Verification of Dynamically Reconfigurable Embedded Systems by Model Transformation Rules*

Felix Madlener, Julia Weingart, and Sorin A. Huss
Integrated Circuits and Systems Lab
Technische Universität Darmstadt
Germany
{madlener, weingart, huss}@iss.tu-darmstadt.de

ABSTRACT

This paper describes a methodology for the verification of reconfigurable embedded systems. The reconfigurable systems are described by means of the Reconfigurable Discrete Event Specified System (RecDEVS) computational model and the verification is performed by a model transformation from the RecDEVS model into an equivalent representation for the UPPAAL model checking methodology. We introduce an algorithm for the automatic transformation of such models, which originate from disjoint application domains. This allows the usage of an state-of-the art verification tool for the verification of arbitrary properties of system specifications denoted in RecDEVS. We also present a set of important system properties, which now may be verified. This set includes some fundamental reconfiguration domain specific properties, which were not addressed by previous formal verification methods. The feasibility of this approach is demonstrated for a complex automotive application.

Categories and Subject Descriptors

B.6.3 [**Design Aids**]: Verification; F.1.1 [**Computation by Abstract Devices**]: Models of Computation

General Terms

Verification, Design

Keywords

Reconfigurable Systems, Verification, Design Methodology, Model Transformation, RecDEVS, UPPAAL

1. INTRODUCTION

Reconfigurable hardware architectures have emerged as a promising technique to replace conventional hardware modules in future embedded systems and systems-on-chip. They do not only allow for a fast and easy exploration of design variants to create efficient hardware solutions, but they also support a dynamic, i.e., during runtime, reconfiguration of logic resources.

The introduction of these new design features has a great potential for the development of future embedded systems. The support of dynamic reconfiguration will clearly expand the present limitations of current hardware design. It especially can provide a more flexible and better resource utilization without loosing hardware-specific advantages such as high performance. However, the design problems of increasingly complex embedded systems is even more present for reconfigurable hardware partitions due to this additional dimension of the design space.

In conventional hardware design the problem of increasing complexity is being addressed by rising the abstraction levels in the design phase. One main disadvantage of the existing reconfigurable systems design methodologies is the lack of an established approach, which supports higher abstractions.

In order to establish an exploitation of higher abstraction levels, formal Models of Computation (MoC) were introduced to the design flow. The *Reconfigurable Discrete Event Specified System (RecDEVS)* [11] is such a high-level MoC, which is dedicated to the reconfiguration property. By being an event-based specification with an integrated reconfigurable and timed behavior, this MoC is highly applicable for reconfigurable hardware and real-time software models. It is based on timed automata and consists of a strict formal specification.

In this work we demonstrate how a formal verification technique can be applied to a reconfigurable hardware system specified denoted by means of the RecDEVS approach. However, RecDEVS was not originally targeted towards verification. There are other, more suitable specialized MoCs available for this purpose. 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 [10]. 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.

This paper reviews related work in Section 2. Section 3 motivates a MoC based design flow for both verification and design refinement. It then presents the RecDEVS model and the UPPAAL model checker. Section 4 details how a transformation from RecDEVS into the UPPAAL MoC is being performed and Section 5 shows how the transformed model can be utilized to prove and verify properties of an existing RecDEVS model of a complex application example. We conclude this paper in Section 6.

2. RELATED WORK

There is a variety of publications which emphasize the advantages of MoC-based design flows ([13], [7], and [15]). For this

*This work was supported by CASED (www.cased.de)

CODES+ISSS'10, October 24–29, 2010, Scottsdale, Arizona, USA.

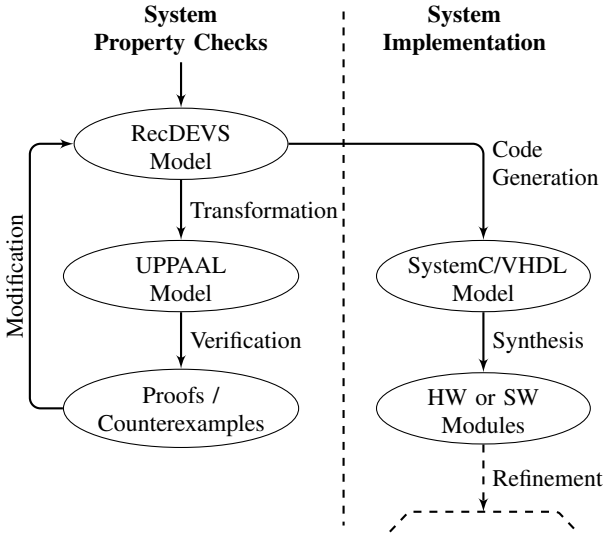

Figure 1: Proposed Design Flow for Reconfigurable Systems

work we have taken the methodology from [13] as reference. It distinguishes horizontal and vertical transformations aimed to denote either the transformation of single implementations or the transformation of complete MoCs.

There are different approaches for modeling dynamic reconfigurable systems based on lower level programming languages like VHDL [16], SystemC [8], or ImpulseC [4]. Due to the lack of a complete formal specification of these languages it is not clear whether they are suitable at all formal verification purposes.

Both the HySAM [2] approach and RecDEVS [11] provide the required formal specification foundation. HySAM splits the descriptions of reconfiguration and functionality, respectively, into two disjoint models which makes a conclusive verification difficult. RecDEVS, because of being based on the DEVS formalism [18], combines both descriptions in a single model and thus supports the verification of function-triggered reconfiguration properties. The SC-DEVS code generation detailed in [12] for SystemC illustrates the capabilities of this computational model for the formal specification of embedded systems with its hardware and software aspects. The flow of implementation steps is summarized in the right hand part of Figure 1.

Regarding the verification of RecDEVS models, there is some preliminary work on the formal verification of the underlying DEVS formalism. The first work [14] implements an own theorem prover, while [17] and [5] benefit to some extend from the established UPPAAL model checking environment presented in [10].

3. VERIFICATION-BASED DESIGN FLOW

As already stated, most existing design flows for reconfigurable systems are directly based on low level implementation languages like VHDL or Verilog. While it is possible to implement a design in these lower abstraction levels, it is highly appropriate to raise the abstraction level and to represent a system by means of several Models of Computation, which provide a more abstract system view.

One benefit resulting from an introduction of MoCs is a considerable simplification of the system development, as a substantial part of the domain specific features is already included in the MoC definition. It is furthermore possible to exploit domain-specific MoCs instead of a generic MoC or a design language based model. They allow to reuse expertise and knowledge, which has already been aggregated for these specific models. Furthermore, the raised abstraction layer is no longer linked to a specific hardware or software specific description language like VHDL or C, but a computational model that can be refined into both implementation domains (HW or SW) in the ongoing design flow.

Another essential aspect of such a MoC-based methodology is the consideration of for formal methods for system design. Such methods are mathematically-based languages, techniques, and tools for modeling, specification, and verification of complex systems. Compared to standard testing techniques, a formal verification is an exhaustive process that can cover the whole possible system behavior, whereas testing techniques can only explore a limited set of test cases. The main goal of formal verification is therefore the revealing of design errors by proving a relationship between an implemented model and a user specified system behavior [6, 9]. Hence, with the help of formal verification, it is possible to prove whether the implementation satisfies a desired specification.

The verification of reconfigurable hardware systems is still an open research topic. To the best knowledge of the authors, this paper is the first work that puts a special focus on the formal verification of reconfigurable hardware systems. For this purpose, we combine two specialized MoCs as illustrated in Figure 1. The picture shows the proposed design flow for the verification and, to some extent, the implementation of reconfigurable embedded systems. The RecDEVS MoC captures the functionality of reconfigurable hardware systems. It focuses on reconfiguration and provides specific features for the description of such 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 approriate 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.

At this time the RecDEVS model can be transformed to embedded system (ES) or hardware description languages. Taking this path of the design flow as outlined in the right hand part of Figure 1 will then lead to a working design implementation. [12] describes the resulting design process for the DEVS formalism.

3.1 RecDEVS Model of Computation

As already stated, the RecDEVS formalism [11] is aimed as a mathematical and theoretical foundation for the functional and time-related specification of reconfigurable embedded systems. It extends the DEVS [18] formalism from Zeigler et al. towards the formal specification of reconfigurable hardware modules. The DEVS formalism itself is based on timed automata and specifies hierarchically, concurrently executed, formal models. It was originally designed for the simulation of concurrent systems and is well-suited for the formal specification of combined hardware and software systems, i.e., embedded systems, as detailed in the sequel.

3.1.1 Definition

A RecDEVS system specification is a structure

$$\mathrm{N_{Rec}} = \langle \mathrm{X_{ext}}, \mathrm{Y_{ext}}, D, \mathrm{C_{\chi}} \rangle \,. \tag{1}$$

The system consists of multiple interacting atomic components. Each atomic component is described by a tuple

$$\mathrm{C} = (X,\, Y,\, S, s_0, \delta_{\mathrm{int}}, \delta_{\mathrm{ext}}, \delta_{\mathrm{con}}, \lambda, \tau). \tag{2}$$

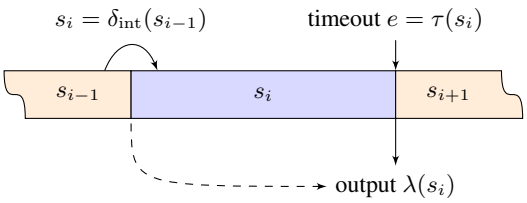

Figure 2: State Transition and Output

S is a non-empty set of states, with $s_0 \in S$ being the initial state of the RecDEVS component. Every state has an associated timeout $\tau : S \to \mathbb{R}$. Atomic components communicate via their input X and output Y. The complete system $\mathrm{N_{Rec}}$ communicates with the external environment via $\mathrm{X_{ext}}$ and $\mathrm{Y_{ext}}$.

RecDEVS defines three different types of state transitions:

Internal transition $\delta_{\text{int}} : S \to S$: After the timeout $\tau(s)$ occurred, the component will do an internal state transition.

External transition $\delta_{\text{ext}} : S \times X \times \mathbb{R} \to S$: Iff an input event occurs on X and no timeout happens, the component will do an external state transition $\delta_{\text{ext}}(s, x, e)$, where $e < \tau(s)$ is the elapsed time since the current state was entered.

Confluent transition $\delta_{\text{con}} : S \times X \times \mathbb{R} \to S$: Iff an input event occurs together with the timeout (i.e. $e = \tau(s)$), the next state will be computed by $\delta_{\text{con}}(s, x, e)$.

Whenever a timeout $\tau(s)$ is hit, the component will also emit a an output, defined by $\lambda(s)$, on the output port Y. This output may either consist of an arbitrary number of messages for other components or of the empty output event $\diamond$. As depicted in Figure 2, an output occurs upon leaving a state, although its value is being determined upon entering a state.

As another difference to the original MoC, RecDEVS incorporates a message based communication scheme. All components share one common communication system that can hold multiple messages at any point in time. Each component instance I can be identified by an unique identifier ID. Thus, all messages consist of tuples $ID \times \mathrm{Data}$, denoting the target and the message body of each message. Messages are transmitted without time delay and trigger an input event at the receiving target component.

3.1.2 *Reconfiguration in RecDEVS*

A system specification $\mathrm{N_{Rec}}$ can be reconfigured by either removing components from the active system or by adding further components to the active system. The list of available components is denoted by D. The list of active components is encapsulated within the system executive $\mathrm{C_\chi}$. This static component is a special instance of an atomic component and handles the configuration of new components as well as the deletion of components. It is statically active in every RecDEVS system and is known to all other components.

Reconfiguration activities in RecDEVS are performed by a set of three dedicated messages: *new(d), del()*, and *confirm()*. The *del()* and *new(d)* messages can be emitted by all active components and are handled by $\mathrm{C_\chi}$. These messages request the deletion of the component id or the creation of a new component of one of the types denoted in D. After the creation a confirmation message *confirm(id)* will be sent back to the requester, so that the newly created component can simply be referenced by means of id from now on.

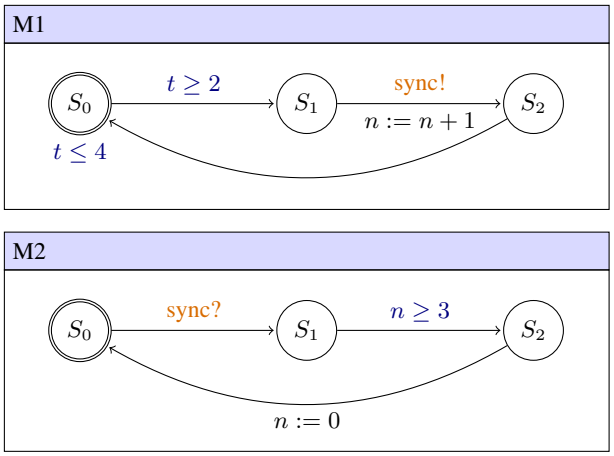

Figure 3: UPPAAL Automata Example

3.2 The UPPAAL Model Checker

UPPAAL [10] 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). Compared to theorem proving approaches it does not require any user-interaction. In addition, the model checker can also produce counterexamples when the implemented model does not satisfy the required specifications.

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 [1]. This model specification language is an extended finite-state machine with 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.

Figure 3 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 `M1.S0`) 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 when the invariant value changes to false at the last point in time. If no invariant is given, then its value is true by default.

All transitions may be attributed by a guarding condition that has to be true for an execution of the corresponding transition. The communication between the automatons M1 and M2 in Figure 3 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 it's transition has to 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.

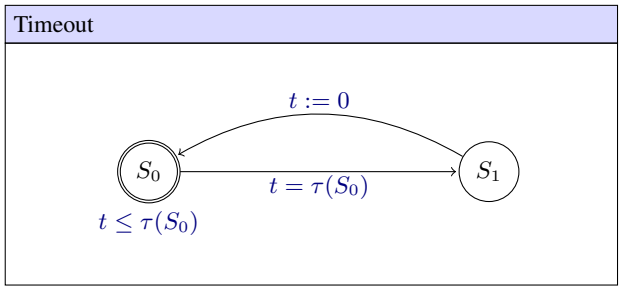

Figure 4: Timeout Realization in UPPAAL

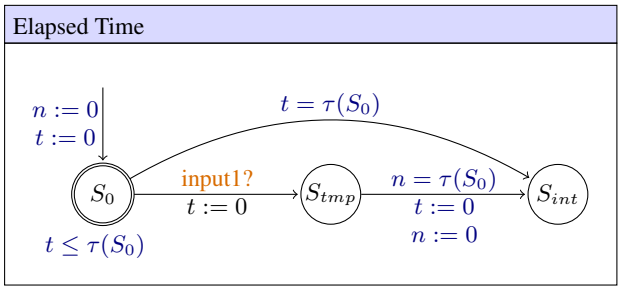

Figure 5: Elapsed Times over Multiple States

In the example of Figure 3 this is demonstrated by means of the shared variable n.

4. RECDEVS TO UPPAAL MODEL TRANSFORMATION

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 a concurrent, state-transition based execution model. As [13] explains, 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.

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 [13], 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 Algorithm 1.

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 4.4 describes how such a behavior can still be represented in UPPAAL.

4.1 Timing Behavior

Every RecDEVS state has an associated timeout function $\tau : S \rightarrow \mathbb{R}$. Figure 4 shows a corresponding UPPAAL model with a timeout $\tau(S_0)$ on state S_0. However, timeouts are not directly supported in UPPAAL. Thus, the timeout for S_0 is realized by a combination of a state invariant $t \leq \tau(S_0)$ for S_0 and a transition guard $t = \tau(S_0)$. 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.

A minor limitation of the model transformation is that UPPAAL only supports natural numbers only and hence can only realize somewhat restricted timeout functions $\tau : S \rightarrow \mathbb{N}$. 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 $\delta_{\text{ext}}(s, x, e)$ and $\delta_{\text{con}}(s, x, e)$ as stated in Equation 2. 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 in Figure 5, where an additional clock n is inserted. Currently, we have not yet implemented an algorithm to detect the described 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.

4.2 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 single state with three leaving transitions is depicted in Figure 6.

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, δ_{int} and δ_{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.

4.3 Inter-Module Communication

While RecDEVS utilizes a message based communication scheme, UPPAAL features dedicated communication channels. It is there-

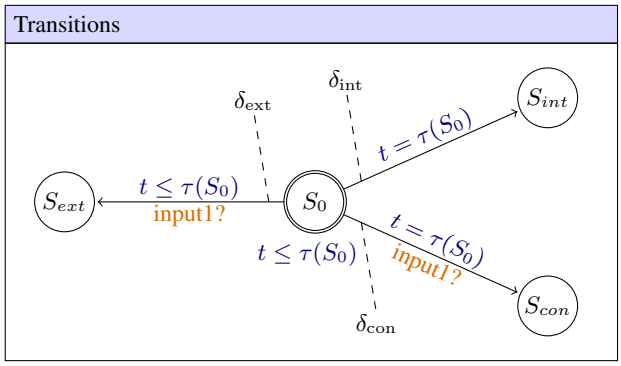

Figure 6: Mapped DEVS Transitions

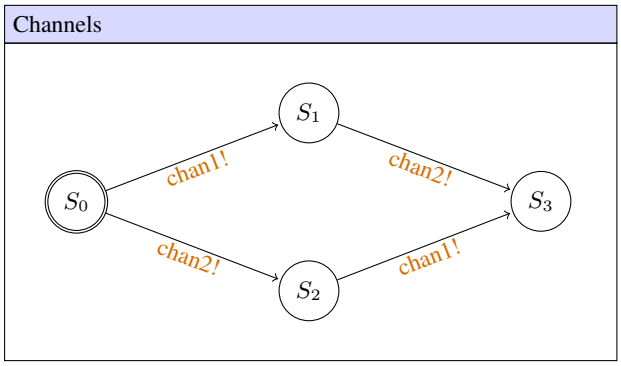

Figure 7: Concurrent Channels

fore 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 Figure 7. This approach can be extended to any number of multiple output events.

4.4 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 4.3.

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 in UPPAAL an equivalent to the system executive C_χ. 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 this special component can simply be removed as demonstrated in Section 5.

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. This approach is also used in other approaches to model reconfigurable systems in other description languages such as SystemC [8].

4.5 Automatic Transformation

Algorithm 1 gives a pseudo code representation of the outlined transformation rules. Please note that this 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, too.

Algorithm 1 *RecDEVS to UPPAAL Transformation*

Input: RecDEVS Specification $\text{N}_{\text{Rec}} = \langle \text{X}_{\text{ext}}, \text{Y}_{\text{ext}}, D, \text{C}_\chi \rangle$.
Output: A corresponding UPPAAL system representation
function Transform(N_{Rec}) **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 $\delta_{\text{int}}(s) = s_{int}$ **do**
        Create Transition t from s to s_{int}
        Update(s,"$x = \tau(s)$","$x := 0$",$\emptyset$,t)
      end for
      for $\delta_{\text{ext}}(s, x_{in}, e) = s_{ext}$ **do**
        Create Transition t from s to s_{ext}
        Update(s,"$x \leq \tau(s)$","$x := 0$","d_input?",t)
      end for
      for $\delta_{\text{con}}(s, x_{in}, e) = s_{con}$ **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

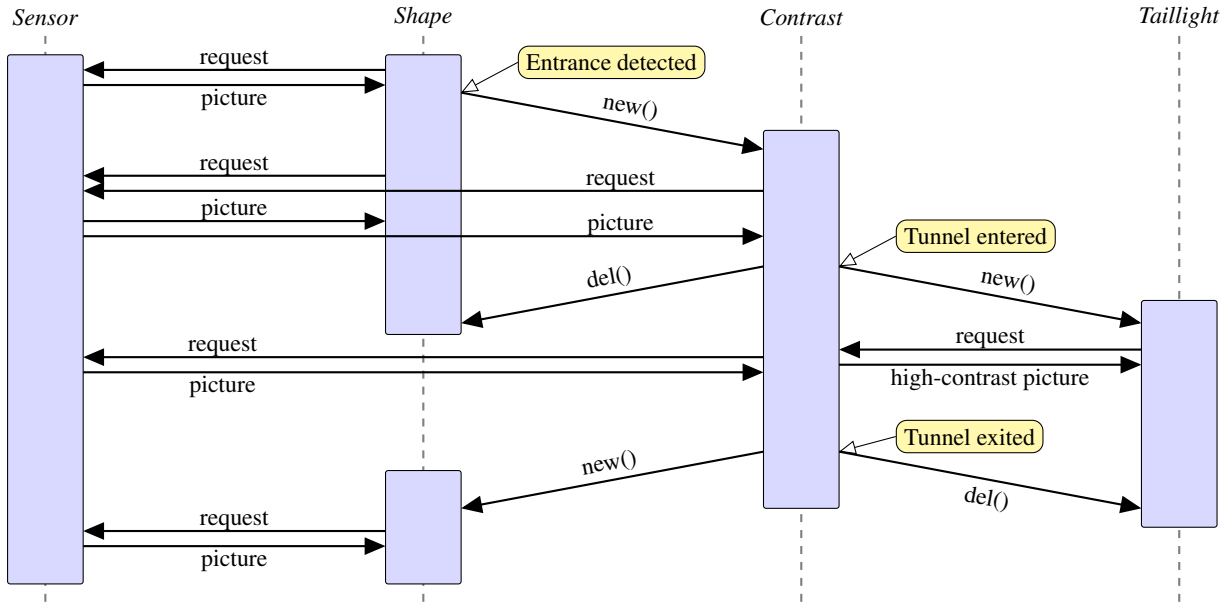

Figure 8: Object Diagram of the AutoVision Example

5. VERIFICATION OF A RECDEVS MODEL

For a detailed analysis of the proposed verification flow, a complex application has been modeled and certain properties of the model have been verified. The application example is based on the AutoVision scenario introduced in [3].

This example taken from the automotive domain consists of several distinct components for vision enhancement and for automated object recognition aimed to a driving assistance scenario. It switches between different components, implementing a shape- or a taillight-based object recognition depending whether the car is on an open road or inside a dark tunnel. Figure 8 gives an overview over the following components and their interaction:

Sensor: Provides pictures to all requesting components. This component persists throughout the whole execution.

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

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

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.

The different situations inside and outside a tunnel trigger the reconfiguration of *Shape, Contrast,* and *Taillight*, respectively. The transformed model consists of four different UPPAAL components, which switch their state values from ‘deleted‘ to ‘active‘ and vice versa. This implies that no *confirm()* messages are required after a *new()* call. A central system executive C_χ is not necessary in this case, because all reconfiguration messages are handled directly by the relevant components.

In the following subsections we outline the most important verification features for reconfigurable systems. We will describe their relevance and their UPPAAL notation. In Table 1 one can find a summary of these statements, a corresponding example for the AutoVision application, and verification results for the tested system specification properties.

5.1 Reachability and Deadlocks

Beside reconfiguration specific properties, there is also a wide variety of important conventional system properties. These properties have to be checked by the presented approach as well.

UPPAAL allows the examination of complex system states, which are composed by multiple RecDEVS components. E.g., the UPPAAL statement “`(Comp1.state1 and Comp2.state2)`” refers to a combination of two atomic component states. All other components may have arbitrary states.

It is possible to examine the reachability of a system state by verifying the UPPAAL specification statement “`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. The existence or absence of deadlocks can also be verified by means of the system specification “`A[] not deadlock`”.

5.2 Communication and Timing

If an arbitrary RecDEVS transition $\delta(S_1) = S_2$ will be taken, it can be verified by the corresponding UPPAAL statement “`E<> P1.S1 imply P1.S2`”. This statement also allows for the verification of the RecDEVS communication being realized by synchronization channels in UPPAAL. As each synchronization channels is linked to a specific transition, we can use this verification statement to prove whether a specific communication message will be sent.

By introducing additional local clocks as explained in Section 4.1, it is also possible to verify timing constraints which cover a longer transition path with multiple state transitions. 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.

Table 1: Set of Verifiable Model Properties of the AutoVision Example

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

5.3 Dynamic Resource Allocation

Regarding reconfiguration we have identified several important generic verification properties, which may be applied to all reconfigurable designs.

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 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 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.

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 4.4 of RecDEVS, this always requires a corresponding message new(B). This message would then require a corresponding transition on which it will be emitted. The activation of such a transition can be tested by verifying the specification statement "E<>(P1.S1 imply P1.S2)". Here, the transition from S1 to S2 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.

6. CONCLUSION

We have presented a systematic approach to transform the Reconfigurable Discrete Event Specified System model RecDEVS into an equivalent representation of the UPPAAL model checking tool.

This allows for the formal verification of reconfigurable hardware systems, which is an essential aspect of a complete design methodology for embedded systems. This approach verifies different properties of the RecDEVS model such as the reachability of certain states or the existence of deadlocks. We have also demonstrated how the verification can be utilized to prove reconfiguration specific properties that do not exist in other verification tools yet. The maximal resource consumption of reconfigurable logic resource is an example for such a new property.

We have also presented certain limitations of the proposed transformation, originating from the different formalisms of both systems, and how some of these limitations can be bypassed for practical applications.

Finally, we have introduced the complex AutoVision example and demonstrated the application of our verification approach. By means of the model checker we have found some unsatisfied properties in our implementation, which have to be corrected in subsequent design steps.

7. REFERENCES

[1] G. Behrmann, A. David, K. Larsen, O. Moller, P. Pettersson, and W. Yi. UPPAAL - Present and Future. *IEEE Conference on Decision and Control*, 2001.

[2] K. Bondalapati and V. K. Prasanna. Reconfigurable Computing Systems. *Proceedings of the IEEE*, 90(7):1201–1217, 2002.

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

[4] S. D. Craven and P. M. Athanas. High-Level Specification of

Runtime Reconfigurable Designs. In T. P. Plaks, editor, *ERSA*, pages 280–283. CSREA Press, 2007.

[5] H. P. Dacharry and N. Giambiasi. A formal verification approach for DEVS. In *SCSC: Proceedings of the 2007 summer computer simulation conference*, pages 312–319, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[6] A. Gupta. *Formal Hardware Verification Methods: A Survey*, volume 1, 1992.

[7] F. Herrera and E. 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.

[8] P.-A. Hsiung, C.-S. Lin, and C.-F. Liao. Perfecto: A SystemC-based Design-Space Exploration Framework for Dynamically Reconfigurable Architectures. *ACM Trans. Reconfigurable Technol. Syst.*, 1(3):1–30, 2008.

[9] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall Modern Semiconductor Design Series. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[10] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer STTT*, 1(1-2):134–152, 1997.

[11] F. Madlener, S. A. Huss, and A. Biedermann. RecDEVS: A Comprehensive Model of Computation for Dynamically Reconfigurable Hardware Systems. In *4th IFAC Workshop on Discrete-Event System Design (DESDes'09)*, Oct. 2009.

[12] F. Madlener, H. G. Molter, and S. A. Huss. SC-DEVS: An efficient SystemC Extension for the DEVS Model of Computation. In *ACM/IEEE Design Automation and Test in Europe (DATE'09)*, Apr. 2009.

[13] H. G. Molter, F. Madlener, and S. 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)*, Oct. 2009.

[14] L. Morihama, V. Pasuello, and G. A. Wainer. Automatic verification of DEVS models. In *Proceedings of SISO Spring Interoperability Workshop*, Orlando, FL. U.S.A, 2002.

[15] H. Patel, S. Shukla, E. Mednick, and R. Nikhil. A rule-based model of computation for SystemC: integrating SystemC and Bluespec for co-design. In *Proc. of the ACM and IEEE Intl. Conf. on Formal Methods and Models for Co-Design, MEMOCODE '06.*, pages 39–48, July 2006.

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

[17] J. Weingart. *Verifikation von DEVS Modellen für rekonfigurierbare Systeme*. Diploma thesis, Dept. of Computer Science, Technische Universität Darmstadt, Germany, Sept. 2009.

[18] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., 2000.