending a message onto a global communication system. Each message consists of a a tuple $(sender, receiver, data)$, where $sender, receiver \in I$. The $receiver$ can identify relevant messages by their address. A predefined port coupling with $EIC, EOC,$ and $IC$ is thus no longer necessary.

In the following we present a detailed description of the special properties of *RecDEVS* and the design considerations leading to them.

### 3.3.4 Port Coupling and Communication

The dynamic behavior of reconfigurable hardware systems can be divided into two separate domains:

- Changes in the communication structure.
- Changes in the component instantiation with respect to the component structure. The changed components will then provide a change in functionality.

DSDEVS addresses both problems by means of the dynamic structure function $\gamma$. However, this concept does not work for the targeted reconfigurable hardware.

As stated in Section 2.2, the reconfiguration process of most available hardware architectures is a time-consuming step. Thus, one design goal for the development of *RecDEVS* was the minimization of reconfiguration activities. In many reconfigurable systems such a reconfiguration step does not affect the component interconnection. With Lemma 7, it has been decided to separate reconfiguration of functional units and a structure change of the communication topology. The reconfiguration process of *RecDEVS* only covers dynamic module instantiations and does not include a dynamic communication topology. This allows to realize a change in the communication structure without a time-consuming reconfiguration.

It may even be possible to implement a change of the functional components without influencing the communication structure. This may be interesting for applications similar to garbage collection, where a component is deleted a certain time after it has been completely removed from the communication structure. In this case, it is not necessary to adapt the communication structure upon deletion of the component.

Nevertheless, some kind of dynamic communication topology is required. Otherwise, it would not be possible to communicate with new components. A static communication system solution with a complete set of point-to-point connections may fit this requirement. However, any real-world implementation of this concept will not be feasible due to scalability issues.

*RecDEVS* obtains the required communication flexibility by introducing message-based communication as suggested by Ullmann and Becker [54]. While the underlying communication structure (hereafter referred as bus system) can remain static, the actual communication topology changes with the known and addressable modules within each component. In contrast to the existing DSDEVS MoC and other solutions that would dynamically change the port couplings $EIC, EOC, IC$ this solutions works locally. All topology changes can be realized locally, there is no need for a centralized component which would require a substantial administration overhead. This follows the requirements of Lemma 9.

If multiple components send messages concurrently, all messages may be on the bus system at the same time. It is up to an actual implementation of *RecDEVS* to limit the number of concurrent messages. While this requirement may be complicated to implement, it is a necessity to stay close to the previous port coupling concept. With port couplings, it is possible to utilize multiple different communication channels (each of the described by one port coupling) simultaneously. This feature should be preserved with *RecDEVS* that provides only one bus system. Thus, it requires multiple messages on the bus to obtain the same functionality.

The change from connection-based to message-based communication has an impact on the definition of atomic *RecDEVS* components as follows:

$$\begin{aligned}
C^{RecDEVS} &= \left\langle X^\sharp, Y^\sharp, S, s_0, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \tau \right\rangle . \\
X &: I \times I \times Data \\
Y &: I \times I \times Data \\
\delta_{\text{int}} &: S \rightarrow S \\
\delta_{\text{ext}} &: Q \times X^\sharp \rightarrow S \\
\delta_{\text{con}} &: Q \times X^\sharp \rightarrow S \\
\lambda &: S \rightarrow Y^\sharp \\
\tau &: S \rightarrow \mathbb{R}^+
\end{aligned}$$

Instead of a fixed tuple of input and output ports only one input bag $X^\sharp$ and one output bag $Y^\sharp$ exist.

These bags may contain multiple messages at one time. The transition functions will have to identify relevant messages by their destination address. This also allows for message broadcasting, which is not feasible within port-coupled DEVS realizations. Again, the output function $\lambda$ is embedded into $\Lambda$ as presented for $\mathbb{S}^{ref}$ in Chapter 3.2.3. And again, the system specification will only include the user-modifiable part $\lambda$.

The *RecDEVS* communication concept is a generalized version of hardware bus structures. In actual hardware implementations, the demand for multiple active bus messages at the same time can be hard to realize. However, as we will see in Chapter 4, it is possible to create appropriate hardware implementations based of a proper bus arbitration mechanism. For software implementations like the SystemC based SC-DEVS simulator, the communication bus can be realized quite naturally with a container data type.

### 3.3.5 Reconfiguration

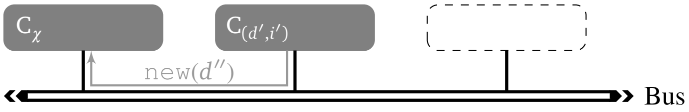
*RecDEVS* handles reconfiguration by a dedicated network executive component $C_\chi$. There are two ways to define the reconfiguration process inside such a component: either with an explicit function like DSDEVS's $\gamma$, or by the utilization of existing functions. We take the second approach in order to minimize the difference between a conventional DEVS component to the dedicated network executive component.

One objective of *RecDEVS* is to model the system such that the network executive stays transparent for a system developer. A reconfiguration process has to be completely encapsulated in the participating DEVS components. This requirement does not hold for DSDEVS, where reconfiguration depends on the current state of the network executive. Other components cannot trigger a reconfiguration directly, i.e., they cannot modify this state.

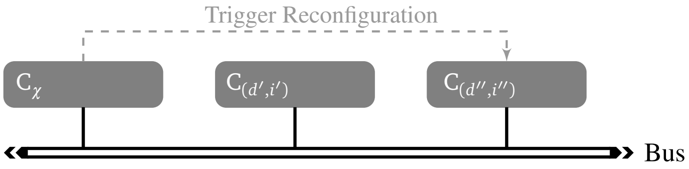
Therefore the reconfiguration task moves from the functional into the communication domain by defining a dedicated set of reconfiguration messages. Reconfiguration is now triggered by sending corresponding messages to the network executive. This requires only a modification of the local output function $\lambda$, a change in $\mathrm{C}_\chi$ is no longer necessary.

This novel reconfiguration scheme models reconfiguration on a higher and thus a more user-friendly abstraction level than DSDEVS. The creation of new *RecDEVS* components consists of a fixed sequence of messages as follows:

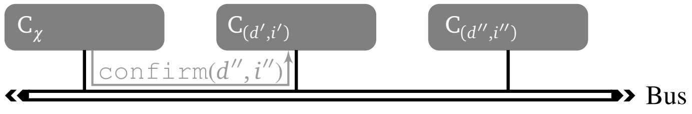
- If the component $C_{d'}^{i'}$ wants to create a new component of the type $d'' \in D$, it sends a message $(C_{d'}^{i'}, \mathrm{C}_\chi, (\text{new } d''))$ to the network executive.



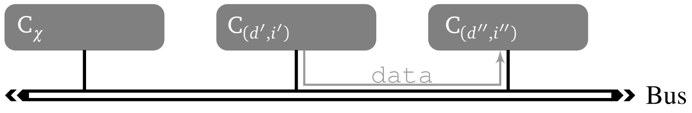
- $C_\chi$ receives the message and performs an external transition $\delta_{\text{ext}}$. This will create a new *RecDEVS* component $C_{d''}^{i''}$ and add it to the list of instantiated components.



- A confirmation message $(\mathrm{C}_\chi, C_{d'}^{i'}, (\text{confirm, } C_{d''}^{i''}))$ with the address of the new component is then sent to the originator.

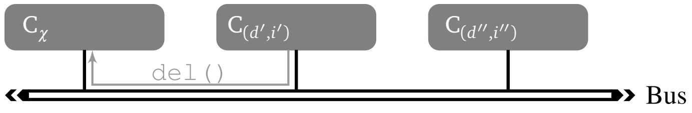


- Upon reception of the confirmation message the originator can address the newly created component.

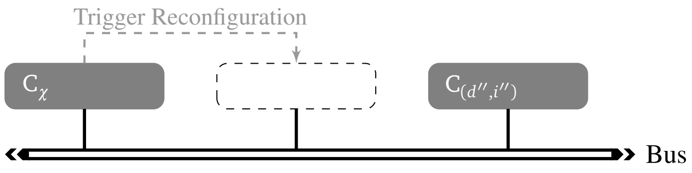


The deletion of components is slightly different. In *RecDEVS* each component can only delete itself, so that it may reach a consistent state before it will cease operation.

- To delete itself, a component $C^{i'}_{d'}$ sends the message $(C^{i'}_{d'}, \mathrm{C}_\chi, (\mathrm{del}))$ to the network executive.



- $\mathrm{C}_\chi$ receives the message and releases the resources of $C^{i'}_{d'}$. However, it is not mandatory to delete this component immediately. The deletion may take place any time after the network executive received the corresponding message.



It is up to the system designer to assure that a component will not react to incoming messages after emitting a deletion message.

## 3.3.6 Definition of the System Executive $C_\chi$

The network executive $C_\chi$ is defined as follows:

$$
\begin{aligned}
&C_\chi = \langle X_d, Y_d, S, s_0, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \tau \rangle \\
&S = \{\text{“Wait”}, \text{“SendConfirm”}\} \times \{\text{Elems} \in D\} \times I \\
&\delta_{\text{ext}}((s, elems, src), e, msg) = \\
&\qquad \begin{cases} (\text{“SendConfirm”}, elems \cup C_d^{ID}, C_d^{ID}) \text{ if } msg = (C_d^{ID}, C_\chi, (\text{new } d)) \\ (\text{“Wait”}, elems \setminus C_d^{ID}, src) \text{ if } msg = (C_d^{ID}, C_\chi, \text{del}) \end{cases} \\
&s_0 = \text{“Wait”} \\
&\lambda(\text{“SendConfirm”}, e, C_{orig}) = (C_\chi, C_{orig}, (\text{confirm } C_d^{ID})) \\
&\tau(s) = \begin{cases} \infty & \text{if } s = \text{“Wait”} \\ 0 & \text{else} \end{cases} \\
&\delta_{\text{int}}(s, elems, src) = (\text{“Wait”}, elems, 0) \\
&\delta_{\text{con}}((s, e), x) = \delta_{\text{ext}}((s, e), x)
\end{aligned}
$$

In *RecDEVS* the set of instantiated components is now represented by the state of the network executive $S = \{\text{Elems}\}$. This set contains all components that are currently active (i.e., instantiated). Reconfiguration processes can then be modeled by simply applying the state transition functions $\delta$. A dedicated structure function $\gamma$ is not required any more.

$C_\chi$ handles two different external input types, which can be regarded as commands. The input message *new(d)* triggers the creation of a new component of the type $d$. The new component $C_d^{ID}$ will be added to the list of instantiated components {Elems} and the originator of the request $C_{orig}$ will be notified with a *confirm* message. This confirmation is necessary since to this point in time no component other than $C_\chi$ knows the actual address *ID* of the new component. The confirmation message contains this information and, thus, the originator component can now communicate with the new component or, it can distribute the address of the new component allowing other components to communicate with the new component as well.

The second message *del()* triggers the executive to delete the component, which sent the message, from the list of active components. Note that there is no broadcasting mechanism announcing that a component is deleted and this concept only allows for deleting itself, but not other components.

This method has two benefits: First, a deletion specific behavior can be modeled differently for each component. E.g., a communication-critical component may notify other components that it will be deleted in the near future and wait for some acknowledge of other components. In contrast, other uncritical components can simply delete themselves without this overhead.

The second benefit is the fulfillment of our encapsulation requirement given in Lemma 9. The complete deletion process is side-effect free because it is handled by each component itself. It cannot happen that a system designer accidently deletes another component which is still in use elsewhere. Instead he has to tell the component that he does not need it anymore and the component itself is able to implement some deciding logic whether it can safely be deleted or must persist longer.

### 3.3.7 Invalid Communication Messages

By utilizing a message-based communication scheme some problems may arise in *RecDEVS*. The formalism does not prohibit messages with a destination address that does not exist in the current network. In other words, a message $m = (M_A, M_B, \text{data})$ is sent by some component but $M_B \notin$ Elems where Elems has been defined as the set of active components held as part of the system state S inside the system executive $\mathrm{C}_\chi$.

The main reason why such messages may occur is the fact that reconfiguration is controlled by an isolated system executive where normal system modules do not need to take care of reconfiguration for themselves. The advantages for this reconfiguration mechanism (like modular component design, the ability to trigger reconfiguration via simple system messages etc.) have already been discussed. Here we see one possible disadvantage arising from this approach. Assume that one component $M_A$ is able to address $M_B$ by having stored that address as part of its component state. In the mean time the deletion of $M_B$ is triggered and executed. Now $M_A$ is still able to send a message to $M_B$. However since $M_b$ is already deleted there is no recipient for this message and the behavior of $M_A$ might not be as expected.

*RecDEVS* does not exclude such invalid messages. The correct behavior would be to simply ignore and discard those messages. In many system implementations this behavior would be just fine and therefore it was decided to keep the *RecDEVS* definition as simple as possible. If a system implementation requires a more advanced error handling for invalid messages, there are three different approaches possible, which will be mentioned in the following. The existence of different suitable error handling models was another reason not to deal with such messages in the original specification. Depending on the actual implementation all of the following approaches have their advantages and disadvantages and *RecDEVS* should not be limited to one of them only.

### Implementation-Specific Mechanisms

The first countermeasure against invalid target addresses is to handle such errors inside a specific implementation and not by the MoC. This can be done in many different ways. The easiest one might be the utilization of acknowledge messages. Given the example above $M_A$ and $M_B$ can be extended in a way that $M_B$ replies with an acknowledge and $M_A$ waits for that acknowledge message. If the acknowledge message is not received within a certain timeout, a suitable recovery is triggered. The main benefit of this solution is its tight adoption to an actual implementation. E.g., there may only be a very small set of messages that would require an acknowledgement. In this case it can save a substantial amount of resources if only exactly these message are monitored but not the remaining part of the messages. A disadvantage is of course that it requires additional implementation logic and the additional work has to be done for each newly implemented model.

### Enhanced Resource Management

The second solution to prevent invalid messages may be a sufficiently smart reconfiguration resource management unit inside the system executive. Messages with invalid destination addresses can only exist if a component has been deleted while it can still be addressed from somewhere else. If the deletion of components would only be performed when no other component can address the component to be deleted any more, this problematic situation can never happen.

This situation is very similar to the mechanism of *Garbage Collection* in object-oriented programming languages. In that case an object may also be deleted as soon as there exist no references to that object any more, but must not be deleted if someone still holds a reference to that object. There is a wide research field regarding efficient garbage collection algorithms, but it is pretty clear that such algorithms can be adopted for *RecDEVS* and improve the reconfiguration mechanism inside the system executive. However, the introduction of garbage collection into *RecDEVS* will also require an additional computational and resource overhead in order to detect which components can safely be deleted and which components will have to be retained.

### Enhanced Communication Infrastructure

The third countermeasure may be the enrichment of the communication infrastructure with a detection mechanism for invalid messages. Each time a message $(M_B, M_B, \text{data})$ occurs a corresponding error message $(*, M_A, \text{“invalid destination”})$ is generated and sent to the originator of the message. The source of the error message is not important and therefore denoted as a wildcard. The detection mechanism itself depends mainly on the actual implementation of the *RecDEVS* communication structure. Thus, it is hard to describe a suitable implementation for this countermeasure approach. If it is possible to implement the system executive $\mathrm{C}_\chi$ in a way that it receives all messages sent by any component, it can be extended as follows:

$$
\begin{aligned}
&\mathrm{C}_\chi = \langle X_d, Y_d, S, s_0, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \tau \rangle\\
&S = \{\text{“Wait”}, \text{“SendConfirm”}, \text{“SendError”}\} \times \{\text{Elems} \in D\} \times I\\
&\delta_{\text{ext}}((s, elems, src), e, msg) =\\
&\qquad\begin{cases}
(\text{“SendConfirm”}, elems \cup C_d^{ID}, C_d^{ID}) \text{ if } msg = (C_d^{ID}, C_\chi, (\text{new } d))\\
(\text{“SendError”}, elems, C_d^{src}) \text{ if } msg = (src, dest, \text{“data”}) \wedge dest \notin elems\\
(\text{“Wait”}, elems \setminus C_d^{ID}, src) \text{ if } msg = (C_d^{ID}, C_\chi, \text{del})
\end{cases}\\
&s_0 = \text{“Wait”}\\
&\lambda(\text{“SendConfirm”}, e, C_{orig}) = (\mathrm{C}_\chi, C_{orig}, (\text{confirm } C_d^{ID}))\\
&\lambda(\text{“SendError”}, e, C_{orig}) = (\mathrm{C}_\chi, C_{orig}, (\text{invalid destination}))\\
&\tau(s) = \begin{cases} \infty & \text{if } s = \text{“Wait”}\\ 0 & \text{else} \end{cases}\\
&\delta_{\text{int}}(s, elems, src) = (\text{“Wait”}, elems, 0)\\
&\delta_{\text{con}}((s, e), x) = \delta_{\text{ext}}((s, e), x)
\end{aligned}
$$

Here, the system executive will verify for all messages that their destination address is currently valid, i.e., that it is listed in $elems$. While this approach might work in a wide area of *RecDEVS* implementations, the overhead of validating all addresses inside one single component may lead to a performance bottleneck in actual implementations. In addition, this approach possibly does not work in more complex *RecDEVS* communication structures, where it is not feasible to forward all messages to $\mathrm{C}_{\chi}$.

For the sake of simplicity the following examples in this work will use implementation-specific mechanisms and handle invalid addresses inside the actual implementation if necessary.

### 3.3.8 Communication Constraints

The introduction of message-based communication revealed some limitations of *RecDEVS*. In a connection-based system the maximum number of input events is limited for each component. Each connection can only transmit one event at a time and thus the number of input ports forms an upper bound for the number of input events. This upper bound can simply be determined locally within each component.

With the global bus system of *RecDEVS* the situation gets more complex. As each output function $\lambda$ can generate multiple messages at the same point in time an upper bound cannot be determined. Since the target of this work is a system specification for reconfigurable hardware systems, the existence of such an upper bound would be very useful. In actual implementations such a bound may be used to dimension message buffers which realize the communication system.

To preserve this feature of connection-based DEVS systems an additional constraint must be introduced that somehow limits the communication capabilities of the system. This limitation should have minimal impact on the system specification, meaning that it has to be easy to implement on the one side and pose on the other as few limitations to the user as possible.

The straightforward solution is to define a limit for concurrent messages that may occur at the same time. However, this constraint is extremely difficult to find as it is system-wide and not component-specific. A system designer has to have detailed knowledge of all other components in the system at each point in time in order to know about the number of messages they are emitting.

Thus, another communication constraint is introduced in order to obtain an upper bound for the number of messages. Each component may still send multiple messages at the same time. However, at one certain point in time it must not send more than one message to each receiver. To provide a clear definition what this means it is necessary to define so called $\tau = 0$-Traces:

**Definition 3.2 ($\tau = 0$-Traces)** *A trace $t(s_1, s_2)$ is called $\tau = 0$-trace iff it is possible to proceed from state $s_1$ to state $s_2$ without any time advancement. This is the case when the trace consists of internal transitions with $\tau = 0$ or external conditions only, i.e.*
$\forall \delta \in t.(\delta = \delta_{\text{int}}(s') \wedge \tau(s') = 0) \vee (\delta = \delta_{\text{ext}})$ .

The existence of $\tau = 0$-Traces can easily be formally analyzed. With the help of this definition it is possible to coin a local communication constraint that can be formally analyzed. The communication constraint can now be refined to

**Definition 3.3 (Communication Constraints for *RecDEVS*)** *To prevent an infinite number of messages on the communication bus all $\tau = 0$-Traces of a RecDEVS may address all other component instances only once. If the number of component instances is given by $|C|$ the upper bound of messages on the communication bus is then $\sharp \leq |I| * |I|$.*

Note that this limitation does also cover multiple time-less (i.e., $\tau = 0$) transitions. In such a situation not more than one transition is allowed to send a message to a receiver. This constraint is not too restrictive as the system designer can decide to merge multiple messages for the same recipient into one larger message structure. It is also component-specific, meaning that the requirement can be checked for each component independently and it is not influenced by other components.

As a consequence of this constraint, each component can create up to $m$ messages at one time step, where $m$ denotes the number of active components in the system. As each component may sent in parallel, there is an upper bound of $m * m$ messages that can occur on the bus. This constraint still does not allow to determine the upper bound locally inside a component because only the network executive is able to evaluate the number of active components during runtime.

### 3.3.9 Resource Management

Up to this point, the definition of the *RecDEVS* network executive $\mathrm{C}_\chi$ assumed that there are always enough resources available to create a new component on demand. In particular, the equation

$$\begin{aligned}&\delta_{\text{ext}}(((s, elems, src), e), msg) = \\ &\qquad (\text{“SendConfirm”}, elems \cup C_d^{ID}, C_d^{ID}) \text{ if } msg = (C_d^{ID}, C_\chi, (\text{new } d))\end{aligned}$$

does not cover any situation, where this might not be the case.

In any real implementation this assumption cannot hold as this would require an infinite amount of resources in the worst case. For *RecDEVS* this means that the new set $elems \cup C_d^{ID}$ would be too large to fit into the available resources, e.g., an FPGA. Thus, some mechanisms to manage the available resources have to be examined. Such *Resource Management* mechanisms have to provide a feasible and useful way to handle resource shortage.

Resource management has already been discussed in the previous Section 3.3.7. In this case it was proposed to prevent the occurrence of messages with invalid destination addresses by retaining components that are still reachable, even if their deletion was requested. But in most cases a resource management will be necessary when handling valid messages of a special type: *create*-messages for the network executive $\mathrm{C}_\chi$ as only those messages can increase the amount of required resources in a system.

As it was the case with the handling of invalid addresses, there exist various approaches for resource management. Each of those may be suited best for certain implementation scenarios. Thus they are not part of the formal specification of *RecDEVS*, but rather suggestions for an actual implementation.

#### Implementation-Specific Resource Management

The approach easiest to implement is based on an introduction of error messages in case that a *create* request cannot be fulfilled. With such an error response the requesting component will be able to realize some own, implementation specific error recovery mechanism. What this error recovery may look like depends completely on the actual implementations and the decisions of the developer. The best error recovery mechanism may differ from solution to solution. This also means that if the developer makes an implementation error or forgets to implement such an error recovery. then the system might fail permanently.

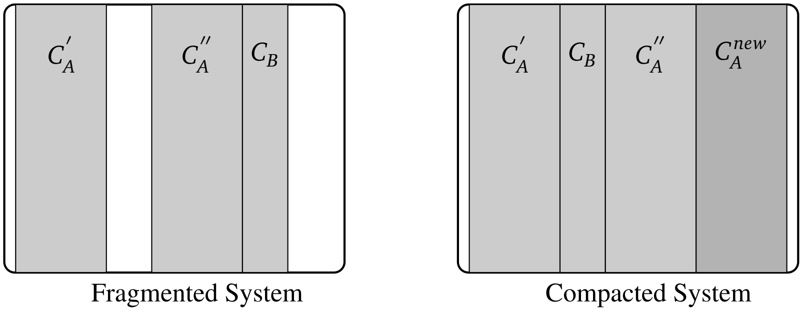


Figure 3.7: Compaction Benefits

## Compaction

By utilizing proper resource management mechanisms it is also possible to optimize the available resources. Depending on the underlying hardware architecture a compaction of the already used resources may result in much more successful *create*-operations than without such compaction. This is the case, if the hardware platform requires individual instantiated components to be locally self-contained. The application of multiple subsequent reconfiguration steps with components of varying size may lead to a fragmented resource allocation, where none of the free fragments is large enough to support the creation of a new component. A compaction step will move these fragments together and thus allows the creation of additional components. This is illustrated in Fig. 3.7, where the system at the left hand side does not allow for the creation of another component $C_A^{new}$, but the situation on the right hand side, which depicts the system after a compaction step, allows to create $C_A^{new}$.

## Component Swapping

Another mechanism to optimize resources usage and thus resulting in less failing *create*-operations is the introduction of component swapping. If there are not sufficient resources for a new component a hardware platform may allow to read out the current system state of an already existing, but currently not running module, and then storing this system state together with the component type on external persistent memory. This component can then be temporarily deleted by the system executive to allow the creation of the new component. The stored component has to be swapped back onto the hardware platform if it is addresses by another component. It is possible to implement a system, where the swapping procedure is completely transparent for the executed *RecDEVS* network. The implementation of such a system demands several features from the *RecDEVS* implementation: The complete system state must be extractable by the network executive and the executive has either to be able to analyze the currently running network to know which components can be swapped out or it has to be able to detect whenever a swapped-out component is addresses, so that it can be restored. In the latter case the additional time required to restore a component has to be taken into consideration when realizing this solution.

### Automated Garbage Collection

The main motivation for an advanced resource management mechanism lies in the analogy of *RecDEVS* software systems, where each reconfigurable component is regarded as an object in the sense of object-oriented software construction. Experience with those systems has shown that a manual resource management by explicitly creating and deleting objects is very error-prone. Developers tend to forget to delete objects that are not longer used and thus do not free the utilized resources. Therefore, automated garbage collection mechanisms have been introduced which analyze the running program and detect objects that are not longer reachable. Those objects can then be deleted automatically by the garbage collector. For *RecDEVS* this means, that it is no longer necessary to perform *delete*-calls manually. Instead, the developer shall only delete the reference to a certain component. If this was the last reference to that module, a garbage collector can delete it. All components that are only known to that component will be automatically deleted as well. Garbage collection requires that the system can distinguish between object references and arbitrary user data, a feature that most object-oriented systems establish by using a type system for variables. To support garbage collection in *RecDEVS*, the implementation must therefore also be able to distinguish between addresses and other data stored as part of the component state. The feasibility of garbage collectors for reconfigurable systems have been examined in the author's diploma thesis in [33], in which a Mark-Sweep-based algorithm has been elaborated and implemented. Its pseudocode representation is given by Alg. 1.

**Algorithm 1** Pseudo Code for Automated Garbage Collection

```
 1: loop
 2:     Suspend forwarding of system messages.
 3:     Enable all components that are denoted in the list of root nodes.
 4:     repeat
 5:         if (A component has been enabled) then
 6:             Set internal Mark flag.
 7:             Send Heartbeat signal to the System Executive $C_\chi$.
 8:             Enable all components listed in the components reference list.
 9:         end if
10:     until No Heartbeat has been received by $C_\chi$ within a certain timeout.
11:     Re-enable forwarding of system messages.
12:     Read all components Mark-flag. All unmarked components may be deleted.
13:     repeat
14:         Normal system execution. No garbage collection is performed.
15:     until A new garbage collection cycle is necessary
16: end loop
```

# 4 Design Flow Methodologies for RecDEVS-specified Systems

In this chapter we demonstrate how the model transformation-based design methodology introduced in Chapter 2 can be applied to *RecDEVS*. Various exemplary model transformation steps that ease system development will be detailed.

In the first part, a horizontal mapping from the *RecDEVS* formalism into UPPAAL will be presented and it will be explained how this can be utilized for the verification of reconfigurable systems. In the second part vertical model transformations will be outlined as they are required for creating hardware implementations of *RecDEVS* models. For this, both a SystemC based model transformation as well as a VHDL implementation of *RecDEVS* are detailed.

## 4.1 Horizontal Transformation for System Verification

First, we demonstrate, how a formal verification technique can be applied to a reconfigurable hardware system specified by means of the *RecDEVS* approach.

While the possibility for verification was one goal of this work, *RecDEVS* itself is not directly targeted towards verification. Instead, it was developed with the general idea of supporting model transformations in mind, as will be described in Section 4.1.1. There are other, more suitable specialized MoCs available for verification purposes. Therefore, a novel mapping method has been developed in order to transform *RecDEVS* models into a timed automata based representation of the UPPAAL Model Checker presented by Larsen et al. [32]. Using this approach the designer can benefit from the model specific features of *RecDEVS* and the expertise of the UPPAAL verification system at the same time.

The *RecDEVS* Model of Computation captures the functionality of reconfigurable hardware systems. It focuses on reconfiguration and provides specific features for the description of such reconfigurable features. On the same abstraction level lies the UPPAAL Model of Computation, which already includes verification expertise and knowledge. A transformation from *RecDEVS* to UPPAAL models is thus highly appropriate to obtain verification results at an early stage of the design process. If the transformation preserves all important model properties, then the results of the UPPAAL verification will also hold for the equivalent *RecDEVS*-based design. These results may then be used to further refine the implementation until all desired verification properties are met.

A model transformation in the other direction, i.e., from the UPPAAL MoC back into a *RecDEVS* specification was not part of this work. While the presented transformation process can map arbitrary *RecDEVS* models to UPPAAL, this is not true the other way around. If the layout of the UPPAAL models is changed, the proposed model transformation step cannot simply be reversed. This may be of importance in cases where the system developer makes some modifications to the UPPAAL model, e.g., after finding a design flaw during the verification process. However, in the various examples that have been developed for this work, this has never been an issue. The correspondence between *RecDEVS* and UPPAAL models always has been so close that it was easily possible to fix the design flaws directly in the original *RecDEVS* specification. This modified specification may then be transformed to UPPAAL and verified again to validate the outcome of the modifications.

As already stated, there are some other approaches for modeling dynamic reconfigurable systems based on lower level programming languages like VHDL, SystemC or ImpulseC by Santambragio [48], Hsiung et al. [26], and Craven and Athanas [17], respectively. One statement of this thesis is that a formally specified model, in contrast to actual programming languages, seems more promising for model transformation in general and verification in special.

Both approaches, HySAM by Bondalapati and Prasanna [10] and *RecDEVS*, provide such a required formal specification foundation. HySAM splits the descriptions of reconfiguration and functionality, respectively, into two disjoint models which makes a conclusive verification difficult. *RecDEVS* combines both descriptions in a single model and thus supports the verification of function-triggered reconfiguration properties, directly.

Regarding the verification of *RecDEVS* models, there is some preliminary work on the formal verification of the underlying DEVS formalism. The first work by Morihama et al. [41] implements an own theorem prover and has been extended towards the verification of continuous systems by Saadawi and Wainer [47]. Other approaches by Weingart [57] or Dacharry and Giambiasi [18] benefit to some extend from the established UPPAAL model checking environment.

### 4.1.1 System Verification

Formal verification techniques as presented by Gupta [24] are becoming increasingly important nowadays. They are used to establish a relationship between an implementation and a formal specification. By creating a formal specification of the desired system behavior it is then possible to validate that the system implementation is functionally correct. Verification techniques are based on mathematical proof methods and are one way to handle the increasing complexity of system specifications. In contrast to testing techniques formal verification techniques are less error-prone to user induced errors, such as forgetting an important test case. In the borders of the formal system specification they can provide an exhaustive and complete conclusion on system correctness.

There are two general approaches for formal verification that are commonly used: *Logical Inference (theorem proving)* and *Model Checking*.

*Logical Inference* is based on automated theorem proving environments. Both, the system specification and its properties are expressed with the help of mathematical formulas. Then, the theorem proving tools try to proof the correctness of the system properties based on the system specification, axioms, and inference rules such as induction. Logical inference allows to verify properties on infinite system states and can be very powerful. However, currently the process of theorem proving cannot yet be fully automated. All automated theorem proving environments require manual user interaction for non-trivial problems. If a proof fails there is not always a clear connection to the original specification, so a system-designer is not always able to obtain counter-examples.

The other general approach to formal verification, *Model Checking*, was developed independently by Clarke and Emerson [15] and Queille and Sifakis [44]. The idea is to model a system as a state transition graph and to define specifications in temporal logic. By using very efficient and fully automated algorithms many model properties can then be verified by processing on the state transition graph. Also, Model Checking is able to generate a counter-example in form of one way through the transition graph that violates the demanded specification. Compared to theorem proving approaches Model Checking is less powerful as it does not use higher order logic such as structural induction. In general, Model Checking is also limited to finite state spaces, however, a lot of enhancements have been applied to extend its capabilities towards infinite state sets. Grumberg and Veith [23] give an overview on the current state of model checking.

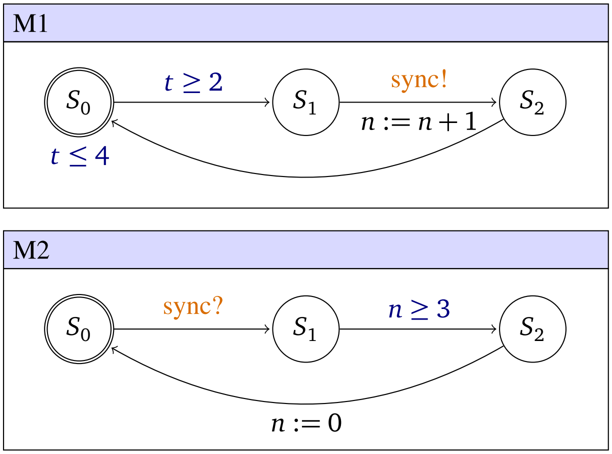


Figure 4.1: UPPAAL Automata Example

For the verification of *RecDEVS* the Model Checking approach was chosen for several reasons. Both MoCs are based on state transition graphs, enabling an easy way to apply model checking techniques to *RecDEVS*. The generation of counter-examples and the fully automated system verification process are other properties, which were of essential importance to the author.

### 4.1.2 The UPPAAL Model Checker

UPPAAL, presented by Larsen et al. [32], is a model checking tool for modeling, simulation, and verification of real-time systems. It is based on constraint-solving and explicit verification techniques. The model checker is suitable for the verification of systems, which can be represented by nondeterministic processes with finite control structures and real-valued clocks, i.e., Timed Automata.

UPPAAL offers the verification of arbitrary user defined specification requirements such as reachability, safety, or bounded liveness properties. Its intrinsic requirement specification language exploits timed computational tree logic. The UPPAAL Model Checker has been successfully used in for many industrial case studies (e.g. Behrmann et al. [8]). Its specification language is a finite-state machine extended by clocks, synchronization channels, state and transition invariants, data variables, and update labels. Time is modeled by means of a set of multiple user-defined clocks. These clock values are incremented continuously, but they can also be set interactively to arbitrary values during the model execution.

Fig. 4.1 illustrates the essential elements of an UPPAAL system with two communicating automatons M1 and M2, respectively. Exactly one state of each automaton is marked by double lines as the initial node. Every state may additionally be labeled with a state invariant (e.g., the invariant $t \leq 4$ for the state $\texttt{M1.S}_0$) to express time constraints. The system may stay in a state as long as the invariant is true. The state has to be left over state transitions at the latest when the invariant value changes to false. If no invariant is given, then the invariant is true by default.

All transitions may be attributed by a guarding condition that has to be true for the execution of the corresponding transition. The communication between the automatons M1 and M2 in Fig. 4.1 is realized with synchronization channels and shared data structures. Whenever a transition is marked with an 'emit' synchronization (denoted by an exclamation mark) a corresponding 'receive' synchronization channel (denoted by a question mark) has to exist and its transition will be executed, too. While synchronization channels contain no additional data, they can be complemented with update labels. These labels are executed on a transition and enable the user to update shared data variables or to modify clock values. In the synchronized receiving transition these variables can then be read with another update label and thus realize the data exchange. In the example of Fig. 4.1 this is demonstrated by means of the shared variable $n$.

### 4.1.3 Model Transformation from *RecDEVS* to UPPAAL

Both models, *RecDEVS* and UPPAAL, have a similar structure and execution model. They utilize an event-based, explicit specification of timed behavior and are based on a concurrent, state-transition based execution model. As Molter et al. [39] explain, this similarity is necessary to allow for an automated transformation process between both models.

The main requirement for all created transformation rules is that they preserve the behavior of the originating model. Verification can only prove properties of the original *RecDEVS* model if the transformation can guarantee the equivalence of both models. However, even without formal equivalence, verification environments can still serve as a counterexample-based test system.

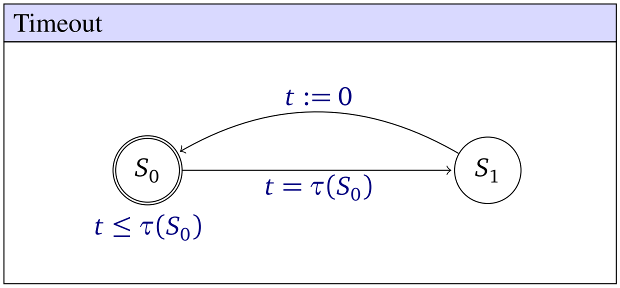


Figure 4.2: Timeout Realization in UPPAAL

It is possible to perform an automated transformation of *RecDEVS* models into UPPAAL ones. First, we present a set of transformation rules for all basic elements of a *RecDEVS* model. As described in Molter et al. [39], this allows the implementation of conversion tools that can automatically transform any user defined *RecDEVS* model. Then, a pseudo-code representation for the conversion of a complete model is given in Alg. 2.

Secondly, we summarize features of *RecDEVS*, which can not be translated properly and we discuss the related consequences. Whenever the preservation of all properties is not feasible, the verification bandwidth will be somewhat limited. These limitations stem from the differences between two distinct models of computation and are unavoidable. The transformation process tries to circumvent such limitations whenever possible.

Both models, *RecDEVS* and UPPAAL, incorporate multiple, communicating components. It is thus feasible to transform each component of a *RecDEVS* model into a corresponding UPPAAL automaton. However, the UPPAAL model does not provide mechanisms for a dynamically changing set of the components as required by the *RecDEVS* formalism. Section 4.1.3 describes how such a behavior can still be represented in UPPAAL.

### Timing Behavior

Every *RecDEVS* state has an associated timeout function $\tau : S \rightarrow \mathbb{R}$. However, timeouts are not directly supported in UPPAAL. Thus, the timeout for an exemplary state $S_0$ is realized by a combination of a state invariant $t \leq \tau(S_0)$ for $S_0$ and a

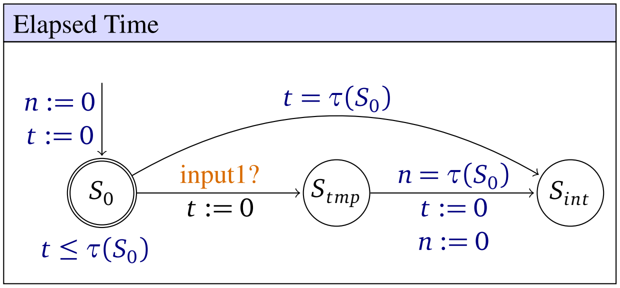


Figure 4.3: Elapsed Times over Multiple States

transition guard $t = \tau(S_0)$. Fig. 4.2 shows a corresponding UPPAAL model with a timeout $\tau(S_0)$ on state $S_0$. The invariant forces the system to leave the state at the latest when $\tau(S_0)$ time units have passed on the clock $t$ and the guard prevents the system to take the transition any time before $\tau(S_0)$. So, the transition with the guard expression has to be taken exactly at the desired timeout time value.

A minor limitation of the model transformation is that UPPAAL supports natural numbers only and hence can only realize somewhat restricted timeout functions $\tau : S \rightarrow \mathbb{N}$ instead of $\tau : S \rightarrow \mathbb{R}^+$. For a correct implementation of the timeout it is also necessary to reset the clock $t$ to zero whenever a state $S_0$ is entered. This has to be done on all incoming transitions using update labels.

UPPAAL does not provide any mechanism to obtain the elapsed time, when an synchronization channel is triggered. This means that it is not possible to obtain the elapsed time $e$, which is required by the *RecDEVS* transition functions $\delta_{\text{ext}}((s, e), x)$ and $\delta_{\text{con}}((s, e), x)$. However, there is a wide range of applications where the elapsed time is either not required, or it is only used to preserve the timeout of the originating state. The latter scenario happens when a short interruption of a longer timeout cycle is triggered. After the interrupt it is likely that the original timeout should continue without restart. This is possible by introducing a second clock which is not automatically reset to 0 on each transition as illustrated by means of the additional clock $n$ in Fig. 4.3. Currently, there is not yet an algorithm implemented to detect the mentioned short interruption of a longer timeout state automatically. Thus, the additional clocks for such interrupts have to be inserted manually into a generated UPPAAL model.

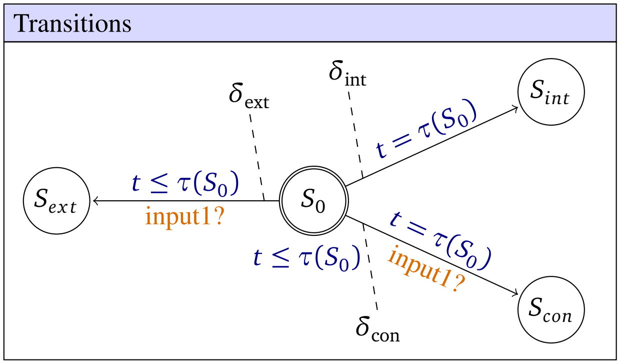


Figure 4.4: Mapped DEVS Transitions

### RecDEVS Transitions $\delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}$

All three *RecDEVS* transitions are realized by distinct UPPAAL transitions. Using the previously described timeout mechanism these three transitions mainly differ in their guard conditions and synchronization channels. The resulting model of a exemplary single state $S_0$ with three leaving transitions towards the states $S_{int}, S_{ext}$, and $S_{con}$ is depicted in Fig. 4.4.

The internal transition $\delta_{\text{int}} : S_0 \rightarrow S_{int}$ is guarded by the timeout condition $t = \tau(S_0)$. The external transition $\delta_{\text{ext}} : S_0 \times X \times \mathbb{R} \rightarrow S_{ext}$ is guarded by a receiving synchronization channel and must not have reached the timeout point in time, i.e., $t \leq \tau(S_0)$. The synchronization channel represents the external event of an *RecDEVS* model inside UPPAAL. The confluent transition $\delta_{\text{con}} : S_0 \times X \times \mathbb{R} \rightarrow S_{con}$ combines the timeout of $t = \tau(S_0)$ of internal transitions and the synchronization channel mechanism of external transitions.

For the timeout $t = \tau(S)$ both transitions, $\delta_{\text{int}}$ and $\delta_{\text{con}}$, may trigger. However, UPPAAL will always prefer transitions with synchronization channels. This behavior is similar to a *RecDEVS* model, where the confluent transition has to be taken and thus no further conditions are required to assure that the correct transition will be executed.

### Inter-Module Communication

While *RecDEVS* utilizes a message based communication scheme, UPPAAL features dedicated communication channels. It is therefore necessary to introduce a synchronization channel for each output message $\lambda : S \rightarrow ID \times \text{Data}$ of a *RecDEVS* model. All synchronization channels must have unique names, which can be guaranteed by the target identifier *ID* that uniquely defines the recipient of the message within *RecDEVS*. Thus, for each message a corresponding synchronization channel pair *ID_Data!* and *ID_Data?* is created in UPPAAL.

*RecDEVS* allows the occurrence of multiple events at the same time. In UPPAAL synchronization channels can only fire sequentially, which eventually leads to an execution mismatch between both models.

To minimize this difference the transformation takes advantage of the fact that UPPAAL chooses indeterministically between possible transitions. To represent two concurrent messages the equivalent UPPAAL model implements both possible synchronization message orders as illustrated in Fig. 4.5. This approach can be extended to any number of multiple output events.

### Reconfiguration

As already stated reconfiguration in RecDEVS is performed by a set of dedicated communication messages. Consequently, these messages are to be mapped into UPPAAL models by means of synchronization channels as described in Section 4.1.3.

However, a problem arises from the static structure of UPPAAL, which does not allow for the creation of new modules. Thus, for the reconfiguration of UPPAAL models a new state is introduced for each model, to denote the 'deleted' property. Then, a set of 'deleted' modules is instantiated. The creation of a new module changes the system state of a free module from 'deleted' to the initial state of the DEVS model for this module. Consequently, a deletion of an instantiated module is performed by resetting the state values to 'deleted'.

Depending on the implemented design it may be necessary to introduce an equivalent to the system executive $C_\chi$ in UPPAAL. This component has to perform the arbitration of available unused components and to distribute the reconfiguration messages. It does also suppress the *confirm()* message when no 'deleted' components are available to fulfill a *new()* request. For implementations with a predefined order of reconfiguration the activities of the network executive can simply be removed.

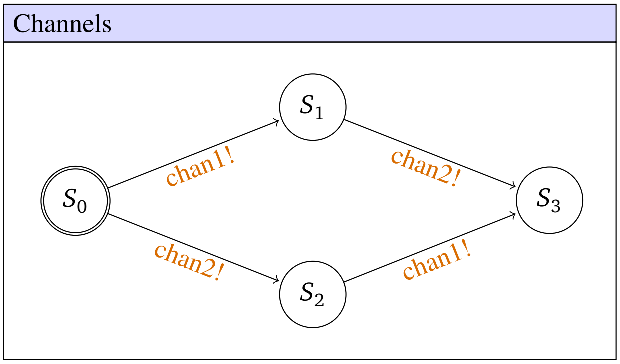


Figure 4.5: Concurrent Channels

While this solution may be viewed as a limitation in comparison with the original *RecDEVS* model, it resembles other reconfigurable hardware architectures with limited communication resources. Such a method is also used in other approaches to model reconfigurable systems in other description languages such as related work on SystemC by Hsiung et al. [26].

### 4.1.4 Automatic Transformation

Algorithm 2 gives a pseudo code representation of the outlined transformation rules. Please note that the following representation of the mapping method is generic in so far, because it creates just one reconfigurable module for each UPPAAL model. For the instantiation of multiple components the algorithm has to be extended appropriately. In that case, the names of the synchronization channels have to be adopted for uniqueness as well.

**Algorithm 2** *RecDEVS to UPPAAL Transformation*

```
Input: RecDEVS Specification $\mathbb{S}^{RecDEVS} = \left\langle \mathrm{X_{ext}}, \mathrm{Y_{ext}}, D, \mathrm{C}_{\chi} \right\rangle$.
Output: A corresponding UPPAAL system representation
function Transform($\mathbb{S}^{RecDEVS}$) is
   Create Global Time Variable $t$
   for all $d \in D$ do
      Create an UPPAAL Component $d$
      for all $s \in S_d$ do
         Create an UPPAAL State $s$
         Create Transition from $s$ to $deleted$ with Synchronization Channel "$del()$?"
         Create State Invariant $t \leq \tau(s)$
         for $\delta_{\mathrm{int}}(s) = s_{int}$ do
            Create Transition $t$ from $s$ to $s_{int}$
            Update(s,"$x = \tau(s)$","$x := 0$",$\emptyset$,t)
         end for
         for $\delta_{\mathrm{ext}}(s, x_{in}, e) = s_{ext}$ do
            Create Transition $t$ from $s$ to $s_{ext}$
            Update(s,"$x \leq \tau(s)$","$x := 0$","$d\_input$?",t)
         end for
         for $\delta_{\mathrm{con}}(s, x_{in}, e) = s_{con}$ do
            Create Transition $t$ from $s$ to $s_{con}$
            Update(s,"$x = \tau(s)$","$x := 0$","$d\_input$?",t)
         end for
      end for
   end for
end function
function Update($s, g, l, i, t$) is
   Add Guard Condition $g$ to $t$
   Add Update Label $l$ to $t$
   Add Synchronization Channel $i$ to $t$
   if If $\lambda(s) = (tar, msg)$ is present then
      Add Synchronization Channel "$tar\_msg$!" to $t$
   end if
end function
```

## 4.2 Vertical Design Flow Methodology for System Implementation in SystemC

To establish a vertical design flow for *RecDEVS* based systems a two-step approach has been chosen. In a first step a vertical transformation step for conventional DEVS models has been established. Only in a subsequent second step this transformation has been extended and modified to reflect the differences and extensions between DEVS and *RecDEVS*.

In the following the first part will be detailed, i.e., how the DEVS Model of Computation can be taken to SystemC. This work has been done together with G. Molter and has been previously published in Molter et al. [39]. We distinguish between two fundamental concepts: Mapping of the whole MoC in a generic way and specialized mapping of a single model from the MoC.

The first approach, the mapping of the whole MoC, can operate on arbitrary models. It reflects the MoC specific computational rules in a generic way. Once the MoC formalism is implemented manually to SystemC code, we can derive every single MoC model from it. The model derivation can be done automatically.

The latter approach, the mapping of a single model, may result in a more fine-grained implementation. As the mapping process is mostly handcrafted, we can cope with model specific characteristics and thus optimize the code. Modeling expertise about the concrete realization of the model is then used to create a sufficiently simple SystemC description. The transformation into SystemC cannot be done automatically. Each time the model is modified, its changes must be transformed into SystemC, too.

### 4.2.1 SC-DEVS Extension for SystemC

Therefore we choose the former approach and integrate the whole MoC into an extended SystemC version. Our implementation can execute arbitrary *RecDEVS* models. The implementation is realized as non-introspective extension to the SystemC 2.2 kernel. Thus, the existing SystemC kernel is not modified, it has just to be extended in an appropriate way. Fig. 4.6 depicts the relationship of the classes from the formal model and their SystemC counterpart.

In the following, a short impression of the SystemC kernel extension with DEVS functionality is given. A more detailed description of this extension is presented in Madlener et al. [35].

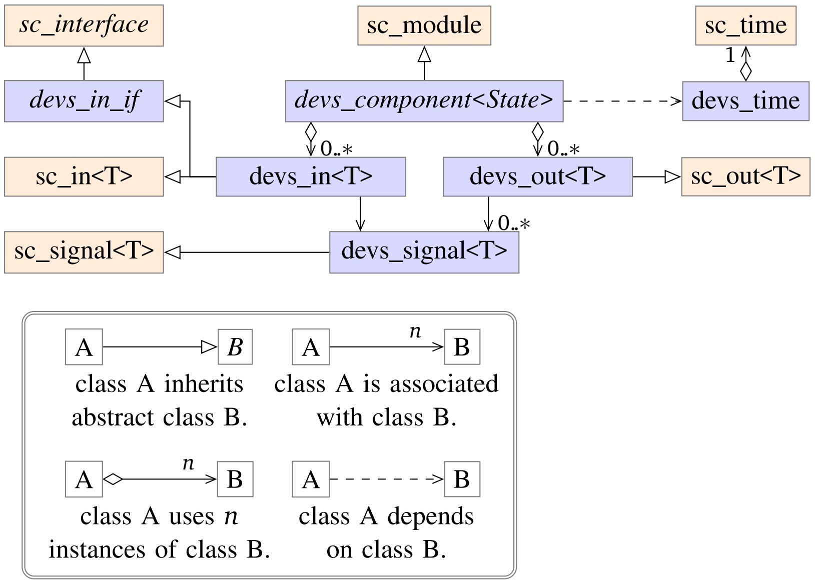


Figure 4.6: UML Class Diagram of an exemplary MoC Mapping to SystemC

The DEVS communication ports differ from the SystemC ones. Like the communication in other event-based MoCs, e.g., TLM [1], *RecDEVS* ports emit events even if the same value is written twice to an output port. Therefore, we enhanced the normal sc_in and sc_out port together with the sc_signal class to include this behavior. See devs_in, devs_out, and devs_signal relations in Fig. 4.6.

In DEVS all specified models run in parallel, like components of other concurrent MoCs, e.g., parallel State Charts execution. This behavior is provided by the sc_module class as SystemC runs all modules concurrently anyway. To integrate the whole MoC into SystemC, the devs_component<State> class has an additional private `advance` function, which maps the complete DEVS behavior to SystemC. It is activated upon the receipt of an external event through the input ports or by the current state timeout event. Iff the `advance` function was activated due to a timeout event, then the output ports emit events. The output function $\lambda(s)$ is directly mapped to the corresponding abstract `output` function of devs_component<State>. The `advance` function computes the next state with the help of the $\delta_{\text{int}}$, $\delta_{\text{ext}}$, or $\delta_{\text{con}}$ functions, respectively.

As SystemC, or rather C itself, does not enforce side-effect free functions, three different implementation variants of the `advance` function have been realized.

Needed Serial: Only the needed transition function is executed. To determine which transition function is needed it has to be checked whether a component was activated by timeout or by external events. This variant is supposed to give the best performance. However, if a certain side-effect of one unneeded transition function is required, this variant might not work.

All Serial: The three transition functions, $\delta_{\text{int}}$, $\delta_{\text{ext}}$, and $\delta_{\text{con}}$, are all executed in a serial, but randomized order. But only the state of the actually required function is kept, the follow-up states of the two other two transitions are discarded. This variant might expose certain unwanted side-effects that have accidentally been modeled in a system specification. It can be especially useful during development and testing.

Parallel: All three transition functions are executed in parallel with the help of `pthreads`. Again, only the follow-up state of the active transition is kept.

Like in any time-based MoC the state transitions appear irregularly, because every state may have an own timeout value. This requires a clock independent global time modeled by the new `devs_time` class. When the timeout was hit, then a state transition occurs. The timeout function $\tau(s)$ is described by the devs_component<State> abstract `timeout` function. After the `advance` function, the component is suspended until reactivation.

In order to transform a *RecDEVS* MoC model into a SystemC representation automatically, the behavior-specifying functions $\tau$, $\lambda$, $\delta_{\text{int}}$, $\delta_{\text{ext}}$, $\delta_{\text{con}}$ have to be denoted in SystemC. These functions are instantiations of the devs_component<State> abstract functions as described above. Besides the declaration of the input and the output ports, they provide the `advance` function with all the necessary information to construct an executable SystemC module from an abstract DEVS model.

### 4.2.2 Extending SC-DEVS towards *RecDEVS*

To illustrate the feasibility of the *RecDEVS* MoC for dynamic reconfiguration the SC-DEVS engine has been further extended towards reconfiguration. Due to its modular concept it can comfortably be enhanced to cover the additional elements of *RecDEVS*. The extension consists of three major modifications:

1. Implementation of a network executive
2. Integration of the message-based communication scheme
3. Extension of the existing DEVS components for dynamic reconfiguration

As Hsiung et al. [26] showed in their Perfecto framework, SystemC based reconfiguration environments have the common disadvantage that this modeling language does not support a dynamic instantiation of modules. Similar to Perfecto, the *RecDEVS* implementation realizes reconfiguration on top of SystemC with a static set of instantiated components. Each component contains a flag for its configuration state. During reconfiguration the network executive sets these flags to 'active' on creation and to 'inactive' on deletion of a *RecDEVS* component.

For the support of message-based communication a *RecDEVS* bus system was realized by using the SystemC communication channel *sc_channel*. SC-DEVS allows the seamless integration of arbitrary *sc_channel* implementations. Listing 4.1 depicts how `write()` and `read()` operations are realized. The messages are kept in the std::list container `bus_data`.

```
std::list<msg> bus_data
messages read(const Model match) {
  messages::iterator iter, endfor;
  if (!(bus_data.empty())) {
    endfor = bus_data.end();
    for (iter = bus_data.begin();
         iter != endfor;
         iter++) {
      if ( match == (*iter).dst) {
        rvalue.push_back(*iter);
        bus_data.erase(iter);
      }
    }
  }
  return rvalue;
}
```

```
void write ( std::list<msg> u ) {
  if ( timestamp!=sc_time_stamp()
       && !(bus_data.empty()) )
    bus_data.clear();
    timestamp = sc_time_stamp();
    bus_data.insert(bus_data.begin(),
                    u.begin(),
                    u.end() );
    bus_event.notify(SC_ZERO_TIME);
}
```

Listing 4.1: Read and Write Operations

Please note that the flexibility of message based communication comes at a certain cost. For each read operation the `bus_data` list has to be traversed to check the receiver of each message. If this list becomes larger this may affect the overall performance.

## 4.3 Vertical Design Flow Methodology for a VHDL-based Implementation of RecDEVS

As an alternative vertical implementation approach a direct implementation of *RecDEVS* in the VHDL programming language has been chosen.

The idea behind this approach was to obtain the most benefit from existing tools for reconfigurable hardware platforms. VHDL is one of the most established hardware description languages. It is widely used in FPGA-based system implementations which represent the most popular platform for reconfigurable hardware.

The implementation has been realized as part of the master thesis by Theisen [52], supervised by the author of this work. The proposed solution offers a framework of skeleton VHDL components. Each of this VHDL entities represent one reconfigurable *RecDEVS* module. The complex communication scheme of *RecDEVS* and the network executive $C_\chi$ are both implemented as part of this skeleton.

So, a system developer only has to implement the VHDL equivalent of transition functions $\delta_{\text{int}}$, $\delta_{\text{ext}}$, $\delta_{\text{con}}$, the timeout $\tau$, and the output $\lambda$. For this functions a fixed set interfaces is defined. In the following the concept and architecture of the framework are described.

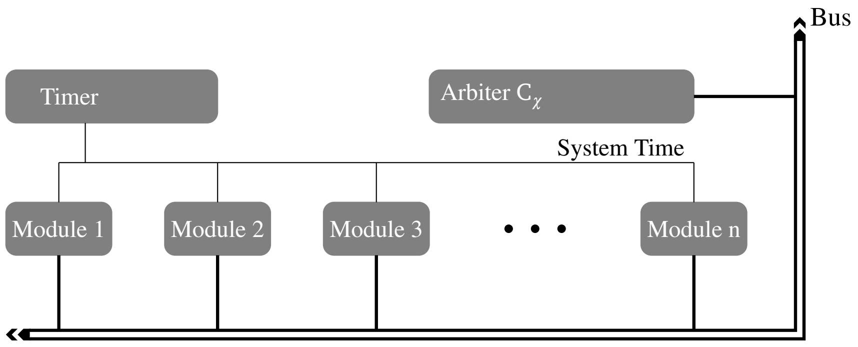


Figure 4.7: Architecture of a VHDL-based RecDEVS network

### 4.3.1 System Architecture

The general architecture of the VHDL implementation for *RecDEVS* is given in Fig. 4.7. Beside the already mentioned *Arbiter* that implements $C_\chi$ and the reconfigurable modules, the framework implements an additional timer element. All reconfigurable components and the arbiter are connected to one centralized communication bus. Generally, FPGAs and other hardware platforms can only offer limited communication resources. It is not possible to send multiple messages on the same bus at the same clock cycle. Thus, the arbiter does also handle bus arbitration in a way that only one component sends at a time.

The *Timer* component generates a global time as it is required by the timed computational model of *RecDEVS*. It is not possible to directly use the clock for *RecDEVS* timing. The main reason for this is that a *RecDEVS* time event will take multiple hardware clock cycles to process (e.g., for arbitration of the communication system). The *Timer* implementation also allows to interrupt and pause the system execution, e.g., if a hardware platform does not support dynamic reconfiguration.

### 4.3.2 Bus System and Protocol

Each message on the bus consists of the address of the originator of this message, the destination of the message, and the message body itself. In this framework both addresses are realized as 16 bit wide elements, the data bus is 8 bit wide. In addi-

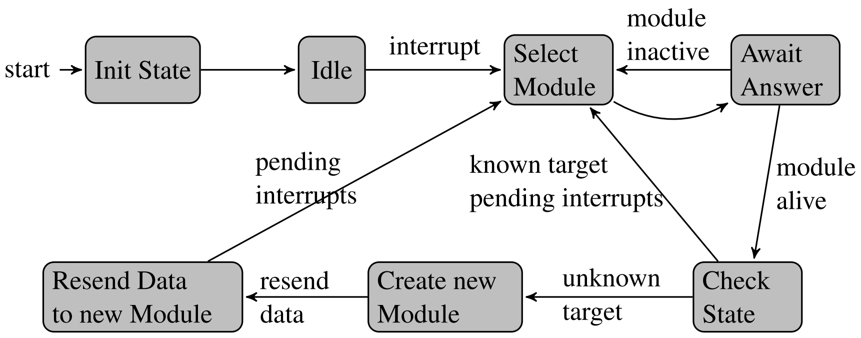


Figure 4.8: State Machine of the Arbiter

tion, there is one global 1 bit wide interrupt line. Each reconfigurable module which wants to write to the message bus (as required by the output function $\lambda$) will pull the interrupt line up to 1.

The arbiter will then hold the *RecDEVS* time and address all active components in a round-robin process. Each component that is addressed in this way may now write its data to the bus. If a component does not start to send data, the arbiter will simply go to the next component in the list.

As soon as the interrupt line is no longer pulled to 1 there are no more components trying to send a message to the bus. In this case the arbiter will release the timer and *RecDEVS* computation time will go on. The state machine of this communication scheme as implemented by the arbiter is denoted in Fig. 4.8.

### 4.3.3 Arbiter Implementation

Beside processing the bus system communication protocol the arbiter is responsible for managing the available resources. This means that an arbiter has to know about free resources such that it can process *new* requests by existing components. It does also hold a list of active components so that it is able to address them in round-robin manner for bus arbitration. A substantial implementation problem is the effective handling of these lists in hardware. Since the removal and creation of random components leads to a fragmentation of the resource lists, an approriate VHDL implementation has to be found. A simple, but somehow inefficient approach would be to compact the list after each processing step. Another more complex way that was chosen for this implementation is with list indirections as depicted in Fig. 4.9.

The first list enumerates all physical positions (i.e. available resources) of the FPGA. If such a resource is currently active then the list entry contains the *RecDEVS* address of that component. This list may be fragmented since not all resources are always used, the list entry of unused resources are invalid.

Each element of the second list contains a pointer to one unused component in the first list. This list is realized as a stack, so when a new *RecDEVS* component needs to be created the arbiter takes the physical resources found from the pointer of the top element of this stack. If a component is deleted then a pointer to this position is put on top of the stack. Part of this second list is an additional stack pointer marking the top element of the stack.

### 4.3.4 Implementation Results

The depicted VHDL architecture has been implemented for the *Xilinx Virtex 5* architecture. On that platform the implementation of the framework components, i.e., the timer and the arbiter, takes about 800 logic cells, which is about 1% of the available resources on that platform. All remaining resources can be used for reconfigurable components. The arbiter requires about 1000 FlipFlops which would be about 50% of the available resources. Please note that no dedicated memory elements, such as BRAM, have been used.

Without any specific focus on optimization, the design can reach 160 MHz maximum clock frequency. In Chapter 5 the implementation of *Game of Life* on this architecture will be presented. It gives an impression of the performance of this architecture and of the logic requirements for reconfigurable cells.

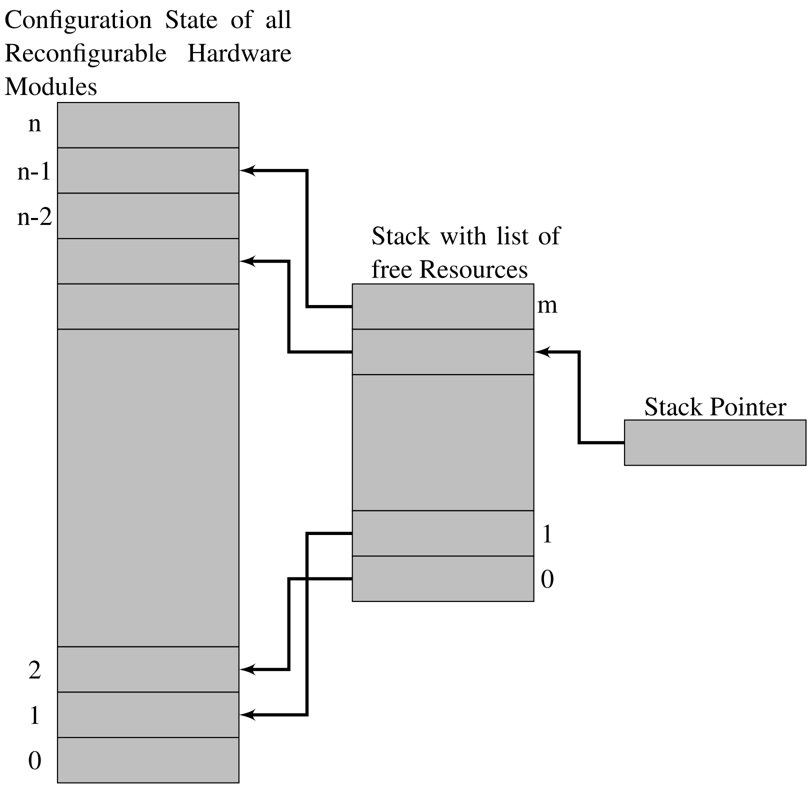

Configuration State of all Reconfigurable Hardware Modules
n
n-1
n-2
2
1
0
Stack with list of free Resources
m
1
0
Stack Pointer

Figure 4.9: Lists for Resource Management inside the Arbiter Component

# 5 Demonstration of Concepts

In this chapter the practical application of *RecDEVS* shall be demonstrated on more complex examples.

For this reason two different examples haven been selected that focus on different aspects of *RecDEVS*. The first example is a more coarse-granular modeling of interacting components in an automotive system. The system is based on external input events and reconfigures functional components depending on these input events. As these events originate from an uncontrollable output environment, the confluent transition often comes into play to address event handling on internal transitions. However, due to the coarser granularity, the number of variations regarding reconfiguration are not too large.

This issue is addressed in the second example, the “Game of Life”. This game represents a cellular automaton with a simple set of rules, where the cells are reconfigured throughout the execution. Compared to the first example, the input events are much more predictable as the system is self-contained. However, the large number of cells leads to a highly dynamical reconfigurable system.

In addition, the span between these two examples illustrate the capability of *RecDEVS* to model applications at various abstraction levels, ranging from the high-level automotive example to the low-level implementation of an cellular automaton.

## 5.1 Autovision

The first application example is based on the AutoVision scenario introduced by Claus et al. [16]. It is taken from the automotive domain and consists of several distinct components for vision enhancement and automated object recognition aimed to a driving assistance scenario. It switches between various components, implementing a shape- or a taillight-based object recognition depending whether the car is on an open road at daylight or inside a dark tunnel. Fig. 5.1 gives an overview over the following components and their interaction:

*Sensor:* An image sensor that provides pictures to all requesting components.

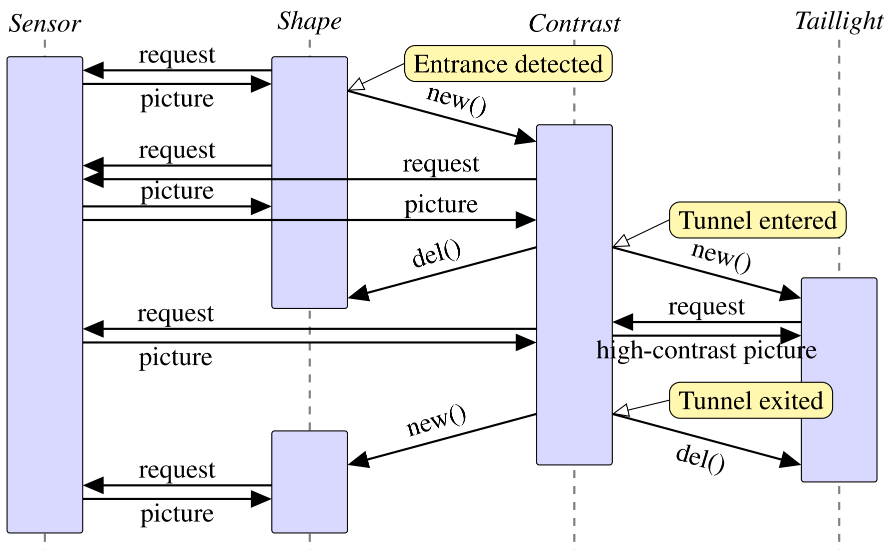


Figure 5.1: Sequence Diagram of the AutoVision Example

*Shape:* Performs picture requests to the Sensor and scans the result for important shapes (e.g., other cars). The detected shapes are provided to the driver. If the shape of a tunnel entrance is found, then the *Contrast* component is invoked.

*Contrast:* Enhances the contrast of a *Sensor* picture and recognizes, when the car enters or leaves a tunnel, in which case it activates or suspends other components.

*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 reliably.

The different situations inside and outside a tunnel trigger the reconfiguration of *Shape, Contrast,* and *Taillight*, respectively. The implemented model consists of four *RecDEVS* components, which switch their state values from ‘deleted‘ to ‘active‘ and vice versa.

### 5.1.1 $\mathrm{C}_{Sensor}$

The Sensor implementation normally stays in an idle *Waiting* state for infinite time. It will be activated by external transitions when some other component asks for the current sensor data in form of a picture. The address of the requesting component is stored as part of the component state so that is available to $\lambda$ in the *SendPic* state. The timeout of $1ns$ is intended to model the internal processing time of the image sensor until it can answer a request.

Note that the two occurrences of $req^{ID}_{Model}$ at the confluent transition are actually two different components. The output message uses still the previously stored address, while the new address is only stored afterwards.

The formal and graphical *RecDEVS* specification of $\mathrm{C}_{Sensor}$ is as follows. A screenshot of the equivalent UPPAAL model denoted as *roi* is presented in Fig. 5.2.

$$S = \{\text{“Waiting”}, \text{“SendPic”}\} \times I$$

$$s_0 = \big(\text{“Waiting”}, \emptyset\big)$$

$$X = I \times I \times \text{“request”}$$

$$Y = I \times I \times \text{“picture”}$$

$$\delta_{\text{ext}}(((\text{“Waiting”}, i), e), \underbrace{(req^{ID}_{Model}, \mathrm{C}^{this}_{Sensor}, data)^{\sharp}}_{msgs}) = (\text{“SendPic”}, req^{ID}_{Model})$$

$$\delta_{\text{int}}(\text{“SendPic”}, i) = (\text{“Waiting”}, \emptyset)$$

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

$$\tau(s', i) = \begin{cases} \infty & \text{if } (s' = \text{“Waiting”}) \\ 1 & \text{if } (s' = \text{“SendPic”}) \end{cases}$$

$$\lambda(\text{“SendPic”}, req^{ID}_{Model}) = \{(\mathrm{C}^{this}_{Sensor}, req^{ID}_{Model}, picture)\}$$

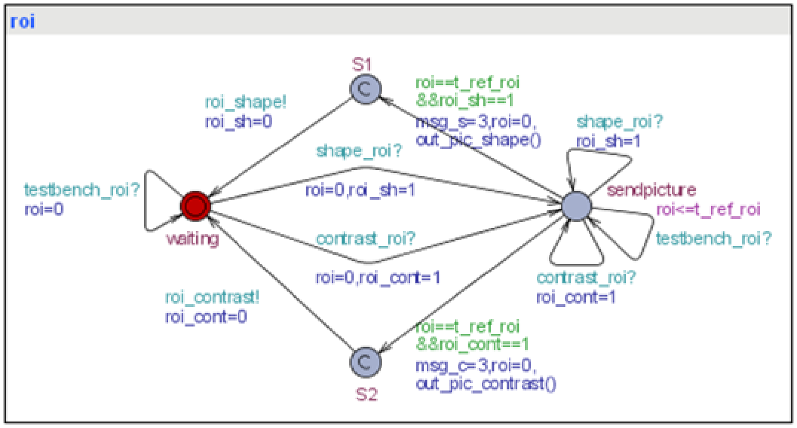
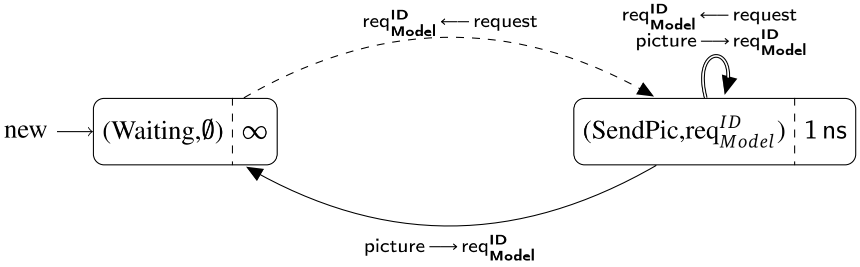


Figure 5.2: UPPAAL Sensor Model

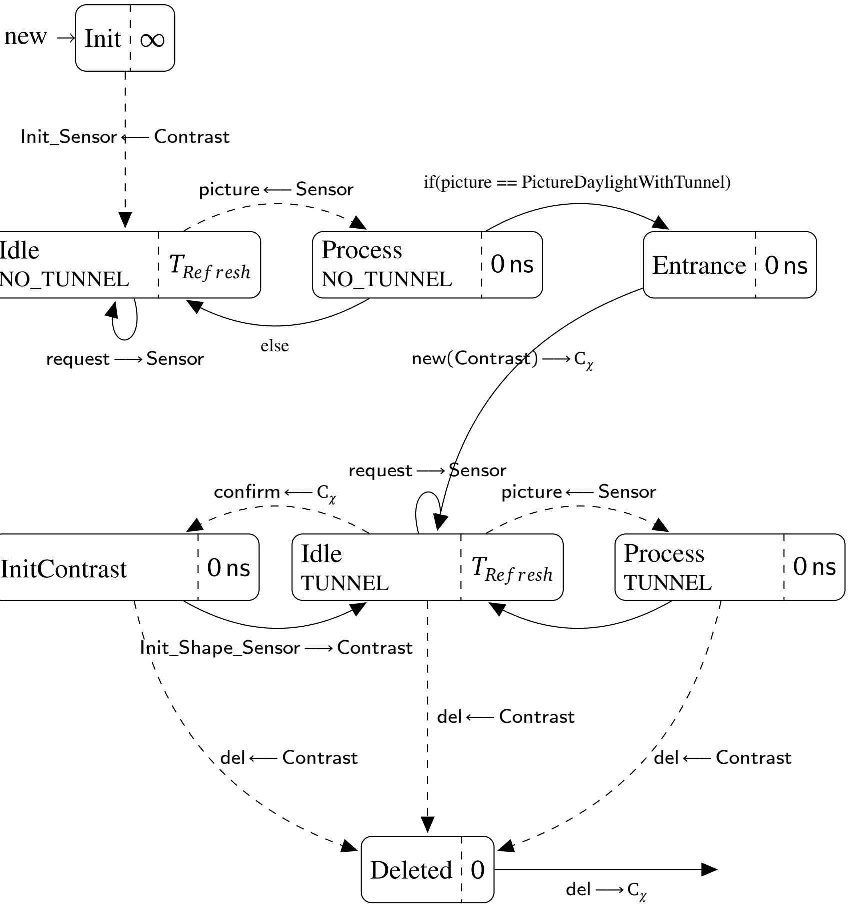

$req^{ID}_{Model} \longleftarrow request$
$req^{ID}_{Model} \longleftarrow request$
$picture \longrightarrow req^{ID}_{Model}$
new
$(Waiting, \emptyset)$
$\infty$
$(SendPic, req^{ID}_{Model})$
1 ns
$picture \longrightarrow req^{ID}_{Model}$

## 5.1.2 $C_{Shape}$

The *RecDEVS* specification of the Shape component illustrates the use of initialization messages to get addresses of other components. After receiving the *Init_Sensor* message this component is able to send messages to the Sensor itself. Upon a new Contrast component was successfully created, Shape sends an initialization message *Init_Shape_Sensor* so that Contrast can communicate with those components as well.

There is no need for transitions back into states without a detected tunnel, because according to Fig. 5.6 the shape component will be deleted if a tunnel has been detected as soon as the contrast component is up and running.

The UPPAAL representation is given in Fig. 5.3 and the *RecDEVS* specification is as follows:



$$
\begin{aligned}
&S = \{\text{“Init”}, \text{“Idle”}, \text{“Process”}, \text{“Entrance”}, \text{“InitContrast”}, \text{“Deleted”}\} \times I \times I \times pic \\
&s_0 = (\text{“Init”}, 0, 0, 0) \\
&X = I \times I \times \{\mathit{picture}, \mathit{confirm}, \mathit{del}\} \\
&Y = I \times I \times \mathbb{N} \times \{\mathit{Init\_Shape\_Sensor}, \mathit{request}, \mathit{new}(\mathit{Contrast}), \mathit{del}\} \\
&\delta_{\text{int}}(s) = \begin{cases} \text{“Idle”} & \text{if } s = \text{“Idle”} \vee s = \text{“Entrance”} \vee s = \text{“InitContrast”} \\ \text{“Idle”} & \text{if } s = \text{“Process” and no Tunnel detected} \\ \text{“Entrance”} & \text{if } s = \text{“Process” and Tunnel detected} \end{cases} \\
&\delta_{\text{ext}}(((\text{“Idle”}, l, m, n), e), \underbrace{(src^{ID}_{Model}, \mathrm{C}^{this}_{Shape}, data)^{\sharp}}_{msg}) = \\
&\qquad \begin{cases} (\text{“Deleted”}, l, m, n) & \text{if } data = del \in msg^{\sharp} \\ (\text{“InitContrast”}, l, ID, n) & \text{elsif } data = confirm(ID) \in msg^{\sharp} \\ (\text{“process”}, l, m, picture) & \text{elsif } data = picture \in msg^{\sharp} \end{cases} \\
&\delta_{\text{ext}}(((\text{“Init”}, l, m, n), e), (src, \mathrm{C}^{this}_{Shape}, Init\_Sensor)^{\sharp}) = (\text{“Idle”}, Sensor, m, n) \\
&\delta_{\text{con}}((s, e), x) = \delta_{\text{ext}}((\delta_{\text{int}}(s), e), x) \\
&\tau = \begin{cases} T_{refresh} & \text{if } s = (\text{“Idle”}, l, m, n) \\ \infty & \text{if } s = (\text{“Deleted”}, l, m, n) \\ 0 & \text{else} \end{cases} \\
&\lambda(s) = \begin{cases} \{(\mathrm{C}^{this}_{Shape}, \mathrm{C}_{\chi}, new(Contrast)) & \text{if } s = \text{“Entrance”} \\ \{(\mathrm{C}^{this}_{Shape}, Sensor, request)\} & \text{if } s = \text{“Idle”} \\ \{(\mathrm{C}^{this}_{Shape}, Contrast, Init\_Shape\_Sensor)\} & \text{if } s = \text{“InitContrast”} \\ \{(\mathrm{C}^{this}_{Shape}, \mathrm{C}_{\chi}, del)\} & \text{if } s = \text{“Deleted”} \\ \diamond & \text{else} \end{cases}
\end{aligned}
\tag{5.1}
$$

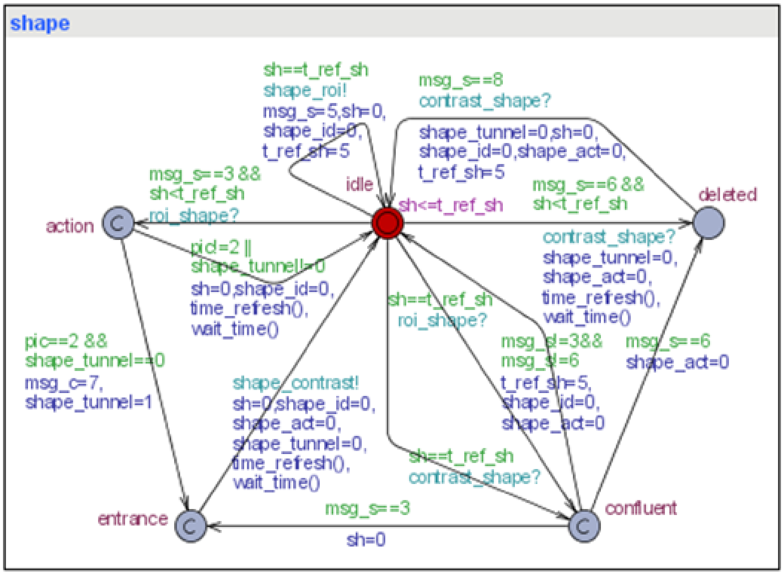
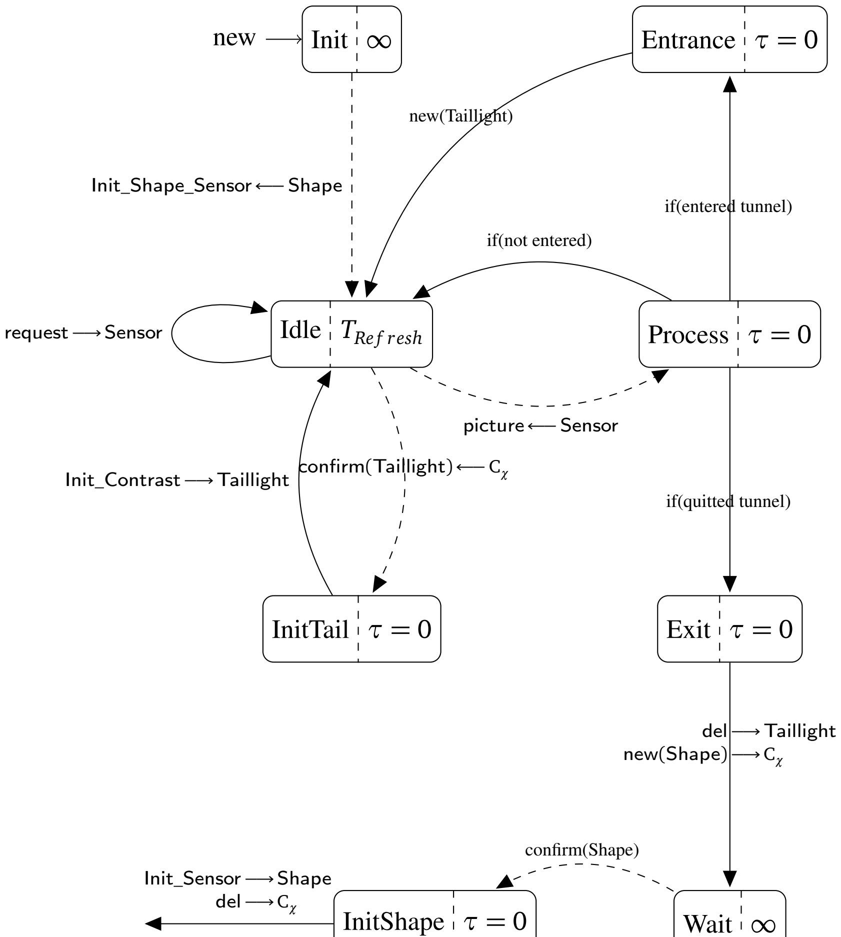


Figure 5.3: UPPAAL Shape Model

### 5.1.3 $C_{Contrast}$

The Contrast component utilizes the fact that we can send multiple messages at the same time. E.g., in the *Exit* state one message is send to the Taillight component and one to the system executive $C_\chi$ at the same time. The equivalent UPPAAL model is given in Fig. 5.4 and the *RecDEVS* specification is as follows:



$$S = \{\text{“Init”}, \text{“Idle”}, \text{“Process”}, \text{“Entrance”}, \text{“Exit”}, \text{“InitTail”}, \text{“InitShape”}, \text{“Wait”}\},$$
$$\times I \times I \times I \times \times Picture$$
$$s_0 = (\text{“Init”}, 0, 0, 0, 0)$$
$$X = I \times I \times \{picture, Init\_Shape\_Sensor, confirm(...)\})$$
$$Y = I \times I \times \{request, Init\_Sensor, Init\_Contrast, del, new(...)\})$$
$$\delta_{\text{int}}(s, ...) = \begin{cases} (\text{“Idle”}, ...) & \text{if } s = \text{“idle”} \vee s = \text{“InitTail”} \vee s = \text{“Entrance”} \\ (\text{“Entrance”}, ...) & \text{if } s = \text{“process” and (entered tunnel)} \\ (\text{“Exit”}, ...) & \text{if } s = \text{“process” and if (leaving tunnel)} \\ (\text{“Idle”}, ...) & \text{if } s = \text{“process” and if (no change)} \\ (\text{“Wait”}, ...) & \text{if } s = \text{“Exit”} \end{cases}$$
$$\delta_{\text{ext}}(((s, k, l, m, pic), e), \underbrace{(src, \mathrm{C}^{this}_{Contrast}, data)^{\sharp}}_{msg}) =$$
$$\begin{cases} (\text{“Idle”}, Shape, Sensor, m, pic) & \text{if } s = \text{“Init”} \wedge Init\_Shape\_Sensor \in msg^{\sharp} \\ (\text{“InitTail”}, k, l, Taillight, pic) & \text{if } s = \text{“Idle”} \wedge confirm(Taillight) \in msg^{\sharp} \\ (\text{“Process”}, k, l, m, picture) & \text{if } s = \text{“Idle”} \wedge picture \in msg^{\sharp} \\ (\text{“InitShape”}, Shape, l, m, pic) & \text{if } s = \text{“Wait”} \wedge confirm(Shape) \in msg^{\sharp} \end{cases}$$
$$\delta_{\text{con}}((s, e), x) = \delta_{\text{ext}}((\delta_{\text{int}}(s), e), x)$$
$$\tau(s) = \begin{cases} T_{refresh} & \text{if } s = \text{“Idle”} \\ \infty & \text{if } s = \text{“Init”} \vee s = \text{“Wait”} \\ 0 & \text{else} \end{cases}$$
$$\lambda(s, Shape, Sensor, Taillight, pic) =$$
$$\begin{cases} \{(\mathrm{C}^{this}_{Contrast}, Taillight, Init\_Contrast)\} & \text{if } s = \text{“InitTail”} \\ \{(\mathrm{C}^{this}_{Contrast}, Sensor, request)\} & \text{if } s = \text{“Idle”} \\ \{(\mathrm{C}^{this}_{Contrast}, \mathrm{C}_{\chi}, new(Taillight))\} & \text{if } s = \text{“Entrance”} \\ \{(\mathrm{C}^{this}_{Contrast}, \mathrm{C}_{\chi}, new(Shape), (\mathrm{C}^{this}_{Contrast}, Taillight, , del))\} & \text{if } s = \text{“Exit”} \\ \{(\mathrm{C}^{this}_{Contrast}, \mathrm{C}_{\chi}, del), (\mathrm{C}^{this}_{Contrast}, Shape, Init\_Sensor)\} & \text{if } s = \text{“InitShape”} \\ \diamond & \text{else} \end{cases}$$

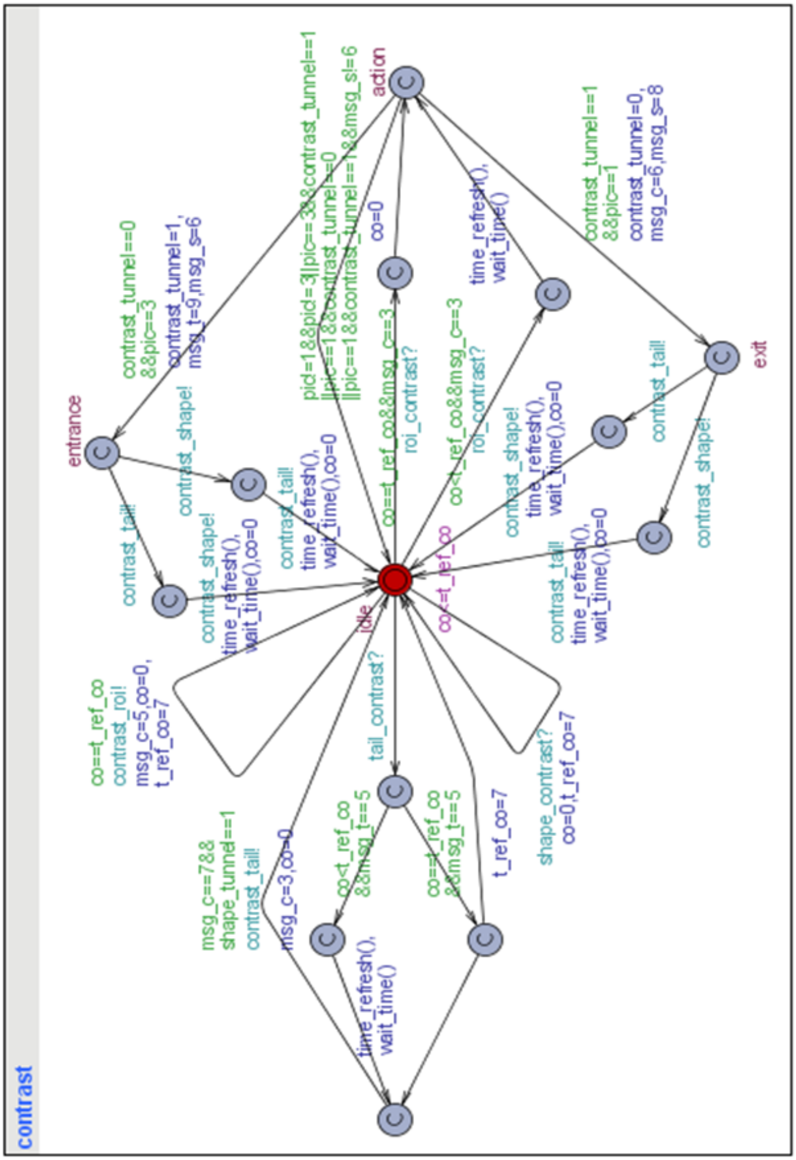


Figure 5.4: UPPAAL Contrast Model

### 5.1.4 $\mathrm{C}_{Taillight}$

The Taillight component shows no additional features. It is a straightforward implementation of the requirements derived from Fig. 5.1. The *RecDEVS* specification is as follows, its UPPAAL representation is given in Fig. 5.5.

$$S = \{\text{“Init”}, \text{“Idle”}, \text{“Process”}, \text{“Deleted”}\} \times I \times picture$$

$$s_0 = (\text{“Init”}, 0, \emptyset)$$

$$X = I \times I \times \{picture, Init\_Contrast, del\}$$

$$Y = I \times I \times \{request, del\}$$

$$\delta_{\text{int}}(s) = \text{“Idle”}$$

$$\delta_{\text{ext}}((s', m, pic), e), \overbrace{(src, \mathrm{C}^{this}_{Taillight}, data)^{\sharp}}^{msgs}) =$$

$$\begin{cases} (\text{“Idle”}, Contrast, pic) & \text{if } s' = init \wedge data = (\text{“Init\_Contrast”}) \in msgs \\ (\text{“Deleted”}, 0, \emptyset) & \text{elsif } s' = idle \wedge data = del() \in msgs \\ (\text{“Process”}, m, picture) & \text{elsif } s' = idle \wedge data = picture \in msgs \end{cases}$$

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

$$\tau(s, k, l, pic) = \begin{cases} T_{refresh} & \text{if } s = \text{“Idle”} \\ \infty & \text{if } s = \text{“Init”} \\ 0 & \text{if } s = \text{“Process”} \vee s = \text{“Deleted”} \end{cases}$$

$$\lambda(s, k, l, pic) = \begin{cases} \{(\mathrm{C}^{this}_{Taillight}, Contrast, request)\} & \text{if } s' = \text{“Idle”} \\ \{(\mathrm{C}^{this}_{Taillight}, \mathrm{C}_{\chi}, del\} & \text{if } s' = \text{“Deleted”} \\ \diamond & \text{else} \end{cases}$$

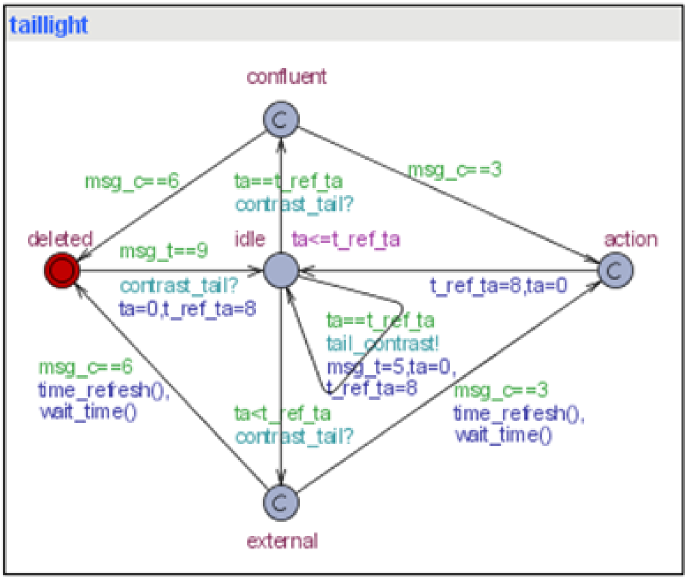
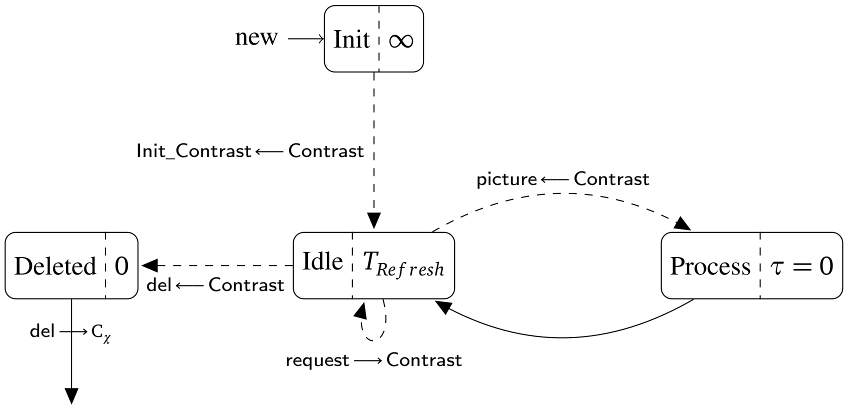


Figure 5.5: UPPAAL Taillight Model



## 5.1.5 Verification Results

In the following subsections we outline the most important verification features for reconfigurable systems by means of the AutoVision example. We will describe their relevance during the development of reconfigurable systems and their UPPAAL notation. In Tab. 5.1 one can find a summary of the important statements, a corresponding example for the AutoVision application, and verification results for the tested system specification properties. The table entries denote some properties as *Not Satisfied* represent design flaws in the first AutoVision implementation. After identifying them in the UPPAAL verification process they may easily be fixed in the corresponding *RecDEVS* model.

### Reachability and Deadlocks

Beside reconfiguration specific properties, there is also a wide variety of important conventional system properties. These properties can be checked by the presented approach as well. UPPAAL enables the verification of statements that address the reachability of specific states of a component. E.g., the UPPAAL statement "`E<>(P1.s1)`" checks whether the state `s1` of component `P1` can be reached. The UPPAAL statement "`E[](P1.s1)`" checks whether that state can be reached at any time during system execution.

UPPAAL also allows the examination of complex system states, which are composed from multiple components. An UPPAAL statement of the form "`(Comp1.state1 and Comp2.state2)`" refers to a combination of two atomic component states. All components that are not explicitly mentioned may have arbitrary states.

As each *RecDEVS* state has a direct representation in UPPAAL it is thus possible to examine the reachability of a system state by verifying UPPAAL specification statements like "`E<>(P1.s1 and P2.s2 and P4.s4)`" to answer the question whether a complex system state is actually reachable, or "`E[](P1.s1 and P2.s2 and P4.s4)`" to verify whether a state is always reachable. Please note that this approach does only work in one direction. Each *RecDEVS* state has a representing state in UPPAAL, however as Fig. 4.5 illustrated multiple UPPAAL states may have been created during the transformation modeling state transitions.

The existence or absence of deadlocks can also be verified by means of the system specification "`A[] not deadlock`". A deadlock is a system state in which no transition can be triggered and thus the system execution comes to a hold. The global deadlock detection is directly useable for *RecDEVS* and does not require any modifications.

## Communication and Timing

An arbitrary UPPAAL state transition from $P1.S1$ to $P1.S2$ will be taken, if the verification statement "`E<> P1.S1 imply P1.S2`" holds true. And because each *RecDEVS* state is represented by a corresponding UPPAAL state an arbitrary *RecDEVS* transition $\delta(S_1) = S_2$ will be taken, if it can be verified by the same UPPAAL statement.

The *imply* keyword also allows for the verification of the *RecDEVS* output function $\lambda$. As previously explained, *RecDEVS* messages are realized by means of synchronization channels in UPPAAL. Unfortunately, UPPAAL does not support the explicit notation of synchronization channels in verification statements. However, each synchronization channel is directly bound to a specific state transition from $P1.S1$ to $P1.S2$. Thus, if one wants to make some statements about a *RecDEVS* message, one has to search the corresponding synchronization channel in UPPAAL and take its transitions start and end state ($P1.S1$ and $P1.S2$). The occurrence of specific *RecDEVS* messages can then be verified with statements of the form "`E<> P1.S1 imply P1.S2`".

By introducing additional local clocks as explained in Section 4.1.3, it is also possible to verify timing constraints. A verification statement like "`E<>(P1.S1 imply P1.S2 imply P1.S3) and t` $\leq 6$" can guarantee that the specified path must not take more than 6 time units. This can also be used to verify whether a state will always be left before the annotated timeout $\tau$ and, thus, its internal transition will never be used.

## Dynamic Resource Allocation

For the implementation of a dynamically reconfigurable hardware system it is of crucial interest to analyze whether there are enough resources for the execution of the reshaped system. As the utilized resources will change during runtime, this question is not trivial to answer. UPPAAL can be used for the exploration of such resource requirements.

Under the assumption that `P1` to `P4` are the only components of a system, the statement “`E[](P1.deleted or P2.deleted or P3.deleted or P4.deleted)`” will only hold true if at least one of the four components is deleted at all times. Thus, a system with resources for three reconfigurable components will be sufficient in this case. It is even possible to optimize this approach by suggesting implementation specific variants of the resource constraint. The statement “`E[]((P1.deleted and P2.deleted) or (P3.deleted and P4.deleted))`” can guarantee that the specified system specification will always have at least either the combination `P1` and `P2` or the combination `P3` and `P4` deleted. Depending on the size of the different components this property may provide stronger information on the required resources than the general resource constraint property, thus resulting in smaller designs.

Unfortunately, there is no simple way to model more advanced wildcards in the UPPAAL verification statements. If the set of components becomes larger then the verification statements will have to be larger as well. To overcome this problem it would be possible to generate the verification statements with the help of an additional, implementation-specific tool or programming script.

### Reconfiguration Activities

Another question is related to the existence of a specific reconfiguration activity, i.e., if component *A* will ever trigger a specific reconfiguration to create component *B*. According to the message-based reconfiguration scheme described in Section 4.1.3 for *RecDEVS*, this always requires a corresponding message `new(B)` and will be answered with `confirm(id)`.

*RecDEVS* defines these reconfiguration specific messages as conventional messages, meaning that they are handled by the same mechanisms as all other communication messages. Consequently, we can extend the same verification techniques described before to evaluate the occurrence of conventional *RecDEVS* messages to gain insight into reconfiguration activities. So, the call of a specific reconfiguration message to create a new component can be tested by verifying the specification statement “`E<>(P1.S1 imply P1.S2)`”, where the transition from `S1` to `S2` contains a synchronization channel, which emits the $new(B)$ message. This statement is only true when a direct transition from state `S1` to `S2` of the model `P1` will be taken.

It is also possible to check the existence of reconfiguration events by verifying whether a component is in special states representing the initial state or the *Dead* state. As the only way into these states is by messages which are part of the reconfiguration mechanism, a component is in such a state indicates that the corresponding reconfiguration event must have happened.

### 5.1.6 SystemC Results

In order to examine the SystemC based implementation, the AutoVision example has also been implemented as a transactional model with UML as the underlying MoC. We used the Rhapsody UML modeling tool from IBM [27] for design entry and system model execution. Fig. 5.6 shows the resulting sequence diagram aimed to give an overview over the components and their interaction.

This model covers a variety of features which allow the evaluation of MoC specific properties:

- All four components run in parallel and thus allow the evaluation of typical concurrency effects.
- The regular timing intervals for *Sensor* requests target timed MoCs.
- The AutoVision model allows multiple competing messages (e.g., from *Shape* and *Contrast*) occurring at the same time. While this behavior is not supported by the UML MoC, it has eventually to be considered in more detailed *RecDEVS* and SystemC models.

Originating from the UML model two different transformation paths have been implemented.

First, a horizontal transformation into *RecDEVS* has been performed. The resulting intermediate *RecDEVS* model has subsequently been transformed into a SystemC model as detailed in Section 4.2. The mapping from *RecDEVS* to SystemC models is performed fully automatic.

In a second approach a direct vertical transformation of the AutoVision example from UML into SystemC has been performed directly. No intermediate MoCs have been exercised on this path.

As a result of both approaches, we yield two different low-level SystemC models. Both originate from the same UML model, but are obtained via different transformation paths. The SystemC models are directly executable, whereas the direct execution

| Description | AutoVision Example | UPPAAL Notation | Property |
|---|---|---|---|
| State Reachability | Is the state `(Shape.idle and Contrast.idle)` reachable? | $E$ <> `(Shape.idle and Contrast.idle)` | Satisfied |
| General Reachability | Is the state `(Shape.deleted)` always reachable? | $E$`[] Shape.deleted` | Satisfied |
| Deadlock Existence | Is the implementation free of deadlocks? | $A$`[] not deadlock` | **Not Satisfied**, all existing deadlocks can be listed by UPPAAL |
| Resource Consumption | Is at least one component always deleted? | $E$`[] (Shape.deleted or Contrast.deleted or Taillight.deleted)` | **Not Satisfied**, e.g., Taillight is created before Shape is deleted |
| Transition Usage | Is the internal transition from `(Shape.new)` to `(Shape.request)` used? | $E$ <> `Shape.new imply Shape.request` | Satisfied |
| Synchronization Channel | Does `Contrast` perform a picture request? | $E$ <> `Contrast.request imply Contrast.idle` | Satisfied |
| Timing Constraint | Can a tunnel be detected in 6 time units? | $E$ <> `(Shape.idle imply Shape.entrance imply Shape.idle) and` $t \leq 6$ | Satisfied |
| Module Reconfiguration | Will Taillight ever be created? | $E$ <> `Contrast.tunnel imply Contrast.idle` (this transition emits the synchronization channel *new(taillight)!*) | Satisfied |

Table 5.1: Set of Verifiable Model Properties of the AutoVision Example

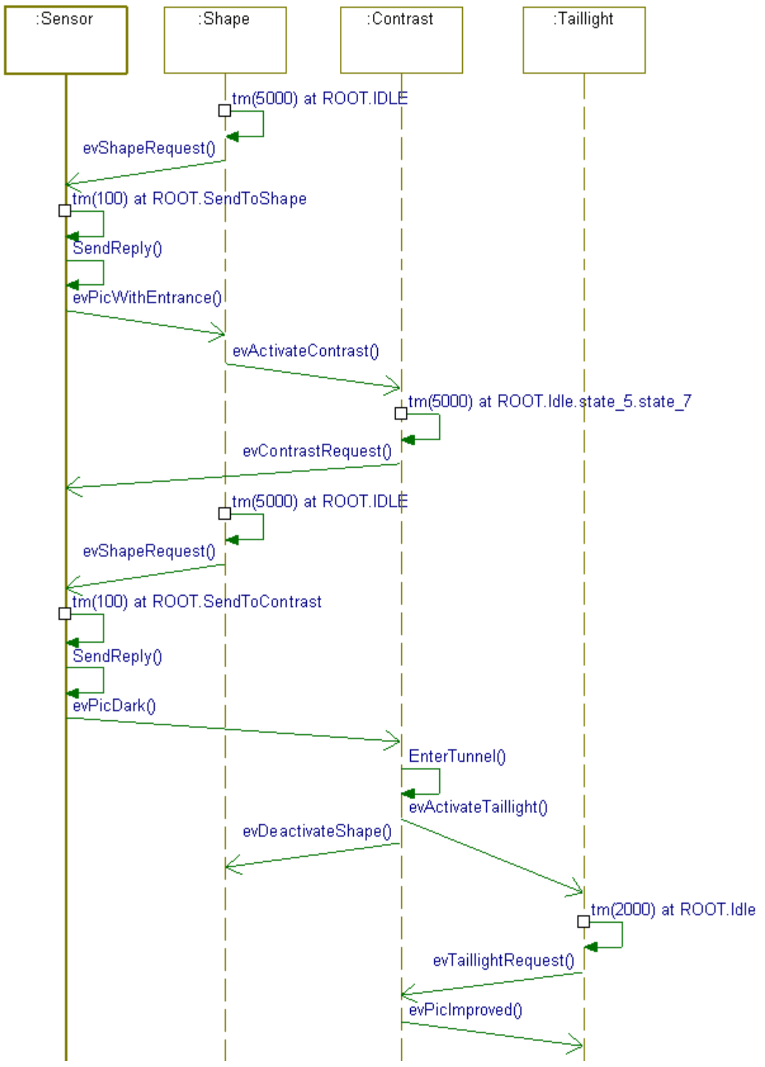


Figure 5.6: UML Sequence Diagram of the AutoVision Example

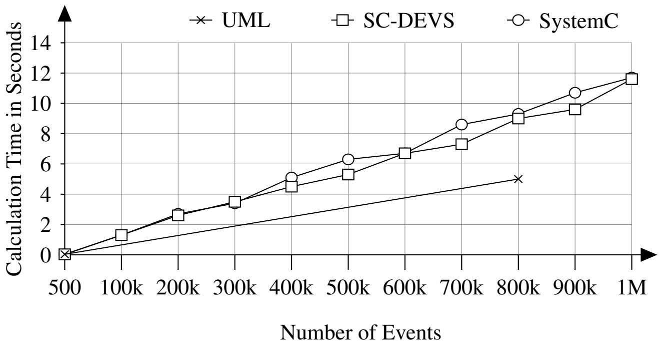


Figure 5.7: Simulation Results for the AutoVision Example

of UML models is supported by the Rhapsody modeling tool. Therefore, comparative performance measurements between the different models are possible. By comparing the SystemC models we can argue about potential performance losses caused by the introduction of additional horizontal transformations. The comparison with the original UML model helps in quantifying any overhead introduced by a vertical MoC transformation (i.e., by lowering the abstraction level).

Fig. 5.7 depicts the results of the performance measurements with the mentioned models of the AutoVision example. The graph visualizes the execution performance for a given number of transaction-level events. In each MoC the same number of events refers to the same state of the AutoVision system and therefore allows a comparison of the model properties.

The execution of the UML model is about two times faster compared to the SystemC variants. This is reasonable because of a higher abstraction level, which does not support the same level of concurrency as SystemC and DEVS models. This allows a less complex and therefore faster simulation. We did not take more measurement points of the UML specification as the difference to the SystemC based models is too big to lead to additional conclusions.

For the different SystemC variants, the directly generated variant is labelled as *SystemC*, whereas the *RecDEVS* based implementation is labelled as *SC-DEVS*. Both implementations show similar performance and scale roughly linearly with the number of events simulated. This clearly demonstrates that an introduction of additional

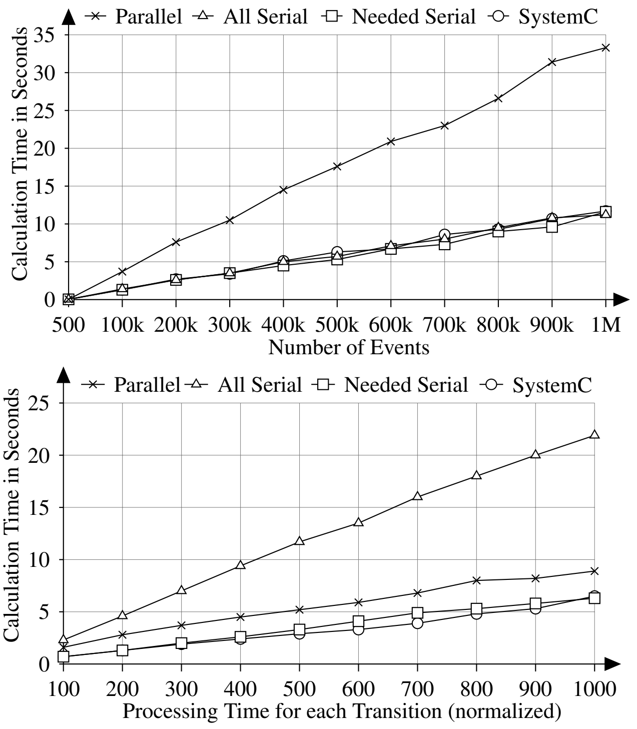


Figure 5.8: Simulation Results for Different SC-DEVS execution variants

horizontal MoC transformations can be (and has been) performed without any substantial overhead.

As the SC-DEVS and the hand-crafted SystemC realization are both executed on the same simulation engine a more detailed analysis can be done here. As already stated in Section 4.2.1, there are three implementation for the SC-DEVS models available, which basically control which of the three transition functions $\delta_{\text{int}}$, $\delta_{\text{ext}}$, and/or $\delta_{\text{ext}}$ are to be executed.

Fig. 5.8 depicts the results of the AutoVision example for the three different execution variants *Parallel, All Serial, Needed Serial* and the direct SystemC implementation. The upper graph visualizes the kernel performance for an increasing number of events. All variants scale linearly with the number of events. While *Needed Serial* and *All Serial* show a similar performance compared to the SystemC simulator, the *Parallel* simulation is approximately 3 times slower (33.3 seconds compared to 11.2 seconds). This stems from the additional kernel overhead for thread handling and synchronization.

The lower Fig. 5.8 depicts the situation for transitions with high processing times. Compared to the *All Serial* execution, the *Parallel* execution is 3 times faster as it can calculate the three transition functions in parallel. The *Needed Serial* option again shows a similar performance to the pure SystemC model. This has been expected as this simulator variant realizes a similar behavior to the original SystemC kernel.

It may seem obvious that the *Needed Serial* should thus be preferred above the other variants. However it may not be able to detect certain design flaws which may originate from synchronization or concurrency issues. The *Parallel* mode is able to detect such flaws and did it multiple times throughout our tests while only being slightly slower in case of our example. So, the selection of the appropriate simulation engine behavior should be decided on a case by case basis.

## 5.2 Game of Life

The original version of the *Game of Life* was first developed by John Conway and has been presented in Gardner [21]. It consists of a simple set of rules that describe the behavior of a cellular automaton. It takes place on an infinite two-dimensional plane that is ordered as a regular grid. So, each position can be identified by its $(x, y)$ coordinates. At each coordinate there is a simple cell that can be in one of two possible states: *alive* or *dead*.

The Game of Life is a simulation game, which means it has no user interaction despite its initial configuration. The initialization specifies the state of all cells at the beginning. Then a simple set of rules is applied in subsequent steps that changes the state of each grid cell.

1. A living cell stays alive if it has two or three living neighbors.
2. A living cell with less than two living neighbors will die (i.e., becomes 'dead' in the next evaluation step)
3. A living cell with more than three living neighbors will die.

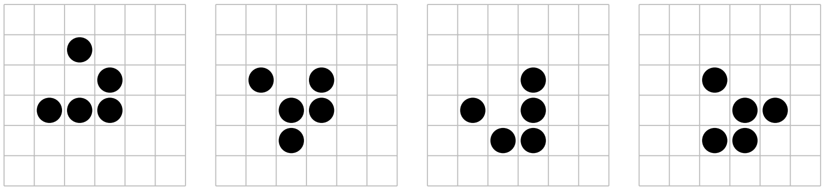


Figure 5.9: A cyclic moving Glider in Game of Life

4. A 'dead' cell will become 'alive' if is has exactly three living neighbors.

These rules are applied to each cell to obtain its next state. After the evaluation is completed the next state becomes the current state and the rules are applied again for the next round.

If the system state is visualized after each computation step, the results can be surprisingly complex, despite the simple set of rules. E.g., Fig. 5.9 depicts some subsequent states of set of cells forming a glider. The black circles mark the living cells on the grid. This object oscillates through 4 different states and "glides" downwards to the right during each oscillation.

Since the memory of computing systems is finite it is not possible to save the infinite number of cell states that exist on an infinite plane. For the glider example from Fig. 5.9 this may quickly become an issue as the glider will continuously move into one direction of the infinite grid and thus the number of crossed grid elements becomes arbitrarily large. A simple approach for a real implementation of a Game Of Life environment simply limits the grid size to a number that can be handled by the actual machine. In this case the grid can be represented as a two-dimensional array. While this approach is easy to implement, the glider would quickly reach the border and can no longer fulfill the rules defined by the Game Of Life.

Another implementation approach does not limit the area of existing cells. Instead, only one kind of cells (either living or dead ones) are explicitly stored together with their coordinates. All cells that do not specifically exist are therefore known to be of the other cell type. As a tradeoff, each cell becomes more complex because it must track its coordinates. Finding cell neighbors becomes more complicated because the coordinates may change during simulation time. The idea behind such an implementation is the observation that in many cases the number of one cell type, mostly the living one, is very small. In contrast to the first implementation approach, the position of cells is now no longer limited by the grid size, however, now the number of concurrently existing cells is limited. This approach is perfectly suited for the glider model, where only a few cells are ever alive at the same point in time.

### 5.2.1 Game of Live for Reconfigurable Systems

The Game of Life is a very interesting candidate for examining reconfigurable systems. By implementing the second presented approach, which implements more complex cells together with their coordinates, it is also an appropriate candidate for implementation as a reconfigurable system. In that case all cells are realized as separate reconfigurable modules.

If a cell becomes dead, it is simply deleted from the reconfigurable network. If it becomes alive, a new component is created. This perfectly shows one of the advantages of reconfigurable systems: existing resources can be reused. Only those component that are actually required at one moment are implemented, the remainder will only be configured as soon as needed. The amount of reconfiguration and concurrent communication is very high, which makes the Game Of Life a good application to prove the fitness of *RecDEVS* for modeling highly dynamic reconfigurable systems.

However, the Game Of Life is not the single perfect example for the whole world of reconfigurable systems. As all cells follow the same rules and thus have identical implementations, the system state and the model execution is highly regular. Compared to the AutoVision example, this Game Of Life implementation has a high reconfiguration dynamic and a high frequency of concurrent communication. But the processes can be regarded as a synchronous behavior without any external influence. The AutoVision example addresses opposite system properties. It demonstrates a set of complex and distinct components that are highly influenced by external asynchronous events. In combination, both examples demonstrate the wide range of systems in which the *RecDEVS* MoC is feasible.

### 5.2.2 *RecDEVS* Implementation of Game of Life

For the *RecDEVS* implementation some minor modifications were necessary. One rule states that a dead cell has to be activated when there are three living neighbors. However, as the cell is currently dead, it is not on an active *RecDEVS* component any more and thus cannot actively count its neighbors. Therefore, a third cell state called *semi-live* is introduced.

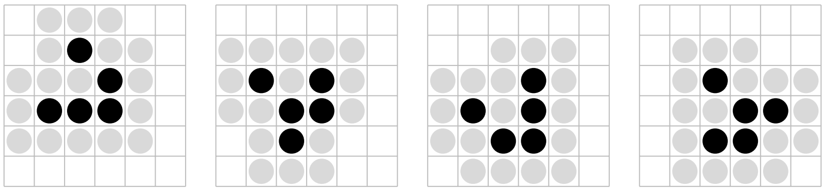


Figure 5.10: The Glider with the Semi-Life Extension

For *RecDEVS* these semi-live cells are active components, however, they are in an inactive state from the Game of Life perspective. These *RecDEVS* components will be configured on all neighboring positions of living cells. Thus, they can receive *RecDEVS* messages and, by that, they can count the number of living neighbors and become living cells if the conditions match. As soon as a cell gets alive, it has to check all neighbor cells and create semi-life *RecDEVS* components if there is not already a component in place. Consequently, if there are no longer any living neighbors, a semi-life cell can be deleted from the *RecDEVS* network.

This results in the following modified rules of the Game of Life:

1. A living cell stays alive if it has two or three living neighbors, otherwise it becomes semi-life.
2. A semi-life cell becomes alive if it is has exactly three living neighbors.
3. A semi-life cell dies if it has no living neighbors.
4. A dead cell becomes semi-life if it gets at least one living neighbor.

For the glider example the modified Game of Life is depicted in Fig. 5.10. The grey shaded circles denote the newly introduced cells that are currently semi-life.

There are some additional aspects of *RecDEVS* which have to be kept in mind when implementing Game of Life. Only the creator of a new component receives its identifier (i.e., address) with the *confirm()* message from the network executive. However, there may be multiple neighbors to a new cell. And while all cells know the logical address $(x, y)$ of the target because it can be calculated by their own address, they do not automatically know the physical *RecDEVS* address corresponding to that logical address. Therefore, a *Mapper* has been introduced as part of the Game Of Life model. As we will see, the *Mapper* will take over the job of creating new components.

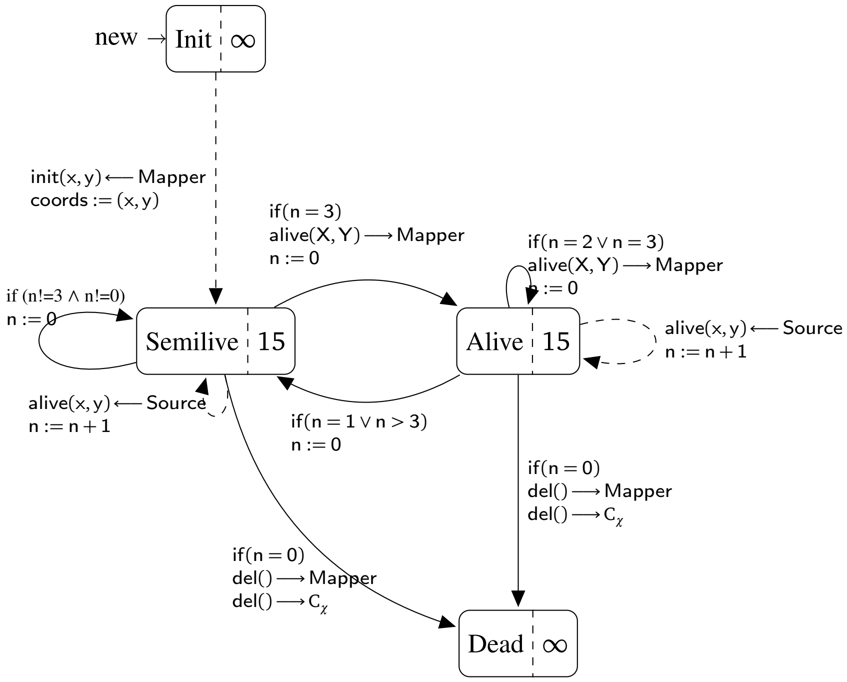


Figure 5.11: Graphical Representation of a Game of Life Cell Specification

It will then assign the correct logical address to each newly created component. Any subsequent communication between cells is directed to the mapper together with the logical address of the target. The mapper will then lookup the corresponding physical address and forward the message to the real recipient.

As all cells are updated synchronously at the same point in time, it may very well happen that there arise situations where a dead cell gets two living neighbors at once. In this case both cells would try to create a semi-life cell with the same coordinates. The presented implementation addresses this issue inside the *Mapper* component. *Mapper* tracks all logical addresses for which a *new()* call is pending at the network executive.

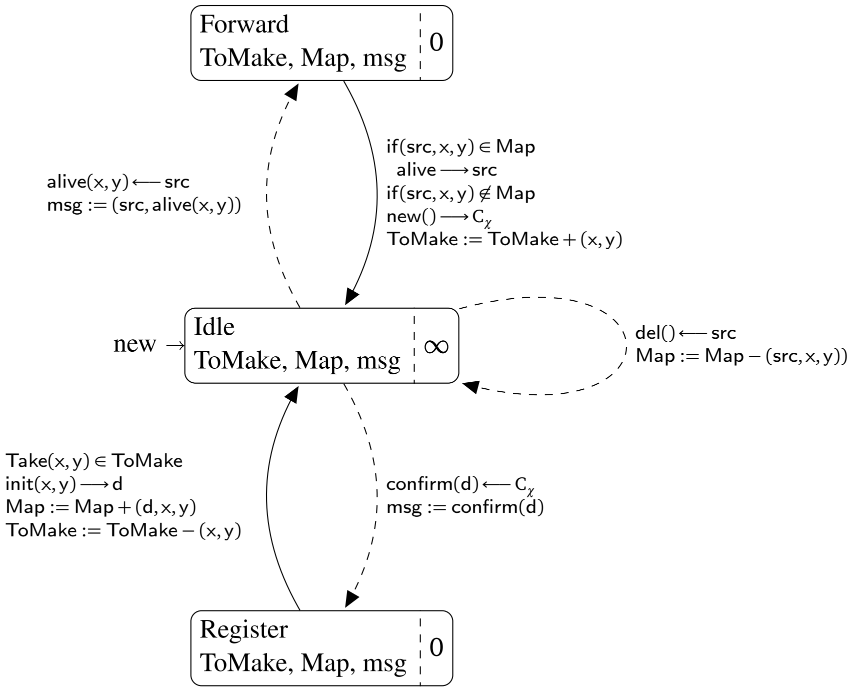


Figure 5.12: Graphical Representation of the Game of Life Mapper Specification

Please recall that *RecDEVS* in its general form does not require that a *new()* call is confirmed immediately. Therefore, all subsequently *new()* calls will simply be ignored as soon as the new coordinates are stored

The resulting *RecDEVS* model for a cell of the Game Of Life is represented by the graphical representation in Fig. 5.11. For ease of reading the list of neighbors for a component $(x, y,)$ is abbreviated by $(X, Y) = \{(x-1, y-1), (x-1, y), (x-1, y+1), (x, y-1), (x, y+1), (x+1, y-1), (x+1, y), (x+1, y+1)\}$. This means that there will be eight *alive()* messages transmitted at the same time, one for each neighbor. The formal *RecDEVS* representation of a single cell equivalent to the graphical representation is given in Fig. 5.13. The address of $C_{Mapper}$ is fixed inside the cell implementations. Of course, it would also be possible to load it together with the coordinates in the $Init$ state.

$$S = \{\{\text{“Init”}, \text{“Semilive”}, \text{“Dead”}\}, (x, y), \mathbb{N}\}$$
$$s_0 = (\text{“Init”}, (0, 0), 0)$$
$$X = I \times I \times \{init(x \in \mathbb{N}, y \in \mathbb{N}), alive(x \in \mathbb{N}, y \in \mathbb{N})\}$$
$$Y = I \times I \times \{alive(x \in \mathbb{N}, y \in \mathbb{N}), del()\}$$
$$\tau(s, I) = \begin{cases} \infty & \text{if } s = \text{“Init”} \vee s = \text{“Dead”} \\ 15 & \text{else} \end{cases}$$
$$\lambda(s, (x, y), n) =$$
$$\begin{cases} \big((\mathrm{C}_{Cell}, C_{Mapper}, del()), (\mathrm{C}_{Cell}, \mathrm{C}_{\chi}, del())\big) & \text{if } n = 0 \\ \big(\mathrm{C}_{Cell}, C_{Mapper}, alive(X, Y))\big) & \text{if } s = \text{“Semilive”} \wedge n = 3 \\ \big(\mathrm{C}_{Cell}, C_{Mapper}, alive(X, Y))\big) & \text{if } s = \text{“Alive”} \wedge (n = 2 \vee n = 3) \end{cases}$$
$$\delta_{\text{ext}}(((s, (x, y), n), e), msg) =$$
$$\begin{cases} \big(\text{“Semilive”}, (x', y'), 0\big) & \text{if } s = \text{“Init”} \wedge msg = init(x', y') \\ \big(\text{“Semilive”}, (x, y), n + 1\big) & \text{if } s = \text{“Semilive”} \wedge msg = alive(x, y) \\ \big(\text{“Alive”}, (x, y), n + 1\big) & \text{if } s = \text{“Alive”} \wedge msg = alive(x, y) \end{cases}$$
$$\delta_{\text{int}}(s, (x, y), n) =$$
$$\begin{cases} \big(\text{“Dead”}, (x, y), 0\big) & \text{if } n = 0 \\ \big(\text{“Semilive”}, (x, y), 0\big) & \text{if } s = \text{“Semilive”} \wedge (n! = 0 \wedge n! = 3) \\ \big(\text{“Semilive”}, (x, y), 0\big) & \text{if } s = \text{“Alive”} \wedge (n = 1 \vee n > 3) \\ \big(\text{“Alive”}, (x, y), 0\big) & \text{if } s = \text{“Alive”} \wedge (n = 2 \vee n = 3) \\ \big(\text{“Alive”}, (x, y), 0\big) & \text{if } s = \text{“Semlive”} \wedge n = 3 \end{cases}$$

Figure 5.13: Formal Representation of a Game of Life Cell Specification

The graphical representation of the corresponding *Mapper* is denoted in Fig. 5.12 and the equivalent formal representation is given in Fig. 5.14. *Mapper* manages two important storage sets. The *Map* contains a mapping between physical addresses and logical coordinates (x,y). When a component deletes itself it does also notify the *Mapper*, which will then remove that mapping from the Map. The *ToMake* set includes coordinates for which a *new* call was required. They are filled in whenever the mapper receives a message for which there exists no entry in *Map* and a new component will be created. Whenever the creations was confirmed one entry from *ToMake* is removed and a corresponding entry in *Map* will be added.

The graphs in Fig. 5.15 and Fig. 5.16 depict the various system activities during the execution of the Glider example in Game Of Live. The Glider iterates cyclically through 4 different system states denoted as *State 1* to *State 4*. Then, the system is back in the original, but shifted state *State 1'* from which the same system activities will continue with a different set of coordinates.

Fig. 5.15 illustrates the number of system messages that are emitted by the active components. Each state is divided into three phases. In the first phase the *Alive* messages are transferred. It can be observed that this is by far the largest amount of concurrent messages during system execution. In the second phase new components are created, which require, to emit a *new(), confirm(), and init()* message for each creation. Finally, in the third phase all components that are no longer used are deleted, which requires two *del()* messages: one for $C_\chi$ and on for *Mapper*.

The amount of reconfiguration during system execution is detailed in Fig. 5.16. Here, each state has two different phases: one in which new components are created and a second one in which components are deleted.

The Game of Life has been implemented as a model in VHDL hardware description in the master thesis by Theisen [52]. The results have been published in Madlener et al. [34] and are summarized in Tab. 5.2. The percentages denote the resource consumption of each component on the target platform Xilinx Virtex 5 XC5VLX110T. It can be observed that this implementation is limited by the utilized FlipFlops, which are mainly used for realizing the communication network. The hardware model has not been optimized towards the Game of Life, instead all cells are generic reconfigurable cells that provide an abstracted *RecDEVS* interface with $\delta_{\text{int}}$, $\delta_{\text{ext}}$, and $\delta_{\text{con}}$.

$$S = \{\text{“Forward”}, \text{“Idle”}, \text{“Register”}\}, \underbrace{\{(x,y)\}}_{ToMake}, \underbrace{\{(d,x,y)\}}_{Map}, I, coords\}$$

$$s_0 = (\text{“Idle”}, \emptyset, \emptyset, 0, 0)$$

$$X = I \times I \times \{confirm(d \in I), alive(x \in \mathbb{N}, y \in \mathbb{N}), del()\}$$

$$Y = I \times I \times \{alive(),\}$$

$$\tau(s, I) = \begin{cases} \infty & \text{if } s = \text{“Idle”} \\ 0 & \text{else} \end{cases}$$

$$\lambda(s, ToMake, Map, src, coords) =$$

$$\begin{cases} \Big(\mathrm{C}_{Mapper}, src, init()\Big) & \text{if } s = \text{“Register”} \\ \Big(\mathrm{C}_{Mapper}, \mathrm{C}_{\chi}, new())\Big) & \text{if } s = \text{“Forward” and } coords \notin Map \\ \Big(\mathrm{C}_{Mapper}, src, alive())\Big) & \text{if } s = \text{“Forward” and } coords \in Map \end{cases}$$

$$\delta_{\text{ext}}(((\text{“Idle”}, t, m, s, c), e), msg) =$$

$$\begin{cases} (\text{“Forward”}, t, m, src, (x,y)) & \text{if } msg = (src, dest, alive(x,y)) \\ (\text{“Idle”}, t, m - (src, *, *), s, c) & \text{if } msg = (src, dest, del()) \\ (\text{“Register”}, t, m, d, c)) & \text{if } msg = (src, dest, confirm(d)) \end{cases}$$

$$\delta_{\text{int}}((s, t, m, src, (x,y))) =$$

$$\begin{cases} (\text{“Idle”}, t - (x,y), m + (src, x, y)), 0, 0) & \text{if } s = \text{“Register”} \\ (\text{“Idle”}, t + (x,y), m, 0, 0) & \text{if } s = \text{“Forward” and } (x,y) \notin Map \\ (\text{“Register”}, t, m, 0, 0) & \text{if } s = \text{“Forward” and } (x,y) \in Map \end{cases}$$

Figure 5.14: Formal Representation of the Game of Life Mapper Specification

| Component | LUTs | FlipFlops |
|---|---|---|
| Network Executive | 974 (1.4%) | 680 (7.6%) |
| Game of Life Cell | 187 (0.2%) | 169 (1.8%) |

Table 5.2: Game of Life Implementation Results

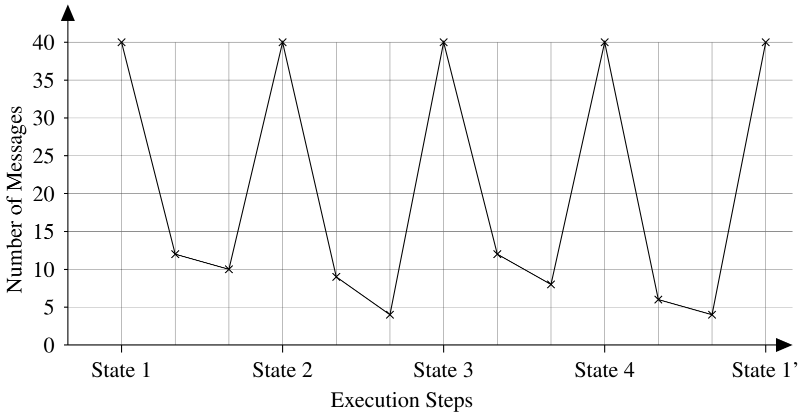


Figure 5.15: Parallel System Messages while executing the Game Of Life Glider

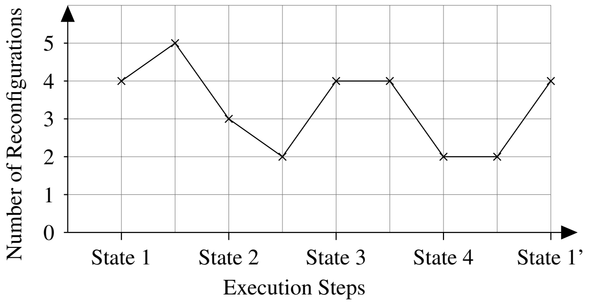


Figure 5.16: Reconfiguration Activities while executing the Game Of Life Glider

# 6 Conclusions and Outlook

## 6.1 Conclusion

In this work the new *RecDEVS* Model of Computation was developed and its suitability for specifying reconfigurable hardware systems was demonstrated. As a formal specification model it introduces new aspects to the development of reconfigurable hardware systems which are not feasible with less formal approaches based on lower abstraction levels.

Before *RecDEVS* was defined two preliminary steps have been performed. First, a novel design flow methodology based on model transformations has been investigated and presented in Section 2.1. Horizontal model transformations create equivalent models on the same abstraction level, but exploiting different MoCs. This allows to benefit from special features, tools and insights from those MoCs without having to incorporate them into our own MoC. Vertical transformation steps set the path from more abstract high-level models down to an actual hardware implementation. The design methodology can easily be extended by additional horizontal transformations. As long as the transformations preserve the functional equivalence, no additional vertical transformations are required. Instead, the newly added model should be transformed horizontally into another model for which a vertical transformation already exists and reuse that transformation.

The second preliminary step was the introduction of reconfigurable hardware systems. As this development field is rather new compared to other integrated circuit design processes, there is not yet a clear list of requirements for a MoC suitable for reconfigurable system specification. In Section 2.2 the target architecture for this work has been presented. It is a generalized version of the most established architecture for reconfigurable hardware, the Field Programmable Gate Arrays (FPGA). However, the results of this work are not limited to FPGAs, but can be applied to other architectures as well. A set of important properties has been defined in Section 2.2 that should be accounted for when developing a model for reconfigurable hardware systems.

With these properties and the design flow methodology in mind the *RecDEVS* formalism has been created, specified, and explained in Chapter 3. *RecDEVS* does not try to reinvent modeling, instead it is based on the existing and well-established DEVS formalism. The *RecDEVS* formalism consists of multiple interacting components. Each component is a timed state machine with different transitions depending on the occurrence of external or internal events. The components interact with a message based communication scheme. Reconfiguration can be realized by a very small set of special system messages and by an additional component, the network executive $C_{\chi}$.

Based on *RecDEVS* three different model transformations are presented in Chapter 4. A horizontal transformation between *RecDEVS* and the UPPAAL model checking environment has been proposed. This enables the verification not only of conventional model checking properties for a *RecDEVS* specification such as reachability, but also for the verification of reconfiguration specific properties, such as if there will be enough reconfigurable resources throughout the execution of the specified model. Verifying reconfiguration specific features is a new research area that has not yet been addressed to the authors knowledge but that may bear much potential for handling the complexity of reconfigurable systems. Two vertical transformations for *RecDEVS* were presented as well: the first one transforming into the SystemC specification language and the second one creating a VHDL model.

In Chapter 5 the so far obtained results are applied to two different examples. The *AutoVision* example is a high-level example for image detection in an automotive environment. The example has been modeled in *RecDEVS*, in UPPAAL, and in SystemC. With the UPPAAL representation various insightful verification results were achieved and the comparison with a hand optimized SystemC implementation demonstrates that the introduction of a vertical model transformation step has no major impact on the execution performance of the example. The *Game of Life* models a cell-based automaton and was chosen as example because of its reconfiguration dynamic. It has been implemented in VHDL notation and demonstrates the feasibility of the *RecDEVS* approach on FPGA platforms.

## 6.2 Outlook

For a fully integrated and flawlessly working design methodology for reconfigurable hardware systems there are many important aspects that should be researched as well. While this work presents possible design flows and with *RecDEVS* a formally specified model that is directly targeted for specifying reconfigurable hardware models, there still has to be done research on how to actually model reconfigurable systems.

As stated in this work, reconfiguration adds a whole new dimension of complexity to the already existing design methodologies for classical hardware architectures. So, additional research is necessary on how to find an optimal partitioning into suited components and an optimal scheduling of such components onto the available reconfigurable resources. Those design steps will have to find trade-offs between the additional time and effort for the required reconfiguration steps and the benefit of saving resources by reconfiguring existing resources when they are not needed any more for an actual computation step.

Verifying reconfigurable hardware systems is another area with a lot of potential for future research. Applying model checking algorithms to classical timed state machines resulted in a lot of important verifiable properties such as state reachability or the existence of deadlocks. Extending verification to reconfigurable systems bears the potential for providing huge benefits for a system designer. “Can the actual implementation reliably be executed with a given set of resources?” is just one of those properties that has been worked out in this work, but a lot of other new interesting questions may arise from examining this area more closely.

# Bibliography

[1] Accellera System Initiative. TLM-2 Whitepaper. Website. URL `www.accellera.org/downloads/standards/systemc/`. Last visited 2013-04-28.

[2] Hessa Aljunaid and Tom J. Kazmierski. SEAMS - a SystemC environment with analog and mixed-signal extensions. In *Proceedings of the IEEE International Symposium on Circuits and Systems ISCAS'04*, pages 281–284, 2004.

[3] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Comliant Metamodeling Framework with Graph Transformations. In *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and Applications*, pages 361–375, 2006.

[4] Maxim Anikeev, Felix Madlener, Andreas Schlosser, Sorin A. Huss, and Christoph Walther. Automated Verification for an Efficient Elliptic-Curve Algorithm - A Case Study. In *Proceedings of the 9th International Scientific and Applied Conference - Information Security*, 2007.

[5] Maxim Anikeev, Felix Madlener, Andreas Schlosser, Sorin A. Huss, and Christoph Walther. A Viable Approach to Machine-Checked Correctness Proof of Algorithm Variants in Elliptic Curve Cryptography. In *Workshop on Program Semantics, Specification and Verification: Theory and Applications at the 5th International Computer Science Symposium in Russia*, pages 95–101, 2010.

[6] Fernando J. Barros. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation TOMACS'97*, 7(4):501–515, 1997.

[7] Fernando J. Barros. Dynamic structure multiparadigm modeling and simulation. *ACM Transactions on Modeling and Computer Simulation TOMACS'03*, 13(3): 259–275, 2003.

[8] Gerd Behrmann, Alexandre David, Kim G. Larsen, Oliver Moller, Paul Pettersson, and Wang Yi. UPPAAL - Present and Future. *IEEE Conference on Decision and Control*, 2001.

[9] Carmen-Veronica Bobeanu, Eugene J. H. Kerckhoffs, and Hendrik Van Landeghem. Modeling of discrete event systems: A holistic and incremental approach using Petri nets. *ACM Transactions on Modeling and Computer Simulation TOMACS'04*, 14(4):389–423, 2004.

[10] Kiran Bondalapati and Viktor K. Prasanna. Reconfigurable computing systems. *Proceedings of the IEEE*, 90(7):1201–1217, 2002.

[11] Christoper X. Brooks, Edward A. Lee, and Stavros Tripakis. Exploring models of computation with Ptolemy II. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[12] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Steve Neuendorffer, Yang Zhao, and Haiyang Zheng. Ptolemy II Heterogeneous Concurrent Modeling and Design in Java - Memorandum UCB/ERL M05/21. Technical report, University of California at Berkely, 2005.

[13] Ewerson Carvalho, Ney Calazans, Eduardo Briao, and Fernando Moaraes. PaDReH: A framework for the design and implementation of dynamically and partially reconfigurable systems. In *Proceedings of the 17th Symposion on Integrated Circuits and System Design*, pages 10–15, 2004.

[14] Edmund M. Clarke. The birth of modelchecking. *25 Years of Modelchecking Festschrift*, pages 1–26, 2008.

[15] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Lecture Notes in Computer Science*, volume 131, pages 52–71. Springer, 1981. ISBN 3-540-11212-X.

[16] Christopher Claus, Walter Stechele, and Andreas Herkersdorf. Autovision - A Run-time Reconfigurable MPSoC Architecture for Future Driver Assistance Systems. *it - Information Technology*, 49(3):181–186, 2007.

[17] Stephen D. Craven and Peter M. Athanas. High-Level Specification of Runtime Reconfigurable Designs. In *Proceedings of the International Conference on Engeneering of Reconfigurable Systems and Algorithms ERSA'07*, pages 280–283, 2007.

[18] Hernán P. Dacharry and Norbert Giambiasi. A formal verification approach for DEVS. In *SCSC: Proceedings of the 2007 Summer Computer Simulation Conference*, pages 312–319, 2007. ISBN 1-56555-316-0.

[19] Mahmood Fazlali, Mojtaba Sabeghi, Ali Zakerolhosseini, and Koen Bertels. Efficient task scheduling for runtime reconfigurable systems. *Journal of Systems Architecture: The EUROMICRO Journal*, 56(11):623–632, 2010.

[20] Terry Filiba, Jackie M.K. Leung, and Vinayak Nagpal. VHDL Code Generation in the Ptolemy II Environment. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[21] Martin Gardner. The fantastic Combinations of John Conway's New Solitaire Game "Life". *Scientific American*, pages 120–123, 1970.

[22] Martin Geisse. Integration of domain-specific languages. Master's thesis, Technische Universität Darmstadt, Germany, 2008.

[23] Orna Grumberg and Helmut Veith, editors. *25 Years of Modelchecking Festschrift*. Springer LNCS, 2008.

[24] Aarti Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in System Design*, 1:151–238, 1992. ISSN 0925-9856.

[25] Fernando Herrera and Eugenio Villar. A Framework for Heterogeneous Specification and Design of Electronic Embedded Systems in SystemC. *ACM Transactions on Design Automation of Electronic Systems*, 12(3):1–31, 2007. ISSN 1084-4309.

[26] Pao-Ann Hsiung, Chao-Sheng Lin, and Chih-Feng Liao. Perfecto: A SystemC-based design-space exploration framework for dynamically reconfigurable architectures. *ACM Transactions Reconfigurable Technology and Systems*, 1(3): 1–30, 2008. ISSN 1936-7406.

[27] IBM. Telelogic Rhapsody, 2009. URL `http://modeling.telelogic.com/products/rhapsody/`. [Accessed: Apr. 16, 2009].

[28] Neil G. Jacobson. *The In-System Configuration Handbook*. Kluwer Academic Publishers, 2004. ISBN 1-4020-7655-X.

[29] Lech Jówiak, Nadia Nedjah, and Miguel Figueroa. Modern development methods and tools for embedded reconfigurable systems: A survey. *Integrated VLSI Journal*, 43(1):1–33, 2010.

[30] Andreas Kühn, Felix Madlener, and Sorin A. Huss. Resource Management for Dynamic Reconfigurable Hardware Structures. In *2nd International Workshop on Reconfigurable Communication-centric System-on-Chips ReCoSoC'06*, Montpellier, France, 2006.

[31] William K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall Modern Semiconductor Design Series. Prentice Hall PTR, 2005. ISBN 0131433474.

[32] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer STTT'97*, 1 (1-2):134–152, 1997.

[33] Felix Madlener. Entwicklung eines Gargbage-Collectors für rekofigurierbare Hardwarestrukturen. Master's thesis, Technische Universität Darmstadt, Germany, 2005.

[34] Felix Madlener, Sorin A. Huss, and Alexander Biedermann. RecDEVS: A Comprehensive Model of Computation for Dynamically Reconfigurable Hardware Systems. In *4th IFAC Workshop on Discrete-Event System Design DESDes'09*, 2009.

[35] Felix Madlener, H. Gregor Molter, and Sorin A. Huss. SC-DEVS: An efficient SystemC Extension for the DEVS Model of Computation. In *ACM/IEEE Proceedings of Design Automation and Test in Europe DATE'09*. ACM/IEEE, April 2009.

[36] Felix Madlener, Marc Stoettinger, and Sorin A. Huss. Novel Hardening Techniques against Differential Power Analysis for Multiplication in GF($2^n$). In *IEEE International Conference on Field-Programmable Technology ICFPT'09*, December 2009.

[37] Felix Madlener, Julia Weingart, and Sorin A. Huss. Verification of Dynamically Reconfigurable Embedded Systems by Model Transformation Rules. In

*4th IEEE/ACM International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS 2010)*, 2010.

[38] Ka Lok Man. SystemC$^{FL}$: a formalism for hardware/software codesign. In *Proceedings of the 2005 European Conference on Circuit Theory and Design*, volume 1, pages 193–196, 2005.

[39] H. Gregor Molter, Felix Madlener, and Sorin A. Huss. A System Level Design Flow for Embedded Systems based on Model of Computation Mappings. In *4th IFAC Workshop on Discrete-Event System Design DESDes'09*, October 2009.

[40] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, page 4, 1965.

[41] Liliana Morihama, Viviana Pasuello, and Gabriel A. Wainer. Automatic verification of DEVS models. In *Proceedings of SISO Spring Interoperability Workshop*, 2002.

[42] Hiren D. Patel and Sandeep K. Shukla. Towards a heterogeneous simulation kernel for system-level models: A SystemC kernel for synchronous data flow models. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(8): 1261–1271, 2005.

[43] Hiren D. Patel, Sandeep K. Shukla, Elliot Mednick, and Rishiyur S. Nikhil. A rule-based model of computation for SystemC: Integrating SystemC and Bluespec for co-design. In *Proceedings of the ACM and IEEE International Conference on Formal Methods and Models for Co-Design MEMOCODE'06*, pages 39–48, 2006.

[44] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982*, Lecture Notes in Computer Science, pages 337–351. Springer, 1982. ISBN 3-540-11494-7.

[45] José L. Risco-Martín, Saurabh Mittal, Bernard P. Zeigler, and Jesús M. de la Cruz. From UML Statecharts to DEVS State Machines using XML. In *IEEE/ACM Conference on Multi-paradigm Modeling and Simulation*, 2007.

[46] Carsten Rust, Achim Rettberg, and Kai Gossens. From high-level Petri nets to SystemC. *IEEE International Conference on Systems, Man and Cybernetics 2003*, 2:1032–1038, 2003. ISSN 1062-922X.

[47] Hesham Saadawi and Gabriel Wainer. On the verification of hybrid DEVS models. In *Proceedings of the 2012 Symposium on Theory and Model Simulation - DEVS Integative M&S Symposium*, pages 21–26, 2012.

[48] Marco Santambragio. *Hardware-Software Codesign Methodologies for Dynamically Reconfigurable Systems*. PhD thesis, Politecnico Di Milano, Italy, 2008.

[49] Andreas Schallenberg, Andreas Herrholz, Philipp A. Hartmann, Frank Oppenheimer, and Wolfgang Nebel. OSSS+R: A framework for application level modelling and synthesis of reconfigurable systems. In *ACM/IEEE Proceedings of the Design and Automation Conference in Europe DATE'09*, 2009.

[50] Hui Shang and Gabriel Wainer. A simulation algorithm for dynamic structure DEVS modeling. In *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, pages 815–822, 2006.

[51] Marc Stöttinger, Felix Madlener, and Sorin A. Huss. Procedures for securing ecc implementations against differential power analysis using reconfigurable architectures. In *Dynamically Reconfigurable Systems - Architectures, Design Methods and Applications*, pages 305–321. Springer, December 2009. ISBN 978-9-04-813484-7.

[52] Alexander Theisen. Entwurf eines Frameworks für rekonfigurierbare DEVS Modelle. Master's thesis, Technische Universität Darmstadt, Germany, 2009.

[53] Florian Thoma, Matthias Kühnle, Phillipe Bonnot, Elena M. Panainte, Koen Bertels, Sebastian Goller, Axel Schneider, Stéphane Guyetant, Eberhard Schüler, Klaus D. Müller-Glaser, and Jürgen Becker. Morpheus: Heterogeneous reconfigurable computing. In *International Conference on Field Programmable Logic and Applications FPL' 07*, pages 409–414, 2007.

[54] Michael Ullmann and Jürgen Becker. Communication concept for adaptive intelligent run-time systems supporting distributed reconfigurable embedded systems. *IEEE Parallel and Distributed Processing Symposium*, pages 8–pp, 2006.

[55] Moshe Y. Vardi. Formal Techniques for SystemC Verification; Position Paper. In *Proceedings of ACM Design Automation Conference DAC'07*, pages 188–192, 2007.

[56] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena M. Panainte. The Molen polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004. ISSN 0018-9340.

[57] Julia Weingart. Verifikation von DEVS Modellen für rekonfigurierbare Systeme. Master's thesis, Technische Universität Darmstadt, Germany, 2009.

[58] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., 2000. ISBN 0127784551.