Towards Formal Construction of Middleware for Distributed Real-Time and Embedded Systems

Venkita Subramonian, Boris Klaydman, and Christopher Gill

Department of Computer Science and Engineering
Washington University, St. Louis, MO, USA
{venkita, bk3, cdgill}@cse.wustl.edu

Abstract. Distributed real-time and embedded (DRE) systems often support applications whose correct operation has important economic, health, or safety implications. Formal techniques for developing these systems are therefore desirable, to give a more rigorous assurance of their correctness. However, the complexity of even small-scale DRE systems makes it difficult to achieve this goal unless both formal techniques and middleware design principles and patterns are closely integrated during the design and implementation of each system.

This paper makes two main contributions to research on DRE systems. First, we examine the problem of integrating formal techniques with middleware design principles and patterns. We identify timed automata as a suitable formalism for expressing many design principles and patterns for DRE systems middleware. Second, we present a solution approach that uses side-by-side composition of (1) primitive software components and (2) timed automata models of those components, to offer sound and extensible system development and verification techniques. We demonstrate our approach with a series of examples of composition of a simple illustrative application, using increasingly sophisticated mechanisms to incorporate additional semantics into each of the previous examples.

Keywords. DRE systems, middleware, timed automata, model checking.

1 Introduction

Distributed real-time and embedded (DRE) systems pose inherent challenges in all phases of their development including design, implementation, verification, and validation. Testing accounts for a significant portion of the cost and lifecycle of complex mission-critical DRE systems, and yet still does not guarantee their correctness. This is very important especially for safety-critical systems. Formal approaches such as model checking have the potential to play a significant role in developing correct systems, in which a model of the system is expressed using a suitable formalism and the model can then be verified to ensure that it meets the specified system requirements before and possibly during actual system implementation.
However, existing approaches to ensuring correctness of DRE systems are inhibited by a chasm between how software systems are currently built in practice, and how inadequately formal analysis and verification methods have been applied to them to date. This chasm exists in part because there are non-trivial technical challenges to overcome in applying formal methods effectively to real-world system software construction. Even so, recent research projects that combine formal methods and systems approaches have shown that this chasm can be bridged using state of the art formal methods and tools, with significant benefits as a result.

For example, DARPA research programs such as the Model Based Integration of Embedded Systems (MoBIES) and Program Composition of Embedded Systems (PCES) programs have yielded a number of important advances in the application of formal methods (particularly in the area of model-driven computing) to the design, development, verification, and validation of DRE systems. However, the most promising technologies in the current state of the art are either general frameworks within which additional definition of the semantics of particular software models is needed, e.g., the Generic Modeling Environment (GME) [1] from the DARPA MoBIES program and its refinements like the VEST toolkit [2] from the DARPA PCES program, or are focused on a particular style of system software, e.g., the Cadena [3] component middleware modeling environment from the DARPA PCES program.

Although these approaches represent significant advances in the state of the art for using formal methods to develop DRE systems, there has been limited work on (1) studying the semantics of a fundamental set of reusable software elements from which a wide range of different system software frameworks can be built; (2) leveraging model checking, type systems, and other formal assurance methods to ensure their correct composition and configuration with respect to application-specific timing and concurrency constraints; and (3) examining practical advantages and limitations of that approach empirically and analytically for production-quality DRE system software.

2 Problem Statement and Solution Approach

Although modeling a system helps to evaluate its properties formally, it is often difficult to ensure that the model maintains high fidelity to the system as the system is refined through multiple design and implementation stages. This potential divergence between the model and the system is the fundamental problem addressed by our research on formal construction of middleware for DRE systems.

Part of this divergence stems from the dissimilar representations which are used by the formal methods and system development communities. The system development community often expresses DRE systems in terms of reusable software design patterns and idioms, whereas the formal methods community expresses DRE system models in terms of formal abstractions such as process algebras [4], actors [5], or virtual machines [6].
From the perspective of the formal methods community, the state of the art is to build systems by modeling them first, verifying model’s correctness and then generating code for the system implementation based on the model. However, in this approach, code generation may target different levels of abstraction, ranging from higher-level (e.g., object request broker) services to lower-level (e.g., operating system) services. The level of abstraction for which code generation is targeted has important implications for both the portability of the code and the fidelity with which the code can express the model.

From the perspective of the system development community, the state of the art is to engineer the code first and then extract the model from it. For example, a C++ generic programming concept may capture interface and behavioral attributes of an iterator type in the C++ Standard Template Library which can then be encoded via the traits idiom to guide composition of the iterator class with different algorithm implementations that may use it, automatically [7]. However, as the complexity of the software increases, policies and mechanisms are more likely to be introduced that are formalized inadequately (or even are undocumented) in the model. The consequence of this is that system developers cannot make well-informed decisions about configuring the middleware or other system infrastructure because of a lack of knowledge of encapsulated policies and mechanisms, which also reduces fidelity between the expected system behavior and what is actually observed.

Our research aims to reduce the chasm between the modeling of DRE systems and their design and implementation. We posit the following hypotheses which we are currently investigating. Our first hypothesis is that the basic software modules in production-quality system software development frameworks like ACE [8,9] are at a level of abstraction that strikes a suitable balance between detailed timing and concurrency semantics and ease of use and reuse. Our second hypothesis is that composing policies and mechanisms for run-time enforcement of timing and concurrency requirements, (e.g., for different forms of scheduling [10] and deadlock avoidance [11]), is sufficient to ensure critical system timing and concurrency constraints are satisfied. Our third hypothesis is that model checking optimization techniques (such as state space reduction and live variable analysis) provided by production-quality modeling environments (such as the IF Toolset [12]) can be combined with middleware-specific and application-specific optimizations to make model checking tractable even with multiple interacting constraints such as meeting deadlines and avoiding deadlock.

3 Illustrative Examples

In this section, we present a series of simple but illustrative examples of how timed automata can be used to specify the timing and concurrency properties, as well as the functional properties, of a system. We examine how the system can be developed using time-triggered and event-triggered approaches, and compare and contrast these approaches. We then show how timed automata can be used
to model middleware building blocks that can be used in the development of such applications in practice.

### 3.1 Cuckoo Clock Example

Our illustrative design challenge is to build the software components for controlling the physical activities in a hypothetical electronic cuckoo clock [13]. The clock has three hands - the second hand (moves every second), the minute hand (moves every minute), and the hour hand (moves every hour). At the completion of a minute, the seconds hand moves first, followed by the minute hand, followed by the hour hand. In addition, a “quail” chirps 4 times at the hour and is then followed by a number of “cuckoos” corresponding to the hour. At quarter-past the hour the quail chirps once, at half-past the hour the quail chirps twice followed by a single cuckoo, and at quarter-till the hour the quail chirps thrice.

To explore asynchronous events, our cuckoo clock example also has a red button that when pressed generates an asynchronous event that causes the clock to replay the most recent sequence of quail chirps and cuckoos. Finally, we extend the cuckoo clock to sound one cuckoo (with no quail chirps) and turn on a green indicator light, when a hypothetical sensor recognizes that coffee is ready in a coffee pot and notifies the cuckoo clock system via another asynchronous event.

This simple example shows both functional and temporal requirements of a real-time system. To analyze the correctness of the system that we build, we need to formalize the system specifications described above using a suitable representation that we can analyze formally. We have chosen timed automata [14] as the representation used by our approach due to its expressive power to specify functional, concurrency, and timing attributes of DRE systems.

### 3.2 Direct Composition

We now describe different ways to develop timed automata models for the cuckoo clock example described in Section 3.1. A direct approach, building a single timed automaton to model the entire system, is shown in Figure 1. As Figure 1 shows, the resulting model is rather complex even for this simple system. Most of the time is spent with the automaton in states S0 and S1. These states represent the movement of the second hand. We use a seconds counter, s, to store the number of seconds that have passed. Since our clock is required to perform a quail chirp action every 15 minutes (and at half-past the hour perform a single cuckoo action), we check a minutes counter, m, every 60 seconds and take appropriate actions in states S2, S3, S4, S5, and S6. Every 60 minutes we update the hours counter, h, and take appropriate actions (e.g., cuckoo as many times as is appropriate for the current hour). States S7, S8, and S10 are responsible for these actions. Finally, the “coffee ready” event is handled in state S9. In this design we used the time tick as a trigger for synchronization of transitions between states. We also used counters to keep track of the current state of the automaton.
This discussion reveals that the complex model shown in Figure 1 can be decomposed readily into several simpler subsystem models - a seconds subsystem, a minutes subsystem, an hours subsystem, and an asynchronous event subsystem (for coffee ready or replay button events) - all sharing a common starting state S0. There are different ways to compose these subsystems of which we discuss two approaches: a time-triggered approach that we discuss in Section 3.3 and an event-and-time-triggered approach that we discuss in Section 3.4.

As we describe in the following sections, we have used UPPAAL to model the cuckoo clock example. The UPPAAL tool has a nice graphical interface for building up system models using timed-automata, can run simulations, and can perform model-checking with respect to temporal logic expressions that describe properties to be checked.

### 3.3 Time Triggered Approach

The time triggered approach shown in Figure 2 uses multiple threads, with each thread responsible for controlling a particular hand of the clock. Each thread executes in a continuous loop, sleeping on a timer of the appropriate period between actions. Once a thread wakes up it moves the appropriate hand of the clock. The three different timers could be logical rather than physical, i.e., all could be mapped to a single physical timer. In any case, if three separate timers are used, then these timers must be synchronized with each other.

Using the time-triggered approach, the subsystems can be modeled as shown in Figures 3 through 6. Consider the seconds subsystem of the cuckoo clock, which we have represented in UPPAAL. Figure 3 shows that the seconds subsystem model has an exact correspondence to the part of the monolithic system model that is labeled “Seconds” in Figure 1.

---

**Fig. 1. Timed Automaton Representation of Cuckoo Clock**
Even though the minutes subsystem shown in Figure 4 appears slightly different from the part labeled “Minutes” in Figure 1, it is easy to see the resemblance between the two: states M1, M2, M3, M4, M5, and M6 in Figure 4 correspond to states S1, S2, S3, S4, S5, and S6 in Figure 1.

In Figure 1, the hours subsystem consisted of states (S7 and S8) that had more transitions back to the originating state, S0, than to each other. The decomposition of this subsystem not only allows us to make the representation more compact, but also to simplify the automaton as Figure 5 illustrates. In fact, state H3 is shown in Figure 5 only in order to reveal similarities with the original model shown in Figure 1.

The remaining two subsystems, “coffee ready” and “tell time”, are nearly identical in their UPPAAL representations, as shown in Figure 6. To simplify execution of our illustrative model, both subsystems are represented here as being driven by periodic events, but either or both of these subsystems could be driven by asynchronous events in practice.

The time-triggered approach thus allows significant decoupling of the different subsystems. Each of the subsystems is unaware of, and hence not explicitly dependent on, any other subsystems. There is only a temporal coupling between the different subsystems since their clocks are all synchronized in the composed system model.

There are, however, some assumptions in this approach that could be impediments to accurate modeling of (especially distributed) systems in practice, e.g., that the timers must be well synchronized. If we use a single hardware timer to drive all the subsystems, there may be a situation where all the subsystems need to be triggered at the same time. For example, at the 3000th second, all the subsystems need to be triggered. In our example, we have a requirement that the seconds hand must move before the minutes hand and the minutes hand before
Fig. 4. Time Triggered Minutes Subsystem

Fig. 5. Time Triggered Hours Subsystem

Fig. 6. Coffee Ready and Tell Time Subsystems
the hours hand. This cannot be guaranteed unless we introduce a notion such as priority that orders the actions of the individual subsystems. In fact, without this, the system would not be correct. For example, we checked the following expression using the UPPAAL verifier:

\[ E \equiv \text{Seconds}.S1 \text{ and } s=59 \text{ and } \text{Minutes}.M1 \text{ and } m=1 \]

Here we are using model checking to determine whether there is any state in the model where the second hand has not advanced yet, but the minutes hand has already advanced. The expression was satisfied during model checking in UPPAAL which indicates a difference between the the intended behavioral requirements and the system model. Once this problem has been detected we can address it, e.g., by running the subsystems in different threads in our system implementation and giving an appropriate priority to each thread. We can then extend our model and assign priorities to the transitions in it, e.g., using timed automata extended with priorities [17], to reflect that change in the system design.

### 3.4 Event and Time Triggered Approach

In the time-triggered approach, there was no communication between the seconds, minutes and hours subsystems. We can extend this approach by introducing communication between the individual automata, as Figure 7 shows. In this approach, the seconds handler sends a notification event to the minutes handler every 60 seconds and the minutes handler sends a notification event to the hours handler every 60 minutes.

![Fig. 7. Event and Time Triggered Approach for Cuckoo Clock System](image)

The main distinction when compared to the purely time-triggered approach is that there is an explicit dependency between the individual subsystems to achieve synchronization between the subsystems. This form of synchronization can be achieved using shared global variables, and is more tightly coupled than an explicit communication model between the individual automata. We now consider the modeling of the individual subsystems using combined event-based and time-based triggering.
The only change that was necessary was the addition of a channel for communication between the seconds and minutes subsystems (named seconds60 in Figure 8), which is responsible for triggering an event in the minutes subsystem to update the minutes counter when the second hand makes a complete cycle.

![Diagram of Event and Time Triggered Seconds Subsystem](image1)

**Fig. 8. Event and Time Triggered Seconds Subsystem**

For the minutes subsystem shown in Figure 9, we have added two channels of communication. The minutes60 channel is necessary to notify the hours subsystem of minutes hand making a complete circle. The minutes30 channel is necessary so that a cuckoo can be sounded once every half hour.

![Diagram of Event and Time Triggered Minutes Subsystem](image2)

**Fig. 9. Event and Time Triggered Minutes Subsystem**

Similarly, the hours subsystem shown in Figure 10, instead of relying on complex transitions, now relies on channel communication with the minutes, coffee ready, and tell time subsystems.
3.5 Modeling the Middleware Layer

In the previous sections, we have described how our illustrative cuckoo clock example can be modeled using communicating timed automata. In doing this we did not take into account how this application might be supported by operating systems or middleware. During the process of design and implementation, one would typically use standard design patterns [18] and pattern languages [19] to promote reuse of existing design and implementation techniques and artifacts. For example, distributed applications often use communication middleware to separate the network programming concerns [20] from the application logic. Frameworks like ACE [8] provide reusable and portable patterns-based implementations of primitive middleware building blocks like reactors [21], timer queues [22], and message queues [22].

To increase the fidelity of a system model to the system’s actual behavior, the middleware building blocks out of which a system is built should also be modeled. In this section, we show how we can extend support for the cuckoo clock example using existing middleware building blocks from ACE, and then model those building blocks. The models thus obtained can be used in modeling more complex systems built using the building blocks the models represent. Figure 11 shows the fine-grained middleware elements that we used, and the interactions between them.
Reactor is an event handling design pattern [21] used in network programming to demultiplex events from multiple sources using one or more threads (here we assume only one). This design pattern is used in Object Request Broker (ORB) middleware [23] to demultiplex and dispatch incoming requests and replies from peer ORBs. Event handlers like request and reply handlers are registered with the core abstraction in the Reactor pattern implementation, which is itself called a reactor. A reactor uses a synchronous event demultiplexor, e.g. the UNIX select [24, 25] system call, to wait for data to arrive via one or more inter-process communication (IPC) service access points. When data arrives, the synchronous event demultiplexor notifies the reactor, which then dispatches the appropriate event handler that is registered for the event source.

Apart from demultiplexing events from multiple connections, the reactor also maintains an internal Timer Queue that is used for providing timeout events to the application. Timeout Handlers are registered with the reactor through the schedule_timer interface method. The application starts running by invoking the run_event_loop method on the reactor which in turn waits on the Event Demultiplexor by passing the earliest timeout obtained from the timer queue into the demultiplexor (e.g., the select system call). On expiration of the timeout the demultiplexor returns and the reactor calls the appropriate timeout handler. This timeout handler can be used as a trigger for controlling the cuckoo clock.

Figure 12 shows a timed automaton model for a timeout handler. The automaton waits for the handle_timeout event from the reactor automaton. When it gets this event it performs appropriate application-specific processing. In the case of the cuckoo clock example, the timeout handler could be responsible for triggering the various cuckoo clock subsystem automata.

Figure 13 shows a timed automaton model for an event demultiplexor. The event demultiplexor could be implemented using various OS specific mechanisms like select, poll, or WaitForMultipleObjects, and the semantics of those different mechanisms would be captured by different demultiplexor models in practice. In this case, we are considering only the timeout events that occur when no other
events have occurred within a period of time that was specified as a parameter. When the timeout event occurs, the reactor is notified.

Figure 13. Event Demultiplexor Automaton

Figure 14 shows a simplified timer queue model that can store only one timeout request. In practice, this model can be extended to store more than one timeout by using data structures like heaps, or hash maps to model more complex timer implementations, but the simplified version is sufficient for our discussions here. The timer queue also keeps track of which timeout handler is associated with timeout events.

Figure 14. Timer Queue Automaton

Figure 15 shows an automaton that maintains a list of the handles to IPC connections that the reactor maintains. This automaton models a simple handle set with which the reactor demultiplexes events to event handlers, using a first-in-first-out policy. Alternative implementations could use prioritized handle sets, or a round-robin policy, for dispatching handlers.

Figure 16 shows a simple reactor model that captures timeout handling semantics. The reactor communicates with the other automata that we have described above, using two-way function call semantics. When a timer expires the reactor calls `handle_timeout` on the appropriate timeout handler.
4 Related Work

The time-triggered architecture (TTA) [16] is designed for fault-tolerant DRE systems. Within the TTA, all system activities are initiated by the progression of a globally synchronized time-base. This stands in contrast to event-driven systems, in which system activity is triggered by events. A central characteristic of the TTA is its treatment of time as a dominant abstraction. The TTA decomposes a large embedded application into clusters and nodes, and at every node provides a fault-tolerant global time base of known precision. The TTA takes advantage of this global time to specify precisely the interfaces among the nodes,
to simplify communication, to establish state consistency, to promptly perform error detection, and to ensure the timeliness of real-time applications.

The time-triggered message-triggered object (TMO) [26, 27] approach removes limitations of conventional object-oriented techniques in developing applications containing real-time distributed computing components. It is a natural and syntactically small but semantically powerful extension to traditional object-oriented design and implementation techniques, which allows the system designer to specify timing characteristics of data and function members of distributed objects.

Our approach is complementary to these approaches. The TTA and TMO define broad architectures under which real-time and embedded systems can be developed. Our research supplements these by addressing issues at a lower level of abstraction - formalizing the building blocks used for developing a variety of higher-level middleware implementations.

Giotto [28] provides an abstract infrastructure model for the implementation of embedded control systems with hard real-time constraints. Giotto specifies time-triggered sensor readings, task invocations, actuator updates, and mode switches independent of any particular implementation platform. Giotto can be annotated with platform constraints that are directives for the Giotto compiler, but do not alter the functionality or timing of a Giotto program. By separating the platform-independent and platform-dependent concerns, a great deal of flexibility is gained in choosing control platforms as well as a great deal of automation in the validation and synthesis of control software. The time-triggered nature of Giotto achieves timing predictability, which makes Giotto particularly suitable for safety critical applications.

Our research differs from this approach in the level of abstraction of the generated code and in the applicability of our approach to both time-triggered and event-triggered systems. We focus on enabling code generation at the middleware level and on modeling the composition of primitive elements out of which the middleware itself is built.

Synchronous programming languages such as Esterel [29] are dedicated to programming reactive systems including real-time systems and controllers. A compiler translates Esterel programs into finite-state machines. The Esterel compiler can be used to generate a software or hardware implementation of an entire reactive program, or can also generate C-code to be embedded as a reactive kernel in a larger program that handles the interface and data manipulations.

LUSTRE [30] is another synchronous language for programming reactive systems. It is a declarative language in that it describes a set of equations that must be verified according to a set of program variables. A program variable in LUSTRE is a function of multiform time: it has an associated clock which defines the sequence of instants where the variable takes on particular values.

The semantics of synchronous languages are formal models upon which their programming environments are defined. Compilation involves the construction of these formal models, analysis and optimization of static properties, synthesis of executable sequential implementations, and automated distribution of programs.
Our research focuses on developing formal models for middleware elements that could be composed while generating the software implementation for such systems. Our approach could also complement synchronous languages by providing a higher level of abstraction in the software elements targeted by their code generators.

5 Concluding Remarks

In comparing various model decomposition approaches, we have discovered the following themes. In general, decomposition of more complicated automata will allow the system developers to identify potential design and modeling problems more easily. There are two main ways that such decomposition can be performed. The first is to start with a complete model, then identify separate components for decomposition and translate them into timed automata separately. We have used that approach in this paper, to illustrate model decomposition using our simple cuckoo clock example. Even for this simple example, we were able to identify problems with the original design.

The second approach, which we are pursuing as future work, is to start from models of individual system software building blocks, and compose them according to how the system is built. The advantage of this approach is that the designer will potentially make fewer mistakes in designing and modeling individual building blocks separately. However, the designer potentially may need to reconsider communication and synchronization assumptions once composition is done, and develop additional formalisms to ensure sound composition.

While timed automata are useful for representing the functional, concurrency, and real-time properties of DRE systems, in some tools it can be difficult to model dynamic creation and destruction of subsystems and communication channels at run-time. This is essential to model ORB-like systems where new connections are created to exchange messages between a client and server, and then either destroyed or cached for later use. Production-quality formal modeling tools like the IF Toolset [12] provide facilities for modeling such dynamic behavior using timed automata. As part of our future work, we are investigating the applicability of such tools to modeling middleware building blocks and to composing those finer-grained models to achieve high-fidelity models of entire systems.

References

18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.


