Published in the Innovations in Systems and Software Engineering, Dec. 2009. Springer.

André L. N. Muniz · Aline M. S. Andrade · George Lima

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

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. Also, a more complex case study regarding train control systems is described.

Keywords components · UML · real-time systems · model checking · UPPAAL

1 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[12] into UPPAAL automata [1]. 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 at a detailed level.

According to Component-Based Development (CBD) [4], software functionality is shared among independent units, called components. One of the considered approaches is the CORBA Component Model (CCM) [14], 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) [15] 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.

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 Service [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 translated 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 [8]. 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. In addition, our approach has been applied to an automatic sliding doors system, also called Platform Screen Doors (PSD), which is commonly used in

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

Programa de Pós-Graduação em Mecatrônica
Universidade Federal da Bahia
Salvador, Brasil
E-mail: amuniz@dcc.ufba.br, aline@ufba.br, gmlima@ufba.br

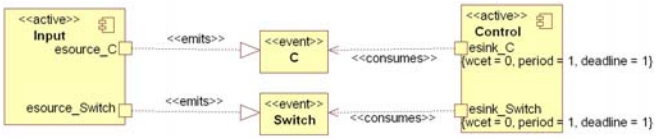

Fig. 1 Example of an extended component diagram.

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

subway stations. We also provide a simulation that indicates that our approach can be scalable to larger systems.

This paper is structured as follows. In Section 2, we show how to use some UML diagrams to represent the real-time component model. A simple but actual real-time system is defined in Section 3 and the resulting automata obtained from TANGRAM are shown in Section 4. In Section 5, we point out what kind of properties can be verified using our approach. A more complex case study is described in Section 6. Section 7 contains an overview of those research results most related to our work. Our conclusions are drawn in Section 8.

2 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 component model, so it lacks some features provided by CCM. Actually, the Object Management Group (OMG) has defined an UML profile for CCM [13], 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.

2.1 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: `wcet` (worst case execution time), `period` and `deadline` (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 [11]. 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.

2.2 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 [14], operations defined by facets and event sinks have their own body of execution, as opposed to receptacles and event sources, which are a means of accessing other components. Similarly, as we mentioned in the previous section, CIAO 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.

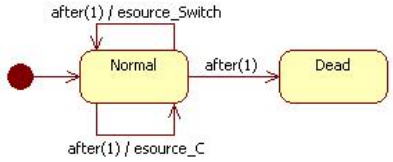

Fig. 3 Behavioral modeling of *Input start* function.

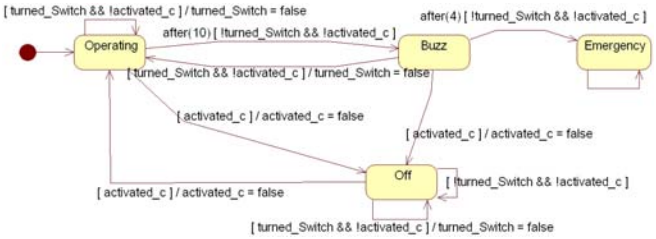

Fig. 4 Behavioral modeling of *Control start* function.

3 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 10 time units, the operator must press/release the switch. If this is not done, the system must activate the buzzer during 4 time units 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 immediately.

3.1 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 [9]. 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*.

3.2 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

[1] The specification of both dead-man's vigilance device and Platform Screen Doors (Section 6) systems have been kindly offered by the **AeS** group, which is an embedded control systems specialized company.

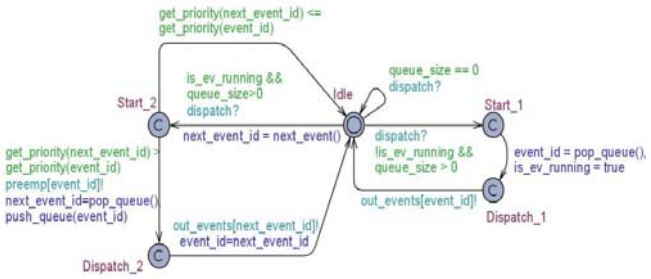

Fig. 5 DispatchingModule automaton based on the real-time scheduling service of CIAO.

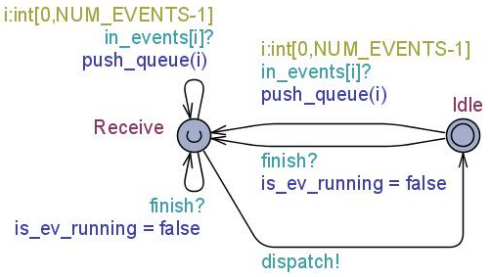

Fig. 6 EventChannel automata based on real-time event service of CIAO.

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.

4 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.

4.1 Global variables and middleware automata

Components can have attributes which are shared among its state machines. As a result, each attribute declared in the component diagram is translated into a global variable in UPPAAL. TANGRAM supports both boolean and integer types, which are the supported data types by UPPAAL. The translated variables are identified by the concatenation between the component name and attribute identifier.

Other variables are generated from the translation of facets and event sinks. A facet represents a set of operations defined by an interface. Each operation is translated into a channel variable in UPPAAL. This channel is used to activate and terminate the execution of the automaton that represents that operation. An event sink represents an entry point for system events controlled by the middleware. Therefore, the middleware must be capable of identifying each event sink and the type of event it consumes. As a result, each event sink is translated into an integer constant that is used by the middleware to identify that event sink during system execution. Similarly, event sink temporal properties are translated into integer constants, which will be handled by 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 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.

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

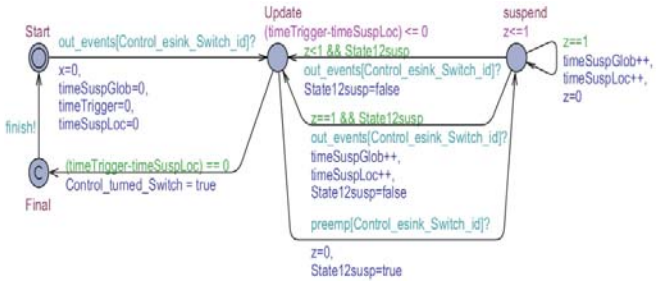

Fig. 7 Automaton obtained from the translation of Control's event sink *esink_Switch*.

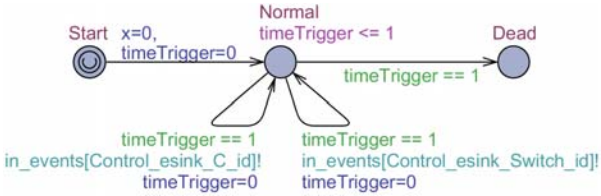

Fig. 8 *Input start* function timed automaton.

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.

The *EventChannel* automaton (Figure 6) has two locations: `Idle` and `Receive`. The `Receive` location can be reached from the `Idle` location by two edges. The first one is fired when an event is published in the system, through the `in_events[i]?` synchronization. The received event is pushed into the queue by the `push_queue(i)` function. The second edge is fired when some running task finishes its execution, synchronizing through `finish?`. The automaton stays in that location until all the events produced in the system at that moment are intercepted. Finally, the *EventChannel* returns to the `Idle` location, synchronizing with the *DispatchingModule* on `dispatch!`. This indicates that the event queue has been updated, then the *DispatchingModule* must decide which task should be actually running.

The *Timer* automaton (not graphically shown here) counts time according to a given period, dispatching a timeout event through a synchronization with the *EventChannel* automaton. It has two locations, `Counting` and `Timeout`. The time spent in the `Counting` location is incremented by the clock variable, x, which is constrained by the invariant $x <= period$. Every time a timeout occurs, clock x is reset so that the required timeout event is dispatched every *period* time units.

4.2 Statechart translation

TANGRAM automatically generates a timed automaton for each statechart diagram. Figure 7 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.

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 state. Another local integer variable, `timeSuspGlob`, accounts for the interference due to 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 7, the time spent in the `Update` location minus the interference due to preemption 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. 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 8) 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.

5 Example system verification

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 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 [9], model checking can point out unschedulable scenarios. This information may well help the designers to take decisions about their designed systems.

6 Case study

A case study regarding train control systems has been carried out. The main objectives of this study were: (i) to evaluate more precisely the benefits of using TANGRAM in a real system project; (ii) to estimate the impact of middleware functionalities over system's state space; and (iii) to identify possible corrections or improvements for future versions of TANGRAM. In this section we highlight the main results obtained from this case study and the experience gained from using TANGRAM in a more complex system.

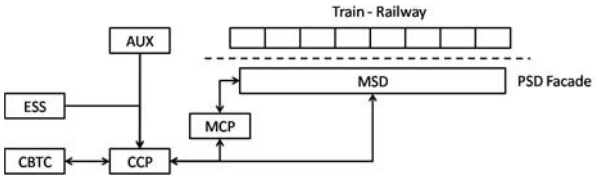

Fig. 9 PSD system physical structure.

6.1 System description

The case study concerns an automatic sliding doors system, also called Platform Screen Doors (PSD), which is used in subway stations. These sliding doors form a barrier between the train and the platform, enhancing passengers safety and preventing unauthorized people from accessing subway tunnels. The PSD system must be aligned and synchronized with the train's doors.

The PSD physical structure (Figure 9) is mainly composed of motorized sliding doors (MSD), a central control panel (CCP), manual control panels (MCP), an auxiliary module for automatic position detection (AUX) and an energy supply system (ESS). The CCP holds the PSD main control system and it is connected to the Communication Based Train Control (CBTC), which is the subway main control system. The MCP allows local operators to send commands to the system. The AUX module detects the train position in the platform and sends this information to the CCP. Finally, the ESS sends information about the PSD power supply to the CCP.

The system must operate automatically, through the commands received from the CBTC. Alternatively, in case of communication failure with CBTC, it can still operate automatically through signals received from the AUX module. In the case of both CBTC and AUX failure, the system doors must be opened or closed remotely through the MCP. As long as the MCP is activated by the operator, any signals received from CBTC or AUX are ignored.

Other system requirements include the possibility of sending a command from the MCP to the CBTC in order to inhibit the departure of the train from the platform, if necessary. Also, the system must detect if the normal power supply is down and the emergency energy supply system is active. In that case, doors must be opened or closed alternately, avoiding peaks of energy consumption.

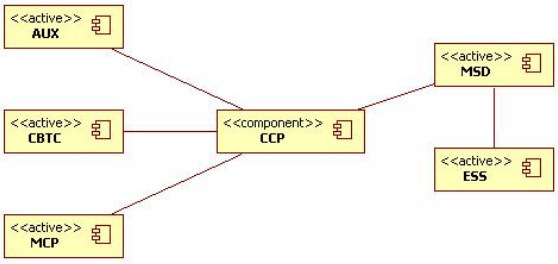

Fig. 10 PSD component diagram abstraction

6.2 System modeling

In this case study we focused on modeling the system that runs in the CCP. As a result, we have modeled one main component to represent the CCP and other five components related to the modules that interact with it. Figure 10 shows an abstraction of the component diagram built for this system. Ports, temporal properties and details about communication between components have been omitted in order to reduce image size. The whole model contains 22 events.

It is worth mentioning that only one component has been created to represent the interaction with all the system ports, which is the MSD component (Figure 10). Otherwise, it would be necessary to model the behavior of each port individually, which would end up in a state space explosion during system verification. In the context of this case study, it is enough to model a behavior like "*there is an open port*" or "*all ports are open*", rather than a detailed description, like "*port1 is open, port2 is open, ..., portn is open*".

The behavioral model consists in 28 statechart diagrams. Most of them represent system functionalities, but there are some that have been created to simulate the interactions from environment. For example, the active behavior of CBTC component specifies the complete cycle of a train in the platform, from its arrival to its departure. Since this behavior is specified in CBTC, it is executed only if the system is in the automatic operation mode. The system operating mode is set up through attributes declared in the CCP component.

6.3 Verification and results

The verification process was divided into 12 different cases, which varied the following characteristics: (i) activation of CBTC or AUX operating mode; (ii) activation of the remote operation through the MCP; (iii) activation of the emergency energy supply system; and (iv) possibility of having doors being opened locally, through special keys. This separation provided a more flexible verification process, according to which cases could be classified from simple to complex scenarios. As a result, it helped avoiding state space explosion in early stages of verification.

In total, sixteen distinct properties have been specified in TCTL. For each of the 12 considered cases, there was a subset of properties involving the characteristics of that case.

In general, the case study objectives have been achieved satisfactorily. One of the main benefits observed in this case study is the possibility of checking the internal behavior of components, as opposed to similar approaches [10].

As for middleware automata impact, the results showed that they do not jeopardize the verification process. The verification results showed that most properties could be verified successfully, while a minor set ran into state space explosion. We identified that the cases where the state space explosion occurred were characterized by a high level of non-determinism. Specially when dealing with the MCP active behavior, which specifies all possible interactions of the operator with that panel. A way of dealing with this issue was to consider a more deterministic scenario, where complex properties could be verified in the TANGRAM generated models.

During the specification and verification of the system, some improvements in TANGRAM were identified. For example, it is interesting to allow the user to to select one or more states so that reachability properties can be automatically generated by TANGRAM without the need of dealing directly with TCTL formulas.

7 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 [7] is an environment for the 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 [5] 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 [2], component-based real-time systems properties can be checked. A framework called DREAM (Distributed Real-Time Embedded Analysis Method) [10] 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.

8 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 both simple and more complex applications, which are related to train control systems. Results pointed out the benefits of using TANGRAM for verifying detailed behavior of components.

It is known that the model checking technique has its own state space explosion problems when dealing with bigger models [3]. Therefore, our approach is limited to the model checker capacity of handling the state space. Nonetheless, by using specific characteristics of the application to be verified, one can circunvent the state explosion problem.

Future research goals include 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. Behrmann G, David A, Larsen KG (2004) A tutorial on uppaal. In: Formal Methods for the Design of Real-Time Systems, Springer Berlin / Heidelberg, vol 3185/2004, pp 200–236, DOI 10.1007/b110123
2. Carlson J, Hakansson J, Petterson P (2005) Saveccm: An analysable component model for real-time systems. In: Proceedings of FACS 2005
3. Clarke EM, Emerson EA, Sifakis J (2009) Model checking: algorithmic verification and debugging. Commun ACM 52(11):74–84, DOI http://doi.acm.org/10.1145/1592761.1592781
4. Crnkovic I (2004) Component-based approach for embedded systems. In: Proceedings of the 9th Workshop on Component-Oriented Programming
5. Demartini C, Iosif R, Sisto R (1999) 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, Springer-Verlag, London, UK, pp 261–276
6. Harrison TH, Levine DL, Schmidt DC (1997) 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, ACM, New York, NY, USA, pp 184–200, DOI http://doi.acm.org/10.1145/263698.263734
7. Hatcliff J, Deng W, Dwyer M, Jung G, Prasad V (2003) Cadena: An integrated development, analysis, and verification environment for component-based systems. In: Proceedings of the 25th International Conference on Software Engineering, URL citeseer.ist.psu.edu/hatcliff01cadena.html
8. Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM 20(1):46–61, DOI http://doi.acm.org/10.1145/321738.321743
9. Liu JWS (2000) Real-Time Systems. Prentice-Hall
10. Madl G, Abdelwahed S, Schmidt DC (2006) Verifying distributed real-time properties of embedded systems via graph transformations and model checking. Real-Time Systems 33(1-3):77–100
11. Natarajan B, Schmidt DC, Vinoski S (2004) The corba component model part 4: Implementing components with ccm. Dr Dobb's Portal URL http://www.ddj.com/cpp/184403884
12. OMG (2005) UML 2.0 Superstructure Specification. Object Management Group, URL http://www.omg.org/cgi-bin/doc?formal/05-07-04
13. OMG (2005) UML Profile for CCM, v 1.0. OMG, URL http://www.omg.org/cgi-bin/doc?formal/05-07-06
14. OMG (2007) CORBA Component Model. OMG, URL http://www.omg.org/cgi-bin/doc?formal/06-04-01
15. Wang N, Schmidt D, Gokhale A, Natarajan B, Rodrigues C, Loyall J, Schantz R (2003) Total quality of service provisioning in middleware and applications. The Journal of Microprocessors and Microsystems 2(27):45–54