# Integrating UML and UPPAAL for Designing, Specifying and Verifying Component-Based Real-Time Systems

André L. N. Muniz, Aline M. S. Andrade and George Lima
*Programa de Pós-Graduação em Mecatrônica*
*UFBA*
*Salvador, Brasil*
*Email: amuniz@dcc.ufba.br, aline@ufba.br, gmlima@ufba.br*

***Abstract*—A new tool for integrating formal methods, particularly model checking, in the development process of component-based real-time systems specified in UML is proposed. The described tool, TANGRAM (Tool for Analysis of Diagrams), performs automatic translation from UML diagrams into timed automata, which can be verified by the UPPAAL model checker. We focus on the CORBA Component Model (CCM). We demonstrate the overall process of our approach, from system design to verification, using a simple but real application, used in train control systems.**

***Keywords*-components; UML; real-time systems; model checking; UPPAAL;**

## I. Introduction

In this paper we describe a tool, named TANGRAM (Tool for Analysis of Diagrams), designed for modeling, specifying and verifying component-based real-time systems. Due to the increasing complexity and use of such systems this kind of tool is of paramount importance toward design productivity, maintainability and correctness guarantee. It translates specifications written in UML[1] into UPPAAL automata [2]. Model checking can then be applied to verify system correctness so that designers are able to come back to the UML specification without dealing directly with formal languages.

According to Component-Based Development (CBD) [3], software functionality is shared among independent units, called components. One of the considered approaches is the CORBA Component Model (CCM) [4], which offers, among other services, an infrastructure to manage the components during their execution time. Services are provided by a middleware that takes care of low level operation services leaving the application free to deal specifically with their domain functionalities. CIAO (Component-Integrated ACE ORB) [5] is a component middleware that implements a simplified version of CCM and provides services to real-time systems. CIAO is recommended for systems with limited resources such as those embedded in modern automobiles, traffic control or signaling, motion-tracking monitoring or autonomous robots.

This work has been funded by CAPES/CNPq (grant number 475851/2006-4) and FAPESB (APR018/2008).

Since the time at which the system actions take place is an important aspect of correctness, timing characteristics of the execution infrastructure should be taken into account during system design phase. In order to incorporate the characteristics of CCM/CIAO during the system specification, we propose an extension of UML diagrams so that CCM features and CIAO services such as Real-Time Scheduling and Real-Time Event [6] can be described.

The translation process of TANGRAM expects both structural and behavioral model as input, which are represented by UML component and statechart diagrams. In general, the information contained within the component diagram will be derived into UPPAAL global variables and functions, while each statechart will be translated into a timed automaton. Besides application automata, other three pre-defined automata representing scheduling policy and event passing mechanism from CIAO are introduced into the resulting model. We have implemented two scheduling policies, a non-preemptive fixed priority scheduling and preemptive Rate Monotonic [7]. The results are exported to an XML file according to the input format defined by UPPAAL.

The applicability of our approach is demonstrated through the translation and verification of a simple but actual real-time application, which is part of a train control system. We also provide a simulation that indicates that our approach can be scalable to larger systems.

This paper is structured as follows. In Section II, we show how to use some UML diagrams to represent the real-time component model. A simple but real real-time system is defined in Section III and the resulting automata obtained from TANGRAM are shown in Section IV. In Section V, we point out what kind of properties can be verified using our approach. Section VI contains an overview of those research results most related to our work. Our conclusions are drawn in Section VII.

## II. Modeling component-based real-time systems in UML

TANGRAM takes UML component diagrams and statechart diagrams as input for the translation process. Nevertheless, the component diagram is not related to any specific

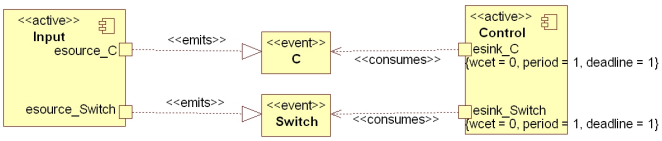


Figure 1. Example of an extended component diagram.

component model, so it lacks some features provided by CCM. Actually, the Object Management Group (OMG) has defined an UML profile for CCM [8], but it only presents the mappings from the definition of a component in isolation and does not cover the composition between components. Due to these characteristics, it was necessary to extend the UML component diagram so that the features of CCM and CIAO could be taken into account in the structural modeling.

### A. Structural model extensions

CCM defines an event passing mechanism between components. As a result, we had to extend the component diagram to represent *event types*, which is a feature not included in this diagram. This was done by adding a class with the stereotype *event* to the diagram (see event *Switch* in Figure 1). In order to distinguish the different types of CCM ports, we extended the associations between ports and interfaces or event types, by adding a corresponding stereotype to them. CCM defines four types of ports: (i) *facets*, which are the provided interfaces; (ii) *receptacles*, which are the required interfaces; (iii) *event sources*, which publish events; and (iv) *event sinks*, which consume events. For example, if the port is an *event sink*, then the stereotype *consumes* is applied (see port *esink_Switch* in Figure 1). All the stereotypes applied to ports are in line with CCM's Interface Definition Language (IDL), which is the language defined by the OMG to declare components.

According to the Real-Time Event Service of CIAO, event sinks may have temporal attributes that will be handled by the Scheduling Service. In UML, we have modeled these temporal properties by defining three tagged values over ports, as shown in Figure 1. Another important feature offered by the Real-Time Event Service is the periodic timeout events. Whenever a component requires a periodic timeout event, it may subscribe to this service provided by the middleware. In order to represent this feature, we created a particular event called *timeout* and introduced the *EventChannel* component to produce it. The rate of the *timeout* event for each component is defined by the *period* tagged value associated to its event sink.

Another feature of CIAO is the active component definition [9]. An active component has its own thread of execution, defined by a callback function named *start*. This function is called by the middleware when the system is initialized. We define a component as active by simply adding the stereotype *active* to it.

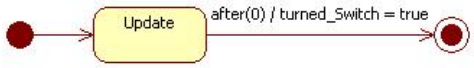


Figure 2. Behavioral modeling of Control's event sink *esink_Switch*.

### B. Behavioral modeling

In the context of component-based systems built on top of CCM and CIAO, behavioral modeling should be aligned with the possible execution points defined by these technologies. According to CCM Implementation Framework [4], operations defined by facets and event sinks have their own body of execution, as opposed to receptacles and event sources, which are only means of accessing other components. Similarly, as we mentioned in the previous section, CIAO's active component also has its own thread of execution, implemented by the *start* function. As a result, our approach considers that statechart diagrams should be modeled for these three points of execution. It is worth mentioning that a *facet* implements one interface, with which several operations can be associated. Hence, each operation must have its own state machine.

Guard conditions and assignment effects can be used to manipulate component attributes. Time trigger is also a very important piece of modeling in terms of timing constraints, because it can be used to specify the duration or execution cost of each state in the diagram. Finally, calling other component operation or dispatching an event can be modeled through effects.

## III. Specification and Design of an Example System

A simple but real application used in train control systems, also known as the "dead-man's vigilance device", is used to demonstrate the applicability of our approach[1]. The main objective of the system is to detect the activity of the operator in controlling the speed and break of the train.

The system contains one main switch associated with inputs A and B. Either one of these inputs is on at a time. Input C is used to deactivate both outputs D and E and must be on when the system is operational. Output E triggers a sound alarm (buzzer) while output D triggers the emergency break of the train.

The system must operate as follows: every T time units, the operator must press/release the switch. If this is not done, the system must activate the buzzer during 4 seconds or until the operator press/release the switch again. If the operator does not respond to the buzzer, it means that he is not controlling the speed of the train anymore. In that case, the system must activate the train emergency break immediatally.

[1]The system specification has been kindly offered by the **AeS** group, which is an embedded control systems specialized company.

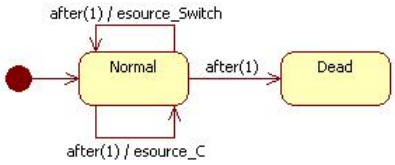


Figure 3. Behavioral modeling of *Input start* function.

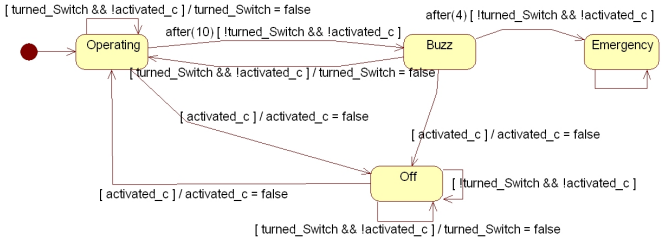


Figure 4. Behavioral modeling of *Control start* function.

### A. Structural modeling

Two active components have been created, which are actually shown by the diagram in Figure 1. One component is defined to represent the interaction between the operator and the device (*Input*), and another to represent the control system (*Control*).

The Input component has two event sources (*esource_C* and *esource_Switch*), which produce the events *C* and *Switch*, respectively. The *Control* component consumes these two events through ports *esink_C* and *esink_Switch*. As can be seen in the model, temporal constraints have been assigned to these ports. Timing constraints have been defined according to the application requirements.

It is important to take into consideration some characteristics of the application during the modeling process. First, it is clear that both *esource_C* and *esource_Switch* are not periodic since they are triggered by the operator. Nonetheless, as we assume that the system time constraints are hard (no deadline can be missed), we take their minimum interarrival time as their periods, as recommended by the real-time community [10]. Also, the ports *esink_C* and *esink_Switch* only update the values of the attributes contained in the *Control* component, and this operation has a very low cost. Therefore, for the sake of simplicity, the associated worst-case execution time (wcet) will be considered negligible.

The *Control* component has also two boolean attributes: *activated_C* and *turned_Switch*.

### B. Behavioral modeling

Considering the behavioral modeling approach previously described, four statechart diagrams have been built to specify the behavior of the system.

The role of the statechart diagram related to *esink_Switch* (Figure 2) is to update the value of the attribute *turned_Switch*. In this case, only one state (*Update*) is needed to model its behavior. The action of updating the attribute is modeled as an effect in the transition that leaves the *Update* state. This transition has a temporal constraint represented by the time trigger *after(0)*, which means that it should be executed immediatelly after the state is entered. As a result, the whole time spent by this event sink is insignificant when compared to the system timing requirements, which are defined in terms of seconds. A similar idea has been used to model the behavior of *esink_C*, therefore it will not be shown here.

The idea behind the state machine related to *Input*'s *start* function (Figure 3) is to represent the interactions of the operator with the system through the switch. The operator can be in either *Normal* or *Dead*. In the former state, the operator can press the switch, deactivate the system (event C) or go to the *Dead* state. In the latter case, no further interaction with the system can be carried out. As can be noticed from the figure, we have constrained the operator's behavior so that he can take only one action at a time. This is done by adding the time trigger *after(1)* on each transition of the state machine.

The state machine associated to the *start* function of *Control* (Figure 4) is initially in the *Operating* state. As long as the operator keeps pressing the switch regularly, the system remains in that state. If the operator activates the input *C*, then the system goes to the *Off* state, until it is activated again. After 10 time units without any action from the operator, the system fires the sound alarm (*Buzz* state) and then waits 4 time units for a response. If nothing happens within this time interval, the emergency breaks of the train are activated and the system goes to the *Emergency* state.

## IV. Translation with TANGRAM

In this section, we show how TANGRAM can be used so that UML diagrams are translated into equivalent timed automata, which can then be verified by UPPAAL model checker. The translation can be divided into two phases. The first one produces both the middleware associated automata and configuration of the global variables. The second phase comprises the translation of each statechart diagram into a timed automaton. We describe the overall translation process rather than unnecessarily getting into its details. Some explanation on UPPAAL are given when the automata of TANGRAM are described.

### A. Middleware automata

According to our approach, there are three automata representing the functionalities of CIAO. The automaton *DispatchingModule*, shown in Figure 5, represents the underline Real-Time Scheduling Service, which is responsible for dispatching each event in the system to its correct client and in the correct priority order. The *EventChannel* automaton is responsible for capturing all the events produced in the

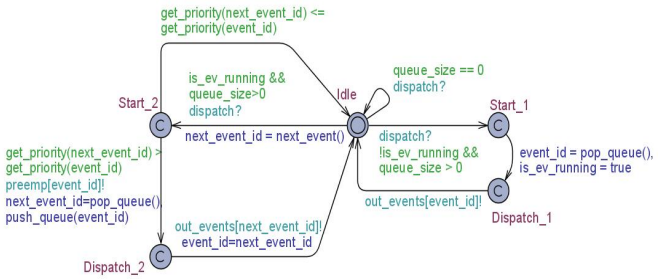


Figure 5. DispatchingModule automaton based on the real-time scheduling service of CIAO.

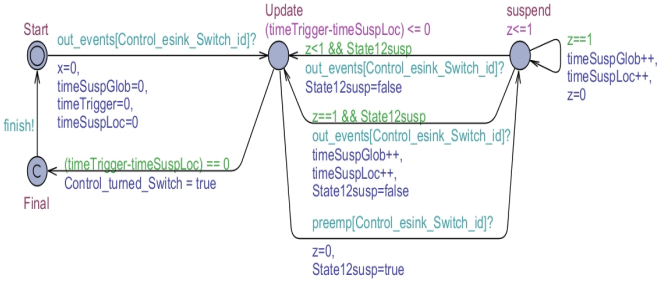


Figure 6. Automaton obtained from the translation of Control's event sink *esink_Switch*.

system, pushing them into a priority queue and warning the *DispatchingModule* about the queue update. The periodic timeout event service is modeled by a *Timer* automaton, which is instantiated for each event sink that consumes a timeout event. However, it has not been generated for the dead man's vigilance device system, because no timeout event was used in this example. Due to lack of space, only the *DispatchingModule* automaton will be explained in this paper.

The automaton in Figure 5 has the following behavior. There are five nodes, namely location in UPPAAL terminology. From the `Idle` location, the automaton waits for a synchronization over the channel `dispatch` (see `dispatch?`). When the dispatching signal arrives, the automaton can take three different edges. The first one leads to the `Start_1` location and it is taken if there is no running task in the system (`!is_ev_running`) and there is some pending event in the queue ($queue\_size > 0$). On the other hand, the second edge leads to the `Start_2` location and it is taken if there is a running task (`is_ev_running`). In this case, it is necessary to check if this running task will be preempted, according to its priority. Therefore the `next_event_id` integer variable is updated by the `next_event()` function, receiving the identification of the next ready event in the queue. The third edge keeps the automaton in the `Idle` location, and it is taken if there are no pending events in the queue.

As can be seen, there is only one possible path from the `Start_1` location. It consists in popping the next event from the queue, using the `pop_queue()` function, and dispatching it through the `out_events` channel, from the `Dispatching_1` location.

There are two possible paths from the `Start_2` location. The first one returns straight to the `Idle` location due to the fact that the running task priority is greater than the next queued event priority. The second path is used to preempt the running event through the `preemp` channel. In this case, the current event is pushed back into the queue by the function `push_queue(event_id)`. After that, the event is dispatched through the `out_events` channel, from the `Dispatching_2` location.

### B. Statechart translation

TANGRAM automatically generates a timed automaton for each statechart diagram. Figure 6 shows the automaton obtained from the translation of the event sink *esink_Switch* state machine (Figure 2). In general, each state in the diagram has a corresponding location in the automaton and each transition has a corresponding edge. The `Start` location is related to the initial pseudostate of the diagram, the `Final` location to the final state and the `Update` location to the *Update* state.

In this case, the `suspend` location is included in order to freeze the automaton execution while it is preempted by a higher priority one. The selection of the higher priority automaton and the preemption itself are carried out by *DispatchingModule*. The time spent in this location is controlled by a clock variable, `z`. Every time unit spent in the `suspend` location, the local integer variable `timeSuspLoc` is incremented. This accounts for the time the automaton stays in the current preemption. Another local integer variable, `timeSuspGlob`, accounts for the duration of all preemptions suffered by the automaton.

The time trigger *after(0)* is translated into a guard and an invariant over a dedicated clock called `timeTrigger`. This clock is declared for each automaton obtained from a statechart diagram. As it can be seen in Figure 6, the time spent in the `Update` location minus the preemption duration cannot be greater than zero (`timeTrigger-timeSuspLoc <= 0`). This means that the automaton cannot stay in that location, unless it is preempted.

The assignment effect *turned_Switch = true* is translated into an equivalent assignment in UPPAAL, `Control_turned_Switch = true`. The main difference is that all translated attributes must receive their owner component name as a prefix, to avoid duplicated identifiers.

The automaton related to the event sink *esink_C* is very similar to previous one, therefore it will not be shown here. The automata of both *Control* and *Input start* functions preserve close similarities with their corresponding statechart diagrams. In other words, there is a one-to-one mapping between automaton locations and diagram states.

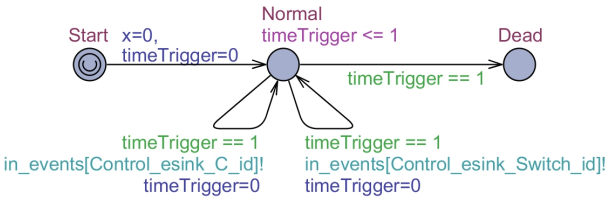


Figure 7. *Input start* function timed automaton.

From a software engineering perspective, this is an important characteristic since it allows the developer to keep track of the component original functionality via both the statechart diagrams and the corresponding automata. The translation of *Input start* diagram is also similar to *Control start* diagram and so its translation will not be shown.

The *Input start* behavior is mainly based on sending events *C* and *Switch* to the *Control* component. The translation of this behavior (Figure 7) must use the channel `in_events` to synchronize with the *EventChannel* automaton. The synchronization over this channel represents the sending of a new event that must be pushed into the priority queue. Time triggers are treated in the same way as for the *esink_Switch* automaton.

## V. Verification of Properties

In UPPAAL one can verify safety, liveness, reachability and deadlock freedom properties. In order to interpret the results generated by the model checking process, it is necessary that the user is familiarized with the simulation environment of UPPAAL and with the name mapping between diagrams and automata generated by the translation. It is unnecessary that the user knows how to specify timed automata in UPPAAL, since most of the counterexample elements can be matched to the original diagrams only by name comparison. In the following we describe two properties specified in TCTL which were verified for our example system automata generated by TANGRAM.

**`A[]`** `Control_start_proc.Emergency` **`imply`** `Input_start_proc.Dead`: This property verifies if for all cases where the *Control start* process is in the `Emergency` location, then the *Input start* process will be in the `Dead` location. This excludes the possibility of having the train emergency break activated while the operator is in a normal condition. This property is satisfied by our model.

`Input_start_proc.Dead` **`-->`** `Control_start_proc.Emergency`: This property is to check whether the `Emergency` location of the *Control start* automaton is reachable when the *Input start* automaton is in the `Dead` location. We wanted to check if the train would not continue running even after the "death" of the operator, which is obviously not a desired scenario. It was pointed out that this scenario may indeed take place. The counterexample showed that this situation may take place when the operator activates the input *C* right before his "death", deactivating the whole vigilance device. Although a simple observation, this kind of verification illustrates well the benefits of integrating formal methods into the system development process, helping the developer in early stages of the system design.

Other properties, not explicitly shown here, have been verified in order to identify possible translation bugs and other application properties. Some of them are related to schedulability analysis, i.e. whether or not the application timing constraints are actually met. Although there are analytical derivations that can be used [10], model checking can point out unschedulable scenarios. This information may well help the designers to take decisions about their designed systems.

The coverage/capacity of our approach has also been tested with another synthetic model composed of 20 components and 20 statechart diagrams. Each pair of state machines represented one periodic task in this model. The test consisted in checking if the system was deadlock-free. We started with 4 components and added a new pair of components after each successful run. The last run finished successfully after consuming 21 minutes and 460 MB of memory. These results indicate that our approach scales well when increasing the size of the system. We also noticed that the impact of introducing middleware functionalities does not jeopardize the verification process.

It is known that the model checking technique has its own state space explosion problems when dealing with bigger models [11]. Therefore, our approach is limited to the model checker capacity of handling the state space.

## VI. Related Work

There has been intensive research on model mapping applied to the design and verification of real-time systems. In this section we summarize those results related to component-based real-time systems.

CADENA [12] is an environment for design and implementation of real-time systems based on the Boeing's Bold Stroke component middleware. Specification models derived from IDL (Interface Definition Language) can be translated into dSpin [13] models to be verified. Middleware functionalities, such as the scheduling policy, have been considered in order to reduce the state space of the generated formal model.

UPPAAL has been considered by some approaches. According to the SaveComp Component Model (SaveCCM) [14], component-based real-time systems properties can be checked. A framework called DREAM (Distributed Real-Time Embedded Analysis Method) [15] has been proposed aiming at non-preemptive scheduling of avionics application based on Bold Stroke. DREAM utilizes a domain-specific modeling language (DSML), which is associated to Bold

Stroke. Their models are translated into timed automata in order to verify schedulability issues using model checking. The resulting automata do not consider the detailed behavior of components, but only their temporal properties.

The approach presented in this paper has some similarities with those described before. First, our model checking process is based on the timed automata formalism. Second, we aim at the verification of functional properties, exploring detailed behavior of components operations. Finally, specific middleware functionalities have been incorporated into the generated model, which improves verification quality. Our results point out that these functionalities do not jeopardize the model state space during the verification process.

However, instead of using domain-specific or non-standard languages, we consider a development process based on UML, a *de facto* specification language. In addition, we work with CIAO, which is a component middleware based on the CCM specification with real-time extensions. Unlike other component models, CCM has been conceived to be interoperable, which means that it is independent of platform and programming language.

## VII. Final remarks

We believe that model transformation approaches like the one presented in this paper provide a suitable design-support tool for software engineers in order to apply formal methods, especially model checking, in the development process of component-based real-time systems.

The translation has been validated using the model checking process itself. Properties were applied to ensure that the generated automata presented the expected behavior defined by the UML diagrams. As a consequence, when a counterexample is found by UPPAAL in a translated specification, then a corresponding behavior leading to the same scenario can be found in the original UML diagrams.

The applicability of our approach was demonstrated using a simple but real application, which is commonly used in train control systems. It is worth mentioning that no further manipulation of timed automata was necessary, avoiding the complexities of dealing directly with formal models.

The next steps of our research are related to introducing more elaborated scheduling policies and more refined configuration of the generated model, for example, defining the granularity of the time unit used in the verification model. Indeed, preemptive scheduling and/or dynamic priority assignment policies may cause state explosion problems that must be dealt with. The results presented here are promising steps toward these goals.

## References


[1] OMG, *UML 2.0 Superstructure Specification*, Object Management Group, 2005. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/05-07-04

[2] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal." in *Formal Methods for the Design of Real-Time Systems*, vol. 3185/2004. Springer Berlin / Heidelberg, 2004, pp. 200–236.

[3] I. Crnkovic, "Component-based approach for embedded systems," in *Proceedings of the 9th Workshop on Component-Oriented Programming*, 2004.

[4] OMG, *CORBA Component Model*, OMG, 2007. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/06-04-01

[5] N. Wang, D. Schmidt, A. Gokhale, B. Natarajan, C. Rodrigues, J. Loyall, and R. Schantz, "Total quality of service provisioning in middleware and applications," *The Journal of Microprocessors and Microsystems*, vol. 2, no. 27, pp. 45–54, 2003.

[6] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time corba event service," in *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 1997, pp. 184–200.

[7] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[8] OMG, *UML Profile for CCM, v 1.0*, OMG, 2005. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/05-07-06

[9] B. Natarajan, D. C. Schmidt, and S. Vinoski, "The corba component model part 4: Implementing components with ccm," *Dr. Dobb's Portal*, 2004. [Online]. Available: http://www.ddj.com/cpp/184403884

[10] J. W. S. Liu, *Real-Time Systems*. Prentice-Hall, 2000.

[11] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: algorithmic verification and debugging," *Commun. ACM*, vol. 52, no. 11, pp. 74–84, 2009.

[12] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An integrated development, analysis, and verification environment for component-based systems," in *Proceedings of the 25th International Conference on Software Engineering*, 2003. [Online]. Available: citeseer.ist.psu.edu/hatcliff01cadena.html

[13] C. Demartini, R. Iosif, and R. Sisto, "dspin: A dynamic extension of spin," in *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. London, UK: Springer-Verlag, 1999, pp. 261–276.

[14] J. Carlson, J. Hakansson, and P. Petterson, "Saveccm: An analysable component model for real-time systems," in *Proceedings of FACS 2005*, 2005.

[15] G. Madl, S. Abdelwahed, and D. C. Schmidt, "Verifying distributed real-time properties of embedded systems via graph transformations and model checking," *Real-Time Systems*, vol. 33, no. 1-3, pp. 77–100, 2006.