Automatic Verification of Component-Based Real-Time CORBA Applications*

Gabor Madl Sherif Abdelwahed Gabor Karsai
{gabe,sherif,gabor}@isis.vanderbilt.edu

Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN, 37205

Abstract

Distributed real-time embedded (DRE) systems often need to satisfy various time, resource and fault-tolerance constraints. To manage the complexity of scheduling these systems many methods use Rate Monotonic Scheduling assuming a time-triggered architecture. This paper presents a method that captures the reactive behavior of complex time- and event-driven systems, can provide simulation runs and can provide exact characterization of timed properties of component-based DRE applications that use the publisher/subscriber communication pattern. We demonstrate our approach on real-time CORBA avionics applications.

1. Introduction

Computing systems are increasingly distributed, real-time and embedded (DRE) and must operate under highly unpredictable conditions. Component-based middleware provides a means to manage the complexity of designing these systems. To deal with the complexity of analyzing DRE systems we need to raise the abstraction level from the implementation to system models that capture crucial concepts of the system such as structure, behavior, environment and the properties the system must satisfy.

In this paper we present a solution that captures the reactive behavior of event-driven systems and can automatically verify quantitative dense time properties. Our approach is not limited to periodic systems and can be used with any component-based event-driven system that uses the *publisher/subscriber* communication pattern.

* This work is sponsored in part by NSF ITR project "Foundations of Hybrid and Embedded Software Systems"

The Boeing Bold Stroke DRE architecture [30] is used at Boeing to develop mission-critical avionics applications that control weapons systems on a common platform. It is built on the real-time CORBA open middleware standard and has been successfully deployed in several military systems. We chose this platform to demonstrate that our approach is feasible for real-life systems. We use the UPPAAL verification tool [28] for the analysis. *We map the ESML application models to the* UPPAAL *timed automata* [24] *using graph transformations implemented by the Graph Rewriting And Transformation* (GREAT) *language* [1].

This paper is organized as follows: Section 2 explains the modeling language of the component-based applications, Section 3 explains the Boeing Bold Stroke architecture, Section 4 gives a brief overview of our approach, Section 5 gives a formal description of the timed automata and the scheduling problem, Section 6 illustrates the key concepts that we have used in modeling the scheduling, Section 7 demonstrates the verification of an example application and Section 8 discusses the related work on the field. We end by drawing conclusions and discussing future directions.

2. The Embedded Systems Modeling Language (ESML)

ESML [22] is a modeling language for embedded systems that tries to overcome a few shortcomings of UML [21], such as the lack of a *component model, interaction modeling, component and system configuration*, etc. ESML models are automatically transformed into the *Analysis Interchange Format (AIF)* that is used to exchange models between tools of different vendors.

ESML is based on Real-Time Event Channel technology defined in the CORBA [26] specification, implemented in TAO [17], and also related to the CORBA Component Model [27]. In this model, components are

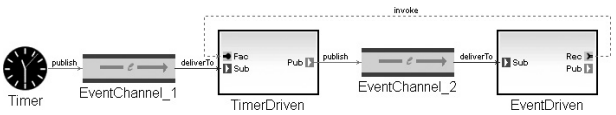

Figure 1. Periodic event-driven control push - data pull processing

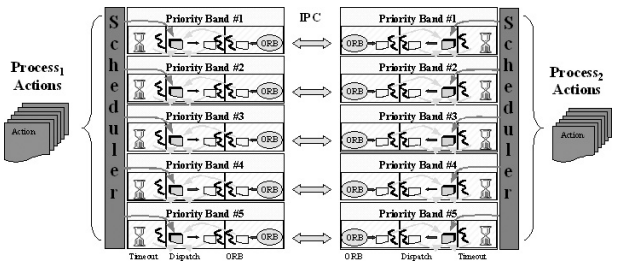

Figure 2. The Boeing Bold Stroke execution framework

complex objects that encapsulate multiple instances of different classes. The interaction between instances of different components is modeled with ports. Ports are interfaces for reaching classes inside the components.

The method invocations follow the traditional synchronous blocking *call-return* semantics, where one component executes a method invocation on another component. The event propagation mechanism follows a non-blocking asynchronous *publisher/subscriber* semantics supported by event channels. When the publisher pushes an event to the event channel the subscribed components will be notified.

To allow the modeling of time-driven tasks, components can receive notifications from special predefined components, called Timers. Timers publish events with a constant period. These events periodically initiate the processing of data read from physical devices, but the processing of the data is usually not synchronized to clocks but triggered by events.

Figure 1 illustrates the periodic event-driven model used in ESML. The `TimerDriven` component is subscribed to the `Timer` and receives notifications peridocially. This will trigger the execution of an action encapsulated in the component. This action will publish events to the `EventDriven` component. When the `EventDriven` component is notified about the availability of data it will issue a remote method call on its receptacle to the facet of the `TimerDriven` component to retrieve that data. This approach separates the flow of control from the flow of data and allows fine-grain scheduling and the incorporation of quality of service (QoS) properties that allow fine-tuned configuration of the system for optimal performance [17].

3. The Boeing Bold Stroke architecture

The Boeing Bold Stroke architecture is a component-based execution framework built on top of the ACE/TAO real-time CORBA implementation [29] and uses the *publisher/subscriber* communication pattern. It uses a proprietary component model called PRISM [30], which emulates the CORBA Component Model (CCM) [27]. While CCM allows components to be dynamically created and connected, PRISM follows a typical practice in safety/mission-critical systems and employs a static component allocation and configuration policy by creating and connecting components only in the system initialization phase. Stateful components can support dynamic reconfiguration by changing behavior based on the system mode settings.

Bold Stroke is event-driven, although driven by Timers. Figure 2 shows the physical model of the Bold Stroke execution framework. Components contain actions that are the smallest units of processing. Bold Stroke uses priority-based scheduling, *actions that have the same priorities are scheduled non-preemptively* in a *priority band* (sometimes referred to as *rate group*) based on their sub-priorities and *preemptive scheduling is used between priority bands.* An action has an assigned priority and sub-priority *(importance)* value for every event channel it is subscribed to. If two actions have the same sub-priority they will be ordered or scheduled non-deterministically according to the configuration. Every action has a measured *worst-case execution time* in the given scenario in which it is used. Actions can be initiated by two ways; method invocations and event propagations. The scheduling strategy is also configurable and uses *Rate Monotonic Scheduling* by default.

A priority band is implemented by three threads; the *dispatcher (work) thread* which executes all the actions initiated by event propagations, the *interval timeout thread* which simply pushes timeout events at predefined intervals, and the *ORB thread* which continually processes inputs from the Object Request Broker (ORB), executing actions initiated by method invocations. This is the implementation of the Thread Pool policy for multi-threading in which a fixed number of threads are generated in the server at the initialization phase to service all incoming requests. This approach offers scalable applications with low latency

times for small duration requests [10].

Facet-initiated actions (that were initiated by a remote method invocation) inherit the Quality of Service execution semantics from the invoking component and do not interact with the runtime scheduler therefore we do not distinguish them from the invoking action in the scheduling perspective. *The smallest unit of scheduling is an event-initiated action together with all the remote method calls it can invoke.* Since facet-initiated actions can also call other actions using remote method calls the complete call chain is an acyclic graph with the event-initated action as root element. *We call this smallest unit of scheduling an invocation unit.*

An executing action may initiate actions on other priority bands, otherwise known as cross rate actions. All processing inside a priority band needs to finish within the fixed execution period of the Timer assigned to the band. This periodicity divides processing into frames. A priority band failing to complete outputs prior to the start of the next frame is said to be in a *frame overrun* condition, meaning that the band did not meet its completion deadline or frame time.

4. Approach

The presence of Timers in the Bold Stroke architecture is an attempt to increase the analyzability of the system by turning it into a time-triggered architecture. This effort is also reflected in the terminology *(rate group, Rate Monotonic Scheduling)*. Many methods assume a time-triggered architecture and try to analyze the scheduling of the Bold Stroke architecture by performing Rate Monotonic Analysis. This approach is widely accepted for fixed priority scheduling.

The Boeing Bold Stroke architecture is driven by Timers, however the system itself is event-driven. Assuming a time-triggered architecture may introduce anomalies in some cases. The main reason behind this is the reactive behavior of the system; certain actions (e.g. identifying targets) will be invoked when external events - coming from the environment non-deterministically as sensor values - are triggered. Actions are not invoked by Timers but by other actions if certain constraints are satisfied, and since the event flow can change the invocations might be aperiodic. Even if we assume that actions have constant execution times an action can publish an event anytime between the execution time and the deadline, since it can wait for other executing processes and the initiation of processes depends on external (non-deterministic) events. The semantics for conditional event triggering and the propagation of missed deadlines is also inexpressible by this approach. This makes it hard to identify vulnerable points in the system, because the impact of a single component failure is undetectable.

In this paper we present a solution that captures the event-driven nature of the Boeing Bold Stroke system and can be used to verify timed properties of aperiodic systems. We show that timed automata [2] as a computational model can describe asynchronous event passing as well as time constraints. It has the necessary tool support [7] [28] and does not need to be extended to handle quantitative dense time features like SPIN [31]. Timed automata has a nice graphical representation that makes the graph transformations suitable. Several model-checking tools use automata theory - usually with some extensions - as a computational model. A few examples are HYTECH [19], KRONOS [7] and UPPAAL [28]. We chose the UPPAAL model-checker tool which is widely used [13] [16] [12] for schedulability analysis and model checking. *We map the ESML application models to the* UPPAAL *timed automata* [24] *using graph transformations and prove system properties by checking the generated timed automata.* The graph transformation has been implemented in the Graph Rewriting and Transformation (GREAT) language [1]. The GREAT tool uses the Generic Modeling Environment (GME) [25] and allows users to specify graph transformations in a graphical form.

5. Problem Description

The system under consideration in this paper consists of a set of dynamics tasks (invocation units). Each task is attributed with its worst case execution time (`WCET`), deadline (`DL`), and priority (`PID`) specification. With respect to timing analysis, computation tasks can be represented by a generic timed automaton model. The task mode is composed of four states: Idle, Ready, Executing, and Timeout. Initially tasks start at an Idle state and are triggered (Idle → Ready) by events which may be received from other tasks or at regulars intervals from a Timer. If the task is scheduled for execution - according to the given scheduling policy - the transition (Ready → Executing) will be triggered. After execution, the task may also initiate other tasks by sending the corresponding event[1]. The task will move to a blocking Timeout state if the deadline is exceeded.

Tasks are executed based on a given scheduling policy which determines at a given time the eligibility of a "ready" task to proceed for execution. In other words, the scheduling policy is responsible for triggering the transition (Ready → Executing). When this transition

1 In general, this may occur during the execution state. However, for non-preemptive scheduling the event would not have any effect until the task terminates

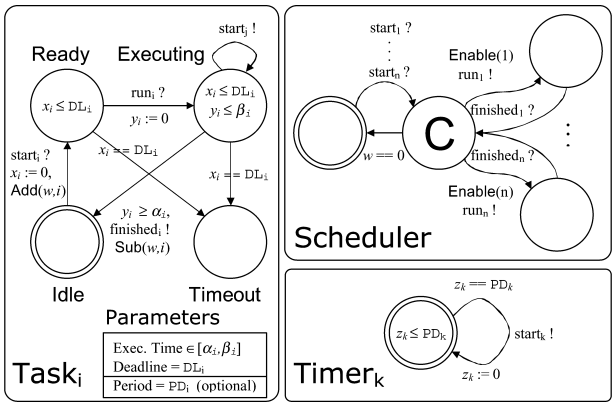

Figure 3. Generic timed automata model for the Task, Scheduler and Timer

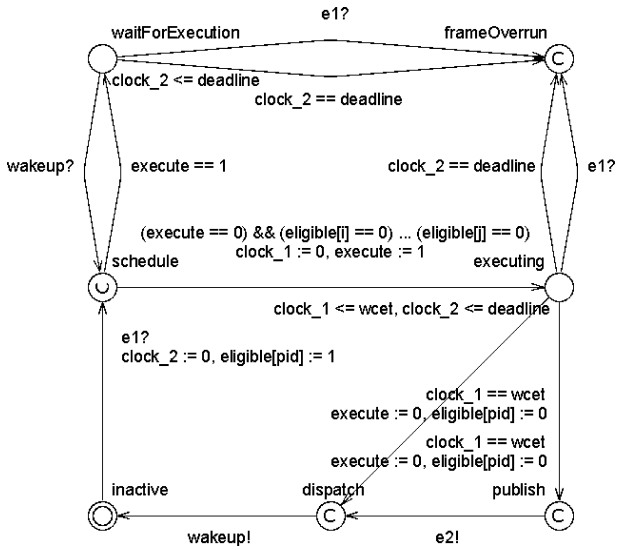

Figure 4. Generic model of an invocation unit

is enabled, the automaton has to take this transition with no time delay. For static scheduling, the policy can be represented by a vector that indicates the eligibility of tasks for execution. A task can move to Executing from Ready only if no higher priority task can make a similar transition at the same time.

The above task model and scheduling policy can be represented as timed automata. A timed automaton is a state machine equipped with clock and data (integer) variables, with precise formal definition [2] [24], tool support [7] [28] and is widely used [4] [11] [3] [13] [16] [12] for schedulability analysis and model checking. Since the focus of this paper is the application of timed automata we refer the reader to these sources for a more formal definition of the timed automata. Transitions in a timed automaton can have guards and reset operations on clock and data variables. Transitions are enabled if the corresponding guard evaluates to `True`. An enabled transition can execute instantaneously while resetting certain variables to new values according to the underlying reset assignments. States in timed automata can be associated with an invariant that determines the validity of clock assignment in the state. The system can be in a given state only if the underlying invariant is `True`.

Figure 3 shows a generic timed automata model for a set of time- and event-driven tasks and the underlying scheduler based on the above description. In this figure a task can be initiated by another task or a timer (or both).

The scheduler will move to a *committed* state if any of the tasks will be eligible for execution. Time cannot pass in this location: the scheduler must take an enabled transition with no time delay. Transitions will be enabled based on the scheduling policy. When taking the transition it will send out an event scheduling a task for execution from a set of ready tasks. The scheduler moves to the wait (initial) state if no task is ready for execution. The scheduling policy is encoded in three functions: $\mathsf{Add}(w,i)$ which increases the current priority level when task_i becomes ready, $\mathsf{Sub}(w,i)$ which decreases the current priority level when task_i becomes ready, and $\mathsf{Enable}(w,i)$ which evaluates to `True` if the ith task is eligible for execution. For example, in a simple index based priority scheme, $\mathsf{Add}(w,i) = w + 2^{i-1}$, $\mathsf{Sub}(w,i) = w - 2^{i-1}$, and $\mathsf{Enable}(w,i) = 2^{i-1} \leq w < 2^i$. Other scheduling schemes can be established by defining appropriate formulas for the above three functions.

In this paper we consider the problem of deciding the schedulability of a given set of tasks with both event- and time-driven interactions. The timed automata formulation of the problem in effect translates the schedulability problem into a reachability problem in which the set of tasks are schedulable if the Timeout state is not reachable in any of the task timed automata, in other words all tasks are completed before their respective deadlines.

6. Generic modeling of the event-flow

Figure 4 shows the generic model of an invocation unit. This model is the extension of the generic timed automaton model shown on Figure 3. Since the Uppaal timed automata semantics is unambigous (the automata executes on the Uppaal virtual machine) we have formally defined our model by giving Figure 4

and using the UPPAAL timed automata execution semantics [24]. In the following we give an informal description of the model to explain how the key concepts, such as *concurrency, asynchronous thread invocation, inter-process communication (IPC)* and *non-preemptive scheduling* are represented in the model.

The inactive location corresponds to the Idle location of the generic automaton, the frameOverrun location corresponds to the Timeout location. The Ready location of the generic timed automaton model is represented by two states (schedule and waitForExecution) in order to express that the enabled transition to the Executing location has to be taken when the transition is enabled. To model the event passing we modeled the Executing location by three locations (executing, publish and dispatch). This timed automaton has two clocks (`clock_1` and `clock_2`) working at the same rate. Attributes can be local (`pid, deadline, wcet`) and global (`execute`).

Concurrency: This timed automaton type may have multiple instances. These instances can communicate with each other using *synchronizations* or *shared global variables*. The `execute` variable is used to restrict that only one action can execute in the priority band at any given time. It acts as a global flag that is set everytime an action starts executing and reset when it finishes the execution.

Asynchronous thread invocation/IPC: When a publisher pushes an event the subscribed components will be eligible for execution. We model this by synchronizing the publish → dispatch transition of the publisher with the inactive → schedule transition of the subscribed components.

Non-preemptive scheduling: The priority-based scheduling is encoded in the guard condition of the schedule → executing transition. *Urgent* locations (schedule) are simple constructs in UPPAAL to express time constraints. When a location is urgent, time cannot pass in that location. If no transitions are enabled at that time the system is deadlocked. This is also true for committed locations (publish, dispatch, frameOverrun), but their outgoing transitions have to be taken first. In other words, transitions belonging to committed locations have preference, while this is not the case for urgent locations. In this particular example we can use these constructs to describe the fact that we want invocation units to publish before any rescheduling happens. We can also formalize clock constraints *(invariance)* for the locations. These are shown next to the locations on Figure 4. When the constraint is violated a transition has to occur otherwise the timed automaton is deadlocked. In the schedule state every timed atomaton checks whether higher sub-priority invocation units are eligible for execution, if yes they move in to the waitForExecution state. If there are multiple highest sub-priority invocation units one will be chosen non-deterministically. Whenever an invocation unit finishes the execution it broadcasts an event (wakeup) to all the timed automata. This will force them to check the guard conditions again.

7. Case study: Timed analysis of a Bold Stroke application

In this section we show a case study on how we use the modeling concepts desribed in Section 6 to verify a Bold Stroke application shown on Figure 5. This application is driven by a single Timer therefore it corresponds to a single rate group/priority band. Each component contains exactly one event-initiated action therefore when we refer to the scheduling of the components we mean the scheduling of the invocation units (the event-initiated action together with the receptacle calls).

The `INS` and `GPS` components are both subscribed to the same event channel. When the Timer pushes an event both components will be notified. These two components correspond to two time-driven invocation units. Since both components receive the same events, they will be eligible for execution at the same time. The scheduler will choose one component according to the scheduling algorithm while the other must wait until the first finishes execution.

The `AIRFRAME` component is subscribed to both time-driven components. The semantics of handling the events is configurable. If we assume `AND` semantics then the component has to wait until all events have ar-

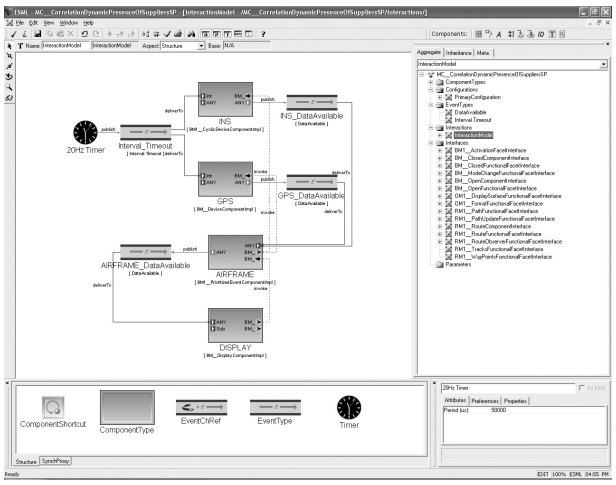

Figure 5. Bold Stroke application model

rived for which the component is subscribed. If we assume OR semantics, any event can trigger the component's execution. In this particular example OR semantics is assumed, because the AIRFRAME component updates its state if any of the INS or GPS components have new data. When the AIRFRAME component is executed, it calls back to the INS and GPS components using a remote method call and publishes an event to the DISPLAY component after it has made the necessary computations. The DISPLAY component will call back into the AIRFRAME component during its execution - in order to display the new data.

7.1. Decomposition rules for the execution paths

We claim that the non-preemptive scheduling of BoldStroke applications can be verified without explicitly modeling the buffering of the event channel. In order to show this, we have to check the properties of the call chains. We already defined the call chains for remote method calls when we introduced the concept of the invocation unit. In that context, the call chain is an acyclic graph with the event-initated action as root element. We can define call chains for the event-flow as well. In a single priority band, the Timer will publish events to the subscribed components. These components will also publish events etc. There are no cycles in the event flow, otherwise we would end up in a loop that can possibly execute forever. The call chain of the event flow in a priority band can be represented as a directed graph forest with the Timers as roots of the trees. The vertices correspond to invocation units, the edges correspond to the published events. The trees can be connected to each other, but the whole graph is acyclic. The event flow graph for this example is shown on Figure 6.

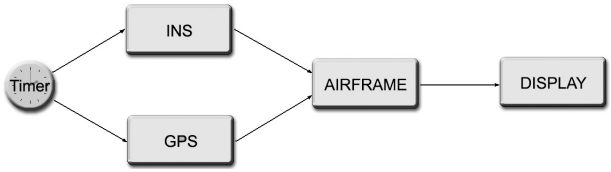

Figure 6. The event flow represented as a Directed Acyclic Graph (DAG)

The AIRFRAME component can receive events from both the INS and GPS components, therefore it is more complex than the other components. Since the number of vertices and the number of edges is finite, we can define a finite number of execution paths on this event flow, that is a path from the root element to a leaf. We need to decompose the AIRFRAME component to get the execution paths. If we assume OR semantics for receiving the events the event flow can be decomposed into the execution paths depicted on Figure 7.

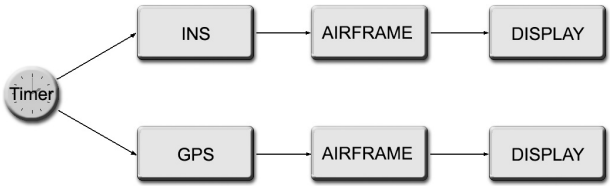

Figure 7. The execution paths assuming OR semantics

The main idea used in the decomposition is that we keep track of the event channel buffer in the state of the timed automata. For every event received the component publishes at most one event in any of its PublishPorts. We keep track of the events received in the state of the timed automata. This allows us to express AND semantics for receiving the events as well. Figure 8 shows this scenario. This rule is shown only to demonstrate that different semantics can also be expressed.

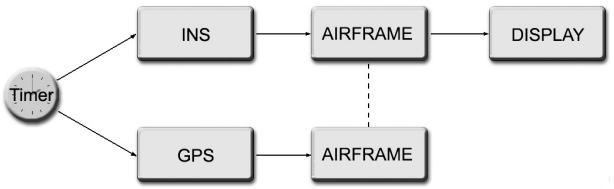

Figure 8. The execution paths assuming AND semantics

The dotted line denotes synchronization between the corresponding timed automata. We force the automata to start executing at the same time. This constraint - together with the guards on the transitions that control when the automaton is scheduled for execution - will enable the set of timed automata for execution if all of them are eligible. Since the component executes only once - unlike with the OR semantics - we allow sending out only one event.

7.2. Analysis using timed automata

We model each invocation unit on these call chains by a timed automaton. In this example - assuming `OR` semantics for receiving the events - we will have a network of 6 timed automata representing the 4 components. The event-flow corresponds to Figure 7. The time-driven invocation units will be represented as a single timed automaton while the event-driven invocation units will be represented by two concurrently executing timed automata.

Figure 9 shows the UPPAAL models for the example application shown on Figure 5. These models extend the generic model of the invocation unit introduced in section 6.

7.3. Verification

In the previous sections we have shown how to model the non-preemptive scheduling of component-based real-time CORBA applications. The timed properties of the model can be checked by the UPPAAL verification tool. UPPAAL uses a subset of the Computational Tree Logic (CTL) [5] temporal logic to formalize statements about the system models. Note that model checking is a formal proof that the model satisfies the desired properties.

To show that the system is correct we checked that the system is deadlock-free by using the following UPPAAL macro:

```
A[] not deadlock
```

We also need to show that all invocation units meet their deadlines. We claim that this requires no additional checking of properties. We have set the `frameOverrun` location to be committed to reduce the state space. However, we also introduced a nice side-effect in the system using this constraint. Whenever a timed automaton reaches the `frameOverrun` location time cannot pass in that automaton. Since we cannot leave that location, this will deadlock the system. *If the above reachability macro evaluates to true, we have proved that there are no deadlocks in the system and every action always finishes the execution before the deadline.* We also prove that every published event is properly consumed in the system and the event channels operate with limited buffer size.

In the previous section we have described a system with correct deadlines. Figure 10 shows a slightly different scenario in which we set the deadline of the `AIRFRAME` component to 32 ms. The deadlock is detected and a short trace is produced automatically by UPPAAL. The simulation runs help us finding the cause of the frame overrun by simulating the execution trace that lead to the fault.

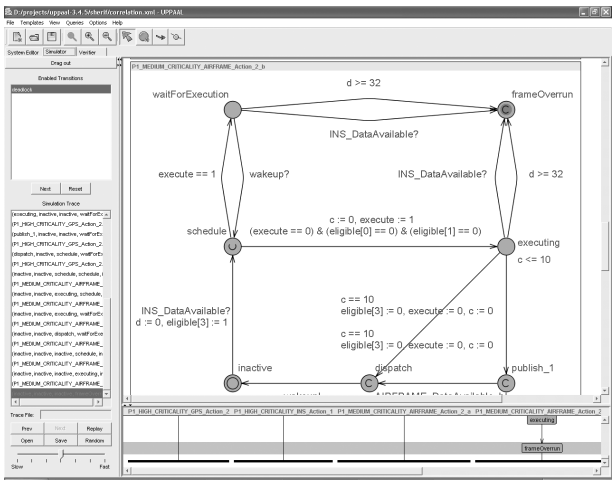

Figure 10. Detected deadlock with the shortest trace

Additional timed properties can also be checked because all the time and dependencies are captured in the models. We claim that our verification can be used to verify the correctness of non-preemptive scheduling and pinpoint components that have frame overrun conditions. Our results show that the deadline is not the function of the period and using slower Timers may produce the same properties in the system allowing better resource allocation and performance gain.

7.4. Performance

The example used in the case study turned out to be analyzable and correct. In order to find the limitations on the size of analyzable systems we have carried out a series of performance tests on a hyperthreaded machine with a 3.4GHz Pentium 4 processor and 1GB RAM. We used the generic timed automata model shown on Figure 4. We have assigned 1 unit of time as WCET to all automata and we allowed non-deterministic scheduling between the automata to increase the state space. This simple system was introduced solely to measure the performance. We checked the system for deadlock using the methods shown in section 7.3.

In the first series of tests we verified applications in which the event flow can be represented as a balanced tree. This case turned out to be exponential propor-

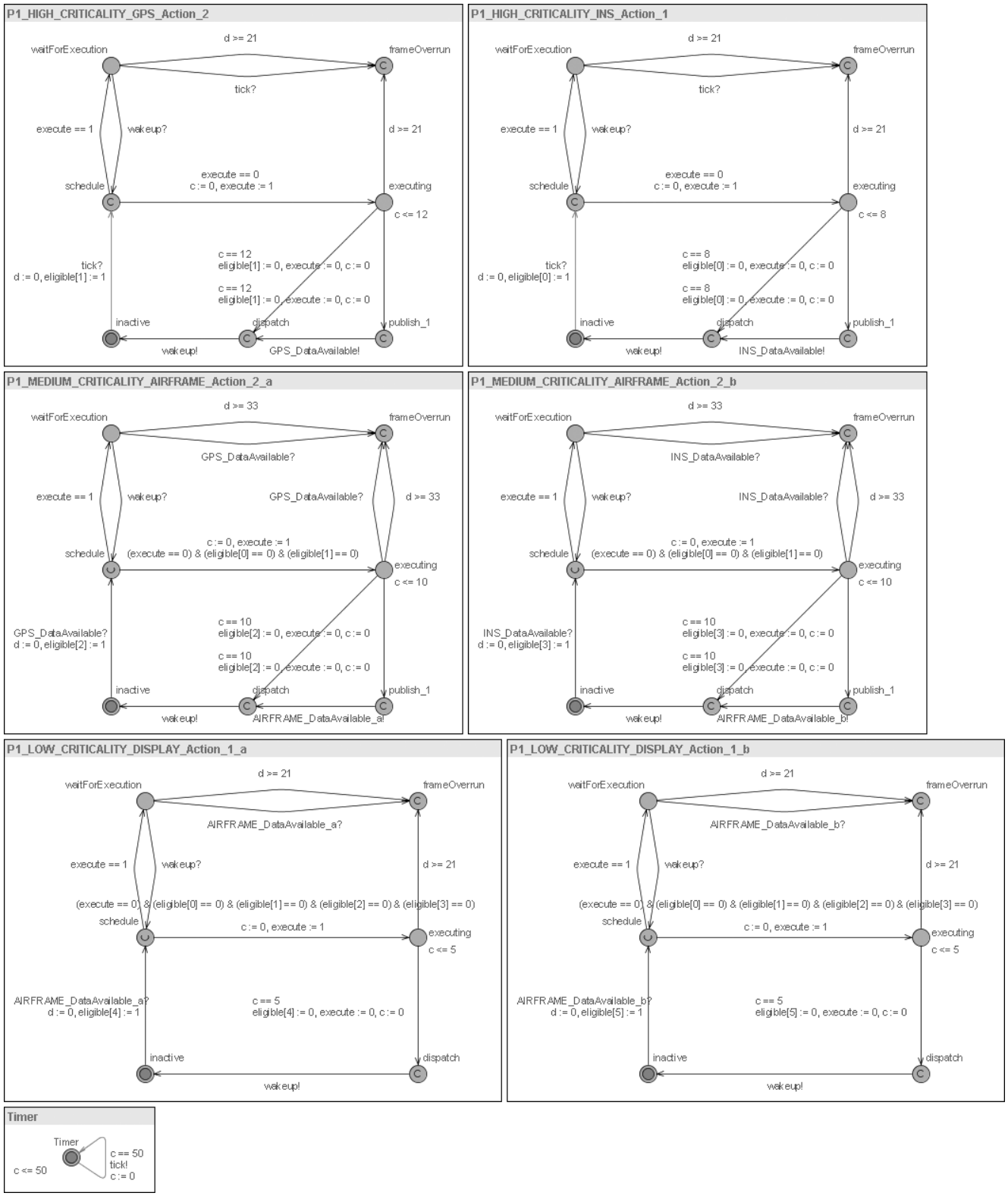

Figure 9. The network of timed automata for the Bold Stroke apllication shown on Figure 5

tional to the number of components. Then we have built a simple path of components consisiting of a single action that publishes to only one component. This system was easily analyzable (it took less than one second) even for 63 components. Our results show that if we limit non-determinism and the branching of event paths the system will be simpler to analyze. As part of out future work we would like to conduct research on how to speed up the verification process.

8. Related work

The DARPA MoBIES (Model-Based Integration of Embedded Systems) program - started in 2000 - focuses on integrating physical application domains with model-based design, composition and analysis of embedded software. The key objectives are to increase dependability, productivity while reducing the costs of development and analysis of automatically generated embedded system software. These objectives drive the development of domain-specific embedded software design tools to the target environment that allow the evolution of correct-by-construction code generation technologies. Within the context of the MoBIES program, researchers from multiple institutions have been working together to produce an end-to-end tool-chain with the Boeing Bold Stroke DRE architecture as the main application domain. Results of the research on Bold Stroke scheduling and verification are the AIRES, Cadena and Time Weaver - TimeWiz® tools.

The AIRES tool extracts system-level dependency information from the application models, including event- and invocation-dependencies, and constructs port- and component-level dependency graphs. Various polynomial-time analysis tasks are supported such as checking for dependency cycles as well as forward/backward slicing to isolate relevant components [14]. It performs real-time analysis [15] using Rate Monotonic Analysis techniques [23].

The Cadena [18] framework is an integrated environment for building and analyzing CORBA Component Model (CCM) based systems. Its main functionalities include CCM code generation in Java, dependency analysis and model-checking with dSPIN [9], an extension of the SPIN model-checker [20]. The emphasis of verification in Cadena is on software logical properties. The generated transition system does not represent time explicitly and requires the modeling of logical time that does not allow quantitative reasoning.

Time Weaver (Geodesic) [8] is a component-based framework that supports the reusability of components across systems with different para-functional requirements. It supports code generation as well as automated analysis. It builds a response chain model [23] of the system to verify timing properties. This model is used by real-time analysis tools such as the TimeWiz® model-checker to build a task set that can be analyzed with Rate-Monotonic Analysis techniques.

Several authors [4] [11] [3] [13] [16] [12] have proposed the use of model checking techniques and tools for dynamics analysis of real-time computation systems. The underlying models are variants of the timed automata model. Early work on this approach has been reported in [6] which uses the HyTech tool to analyze multi-tasking programs. A generic form to analyze scheduling behavior based on the timed automata model was proposed in [13] for single processor scheduling using the Immediate Ceiling Priority protocol and the Earliest Deadline First algorithm.

9. Conclusion and future directions

This paper presents a solution that captures the event-driven nature of component-based real-time CORBA applications that use the publisher/subscriber communication pattern. This approach captures the reactive behavior as well as the non-determinism present in these systems and demonstrates that timed automata can represent component interactions and asynchronous event passing allowing the verification of quantitative dense time properties. The verification process can provide simulation runs and pinpoint components that fail to meet deadlines and captures the propagation of failures in the system. This approach allows finding the "bottlenecks" in the system that limit the performance of the verified system.

Our solution has been implemented using the GReAT graph transformation tool that allows automatic verification of the system models by providing a tool-chain to the Uppaal model checker. As the performance tests show the state space explosion problem can be managed by restricting the branching of the event flow between components. As part of our future work we would like to include the modeling of preemptive scheduling in the models as well as the verification of dynamic scheduling algorithms. In order to keep the models analyzable we want to examine how to decrease the complexity of the verification.

References

[1] A. Agrawal, G. Karsai, and A. Ledeczi. An End-to-End Domain-Driven Development Framework. In *Proceedings of the 18th Annual ACM SIGPLAN Conference*

on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Oct 2003.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] V. A. Braberman and M. Felder. Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification. In *Software Engineering-ESEC/FSE 99*, pages 494–510, 1999.

[4] S. Bradley, W. Henderson, and D. Kendall. Using Timed Automata for Response Time Analysis of Distributed Real-Time Systems . In *24th IFAC/IFIP Workshop on Real-Time Programming WRTP 99*, pages 143–148, 1999.

[5] E. Clarke and E. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. *Logic of Programs, Lecture Notes in Computer Science*, 131:52–71, 1981.

[6] J. Corbett. Timing Analysis of Ada Tasking Programs. *IEEE Transactions on Software Engineering*, 22:1–23, 1996.

[7] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219. Springer-Verlag New York, Inc., 1996.

[8] D. de Niz and R. Rajkumar. Time Weaver: A Software-Through-Models Framework for Real-Time Systems. In *Proceedings of LCTES*, 2003.

[9] 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*, pages 261–276. Springer-Verlag, 1999.

[10] M. Deshpande, D. C. Schmidt, C. O'Ryan, and D. Brunsch. Design and Performance of Asynchronous Method Handling for CORBA. In *Proceedings of Distributed Objects and Applications (DOA)*, October/November 2002.

[11] C. Ericsson, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *Proceedings of RTSCA '99*, 1999.

[12] A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.

[13] T. Gerdsmeier and R. Cardell-Oliver. Analysis of Scheduling Behaviour using Generic Timed Automata. 42, 2001.

[14] Z. Gu, S. Kodase, S. Wang, and K. G. Shin. A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software. In *Proceedings of Real-Time Applications Synopsium*, 2003.

[15] Z. Gu, S. Wang, S. Kodase, and K. G. Shin. An End-to-End Tool Chain for Multi-View Modeling and Analysis of Avionics Mission Computing Software. In *Proceedings of Real-Time Systems Synopsium*, 2003.

[16] L. Halkjaer, K. Haervig, and A. Ingolfsdottir. Verification of the legOS Scheduler using UPPAAL. In F. Corradini and P. Inverardi, editors, *Electronic Notes in Theoretical Computer Science*, volume 39. Elsevier, 2000.

[17] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-Time CORBA Event Service. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 184–200. ACM Press, 1997.

[18] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of International Conference on Software Engineering*, 2003.

[19] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[20] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004. HOL g 03:1 1.Ex.

[21] J.Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. 1998.

[22] G. Karsai, S. Neema, A. Bakay, A. Ledeczi, F. Shi, and A. Gokhale. A Model-based Front-end to TAO/ACE. In *Proceedings of the 2nd Workshop on TAO*, 2002.

[23] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

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

[25] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, and J. Sprinkle. Composing Domain-Specific Design Environments. *Computer*, pages 44–51, Nov 2001.

[26] Object Management Group. Common Object Request Broker Architecture (CORBA/IIOP). 2002.

[27] Object Management Group. CORBA Component Model. 2002.

[28] P. Pettersson and K. G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, feb 2000.

[29] D. C. Schmidt, A. Gokhale, T. H. Harrison, and G. Parulkar. A High-Performance Endsystem Architecture for Real-Time CORBA. *IEEE Communications Magazine*, 14(2), 1997.

[30] D. C. Sharp and W. C. Roll. Model-Based Integration of Reusable Component-Based Avionics Systems. In *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[31] S. Tripakis and C. Courcoubetis. Extending PROMELA and SPIN for Real Time. In *Proceedings of TACAS '96, LNCS 1055*, 1996.