Towards Integrated Model-Driven Verification and Empirical Validation of Reusable Software Frameworks for Automotive Systems *

Venkita Subramonian and Christopher Gill

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

Abstract. Software for automotive systems is rapidly increasing in complexity and scale, and leveraging reusable software frameworks in the development of these systems offers significant potential to reduce engineering costs and cycle times. However, the development of practical models and verification and validation techniques for automotive software built with reusable frameworks remains an open research challenge. This paper makes three main contributions to the state of the art in software engineering for automotive systems. First, it summarizes ways in which reusable software frameworks are relevant to automotive software engineering. Second, it describes an approach to verification and validation of reusable software frameworks which we have developed for other application domains. Third, it presents an evaluation of our approach in the context of an illustrative verification and validation scenario.

1 Introduction

The increasing complexity and scale of software for automotive systems argues for increasing re-use of software in the development of those systems. Because interacting software functions are increasingly distributed across many embedded microcontrollers in automotive systems, leveraging reusable middleware in the development of these systems offers significant potential to reduce engineering costs and cycle times.

However, these benefits only can be realized if the reusable middleware can be specialized through configuration and customization to address constraints, optimizations, and trade-offs in timing and other quality of service (QoS) dimensions that are specific to the automotive software applications being developed. Furthermore, system developers must be able to verify designs involving middleware prior to investing in their implementation, and to validate those implementations prior to investing in their commercial deployment.

Model checking can play a valuable role in verifying automotive applications' increasingly heterogeneous constraints, e.g., for safety-critical functions like computer assisted steering and braking and for comfort functions such as in-vehicle navigation

* Research supported in part by NSF CAREER award CCF-0448562.

M. Broy, I.H. Krüger, and M. Meisinger (Eds.): ASWSD 2006, LNCS 4922, pp. 118–132, 2008.

and entertainment systems. As we discuss in Section 5, model checking also can generate verification traces to guide validation experiments, and the comparison of verification and validation traces can be valuable to assess and improve the fidelity of the models with respect to the actual behavior of the implemented software.

Our previous research has focused on specializing reusable middleware frameworks to address footprint and timing trade-offs in networked embedded systems [1], enforcing run-time timing [2] and liveness [3] constraints at run-time, and developing high-fidelity timed automata [4] models of canonical reusable software building blocks that are widely used in practice [5]. In this paper we describe our most recent research on the integrated verification and validation of systems built with reusable software frameworks. The results of that investigation (which we summarize in Section 5) demonstrate the need for careful co-design in both the models and the reusable software itself to ensure that (1) scientifically valid comparisons are made between the software and the models, (2) irrelevant differences between the software and the models are abstracted away to reduce the effort required for verification and validation, and (3) relevant differences between the software and the models are preserved and analyzed to reveal important mis-matches between the software, the models, and the application requirements, which at least must be documented and where possible corrected .

The rest of this paper is structured as follows. Section 2 summarizes other research related to our approach, and to its application in the automotive software application domain. Section 3 summarizes software platforms, frameworks, and design patterns that are relevant to automotive applications. Section 4 describes our solution approach, in which timed automata models of reusable middleware building blocks are used to verify properties of software built using them, and to generate traces used for integrated verification and validation. Section 5 presents an illustrative example drawn from our previous research in the avionics software domain, and shows how our approach can be used to verify and validate that example: our results show that (1) checking models of reusable middleware building blocks can verify timing properties of software that uses those building blocks, (2) verification and validation can be integrated through collection and comparison of detailed time and event traces, and (3) observed differences between the verification and validation traces can be used to refine models to reflect more accurately the actual software implementations they represent. Section 6 concludes the paper with a summary of observations and recommendations arising from this research, and describes remaining open research challenges and future work for verification and validation of automotive software built upon reusable software frameworks.

2 Related Work

DREAM [6,7,8] provides an open-source tool and methodology that allows distributed real-time embedded (DRE) system designers to do model-based schedulability analysis of time and event-driven DRE systems. DREAM offers a computational model called the DRE semantic domain [7]. The key elements in this computational model are tasks, timers, event channels and schedulers. Tasks are triggered either by a timer or by external aperiodic events, and tasks communicate among themselves by means of an event channel. Within this computational model, DREAM considers the problem of

deciding the schedulability of a given set of tasks with time and event-driven interactions. By using timed automata models for each of the elements in the computational model, the schedulability problem is converted [8] into a reachability problem in the composed model through a model checking tool like UPPAAL. DREAM also provides a model transformation facility by which a model of the DRE system is expressed using a domain specific modeling language (*e.g.*ESML [9]), and is transformed using model transformation [6] tools to create timed automata models in the DRE semantic domain.

Even though our approach is similar to DREAM in that we use timed automata models to verify system properties, the problems that these two bodies of research address are different. Whereas DREAM addresses the problem of deciding schedulability of a set of tasks under the DRE semantic domain, our research addresses the problem of correct composition of reusable software elements that are at a finer level of granularity than the elements in the computational model offered by DREAM. Both these kinds of analysis are important - while the higher level computational model provided by DREAM helps the DRE systems designer to address the schedulability problem in time and event-driven systems, our approach helps the system designer choose configurations that are appropriate for the specific application. Moreover, the computational model in DREAM makes an assumption that all communication between tasks uses an event channel, and the communication between tasks and event channels themselves are abstracted away using synchronized transitions in UPPAAL. During actual implementation, these synchronized transitions are realized using reusable software which could have different configurations that impact the timing and liveness properties of a DRE system in different ways. Hence a more detailed model of the fundamental reusable software elements is necessary, which has been the focal point of our research.

Automotive Software Verification [10,11] uses a design pattern based approach to build reusable software that provides high-level communication services to higher layer automotive software tasks. A middleware architecture for communication is realized in the context of the OSEK/VDX operating system. The communication activities carried out by the middleware are mapped on to tasks in OSEK/VDX. [12] discusses code generation from a high level RT-UML [13] model for OSEK/VDX. That work identifies key issues in mapping of UML models where some annotations in RT-UML cannot be mapped directly to the constructs and primitives offered as part of OSEK/VDX.

AUTOSAR (AUTomotive Open System ARchitecture) [14] is an open standard for automotive software architecture that specifies standardized interfaces for communication among automotive electronic components. It aims to alleviate the complexity involved in developing and upgrading software based control systems in the automotive domain. The use of abstract concepts that are fundamental building blocks in a particular domain combined with adequate modeling techniques and models based on these fundamental building blocks can have a valuable and sustainable impact on automotive software engineering.

3 Automotive Software Engineering

Fine-grained reusable middleware frameworks like ACE [15] address the challenge of providing common domain-specific building blocks that can be used to build higher

level software abstractions. For example, the ACE framework provides building blocks like Reactor, Acceptor, Connector and Active Objects, which have been used to build a wide range of reusable middleware frameworks and services for distributed real-time embedded systems - *e.g.*, for real-time scheduling (Kokyu [16]), distributed communication (TAO [17], nORB [18,19,20]), and component middleware (CIAO [21]). The presence of such foundational fine-grain abstractions is not limited to reusable software built on ACE. For example, in the sensor networks application domain, TinyOS [22] provides building blocks like Timer, ADC, RFM, Active Messages, *etc.* for building different kinds of sensor network middleware - *e.g.*, for reconfiguration, scheduling, group communication, and self-stabilization. We now discuss a similar low-level framework in the context of automotive software engineering.

Operating System Features. OSEK VDX [23] is a set of interface specifications for operating systems, communication, and network management in the automotive domain. The OSEK operating system is targeted to run on micro-controllers and is therefore designed to require a minimum of hardware resources (e.g., CPU and memory). These specifications enable automotive OEM and third-party ECU (electronic control unit) suppliers to use a standardized set of APIs to facilitate system integration, thus making automotive applications more portable, reusable and interoperable. A customized version of the OSEK OS can be generated by using the OSEK Implementation Language (OIL), through which one can specify a portable description of all OSEK-specific objects (*e.g.*, events, tasks, resources).

An OSEK compliant operating system implementation provides automotive application developers with a set of reusable primitive building blocks that include (1) tasks, (2) event objects, and (3) messages. Tasks are the equivalent of threads in general purpose operating systems. They are the basic schedulable entities in the OSEK/VDX and forms the basis for enforcing the various real-time requirements of automotive applications. Event objects are used to inform tasks of various events occurring in the system - *e.g.*, arrival of messages on a communication link, or expiration of a timer. Messages are used to communicate between software components residing within an ECU.

POSA2 Abstractions. A task in OSEK/VDX can wait on multiple events at a time using the *WaitEvent* function, which is a key feature of the Reactor pattern [24]. Reactor is an event handling design pattern used in network programming (*e.g.*, in ACE [15]) to demultiplex events from multiple sources, possibly using just a single thread. This design pattern is used in low level reusable middleware to demultiplex and dispatch incoming requests and replies from peers. Event handlers like request and reply handlers are registered with a reactor. The reactor uses a synchronous event demultiplexer, *e.g.*, the UNIX *select* system call, to wait for data to arrive from one or more peers. When data arrives, the synchronous event demultiplexer notifies the reactor, which then dispatches the appropriate event handler based on the event source.

The Active Object pattern, which separates method invocation from method execution, is also relevant. Since the thread of execution is separate from the thread of invocation, this pattern can be used to serialize access to resources used by multiple threads. This pattern can be used, as is discussed in [10,11], to separate the communication subsystem from the automotive application by using different tasks for each of these layers of the system.

4 Solution Approach

In previous research [25,5], we have developed timed automata [4] models of reusable software building blocks that have been used to implement a wide range of software frameworks and applications. These lower-level timed automata models of the reusable building blocks can be combined with higher-level formal models of the applications and frameworks that use them to provide a faithful model of a system *including the reusable software platform on which the system is deployed*, such that the composite models can be verified for correctness with high fidelity to the implemented system.

Our focus has been on creating timed automata models of reusable software objects provided by ACE [15]. ACE is a portable C++ framework used for developing high-performance concurrent and real-time software using threads, sockets, and other mechanisms provided by a wide range of OS platforms. By developing high fidelity formal models at a level of abstraction that is just above the operating system, our approach adds rigor to other model-based approaches currently being pursued by the systems research community, which target reusable software architectures at higher levels of abstraction. Our approach also provides sound and composable models of foundational reusable software building blocks to the formal methods community, offering new opportunities for innovation in formal methods that can directly impact the design, implementation, verification, and validation of real-world software systems.

In Section 4.1 we first present a simple example drawn from our previous research in the avionics software domain, which serves to illustrate and motivate our approach. In Section 4.2, we then give an overview of the ACE building blocks we have modeled, and discuss the suitability of several model checking tools for verification of timing and liveness properties in software built using those building blocks.

4.1 Illustrative Example

In this section, we illustrate how timed automata models can be used to analyze timing and liveness properties in software built upon reusable software building blocks. We first present a motivating example [26] - a simple distributed real-time embedded subsystem from the domain of avionics mission computing [27] - and describe how our modeling approach presented in Section 4.2 can be used to describe the reusable ACE software building blocks incorporated within that example. In Section 5 we then show how our approach can be used to analyze timing properties of this example subsystem *taking into account the semantics of the reusable software building blocks with which this system is implemented.*

Figure 1 shows the elements of our example avionics system: (1) a *Rate Generator*, which wraps a hardware timer and sends periodic events to event consumers that register for those events; (2) a *GPS Subsystem*, which wraps one or more hardware devices for navigation and caches a periodically refreshed location value to provide low-latency response; (3) a *Graphical Display*, which wraps the hardware for a heads-up display device in the cockpit to provide visual information to the pilot and a location value that is updated by querying an interface on the GPS component when the controlling software receives a triggering event.

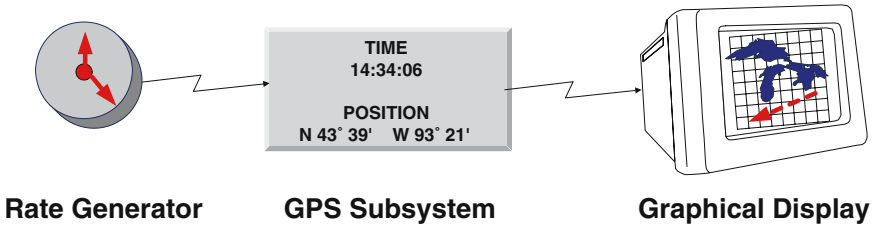

Fig. 1. Example Avionics System

This example is representative of a broader class of distributed real-time embedded systems where clusters of closely-interacting components are connected via specialized networking devices, such as VME-bus backplanes. Although the functional characteristics of these systems may differ, they often share the rate-activated computation and display/output timing constraints illustrated here.

Both control flow (rate generator to GPS and GPS to display) and data flow (display to GPS) interactions occur in this subsystem. An event push style of communication is used by the rate generator (to send a timer-driven triggering event to the GPS), and by the GPS (to communicate the availability of data to the display). A data pull style of communication is then used by the display subsystem (to obtain location data from the GPS). In the middleware-based software framework from which this example was drawn, the push style of communication is typically implemented using a publish-subscribe event channel, and the pull style of communication is typically implemented using a remote function call.

Even though middleware-based software architectures currently are not prevalent in the automotive software engineering domain, the low-level software building blocks that are the focus of our work are directly relevant there, as we have discussed in greater detail in Section 3. Furthermore, as reducing development costs and cycle times becomes increasingly important, specialized reusable middleware solutions designed for stringent timing and footprint constraints [18,19,20] may be adapted further for the automotive software engineering domain. Therefore, the observations and lessons learned from our verification and validation studies of this example, which we present in Section 6, are relevant to automotive software engineering.

Figure 2 illustrates how reusable low-level building blocks like the reactors, event handlers, and thread pools provided by ACE, are incorporated into the example shown in Figure 1. Each communication channel in the example subsystem illustrated in Figure 1 has a corresponding event handler. For example, the Timer_EC_EH event handler handles requests sent from the rate generator to the Event Channel (EC), the GPS_EC_EH event handler handles requests sent from the GPS unit to the EC, etc.

To illustrate how timed automata models of reusable software can be used to analyze timing and liveness properties in practice, we now consider a simple but representative example scenario using the low-level models of reusable ACE building blocks described in Section 4.2 in order to (1) capture the semantics of the reactor and event handler models, (2) illustrate how interference with specified constraints on timing can arise in software built with those reusable software building blocks, and (3) show how the particular form of interference that may arise can be analyzed through model checking.

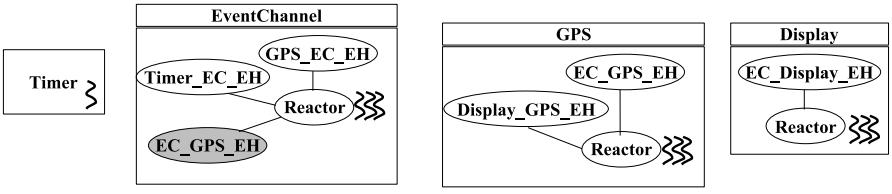

Fig. 2. Reusable Software in the Avionics Example

In many distributed real-time embedded systems, correct operation can depend on satisfying stringent but relatively simple timing constraints, such as receiving the result from a remote method invocation before a relative deadline. In this example, system timing is affected by interference between nominally independent call sequences, when they must contend for shared resources such as the CPU. We consider a scenario where a single thread is used by a reactor to demultiplex events to its registered event handlers. The extent to which the event handlers contend for shared resources impacts whether or not a deadline miss can occur. Using our models we then can determine (1) whether any deadline misses *can* occur due to interference between call sequences, and (2) if a deadline miss is possible, which sequences of actions can cause it to occur. For example if the rate generator and GPS push events at roughly the same time, then whichever event handler (Timer_EC_EH or GPS_EC_EH) is dispatched first could delay the other event handler, potentially resulting in a missed deadline.

4.2 Modeling in ACE

To be able to verify the correctness of customized reusable software in the context of each specific application, we have developed detailed and formal models of common reusable software building blocks found in the widely used ACE [15] framework, such as reactors, thread pools, event handlers, and interaction channels, which can be composed and checked rigorously to evaluate timing and liveness properties in each particular application and its supporting reusable software configuration. A crucial challenge is to determine the appropriate level of abstraction at which to model system software. To answer this question, one must look at the kinds of abstractions used in state-of-the art system implementations. For example, distribution middleware services such as CORBA [28] object request brokers (ORBs) provide a level of abstraction that promotes portability and reusability and hence makes an appealing candidate for formal modeling. Since many state-of-the-art distribution middleware implementations expose sets of configuration options used to tailor the reusable software for particular applications, modeling the combinations of configuration options [29] is a useful and necessary step toward model-driven construction and verification of distributed real-time embedded systems. We contend, however, that to evaluate issues such as timing and liveness, which are crucial to many distributed real-time embedded systems, finer-grained models of lower-level reusable software building blocks are needed to capture (and supplement analysis of) crucial details related to concurrency and interaction.

Results of our previous experience with system software construction indicate the efficacy of such a fine-grained approach. In that work we built a special-purpose ORB called *nORB* [18,19,20], with support for real time operation dispatching in the context of memory constrained networked embedded systems. We took a fine-grained bottom-up approach to the development of nORB, starting with lower level elements of the ACE [15] framework: Reactor, Acceptor, Connector, CDR Stream, *etc.* Along with taking a fine-grained approach to building nORB, we used the application itself as a guide for making fundamental design and implementation trade-offs. That work has given us insights into application-driven construction and customization of reusable software for this and other domains, allowing us to define composable models with a high degree of fidelity to how reusable software is built in practice.

Our modeling approach is designed specifically for analysis of timing and liveness in concurrent software with real-time constraints. We rely first on model checking to ensure soundness. Due to the potential size of the state spaces that need to be checked, we then apply several optimizations: (1) building highly modular models, by sub-dividing them into fine-grain composable automata; (2) encoding our models in formats used by model checkers that allow automata to be added to a model, or removed from it, dynamically; and (3) adopting a hybrid approach in which parts of the analysis are provided by other analysis techniques [30,3] thus reducing the state space that must be explored through model checking. Model checkers such as UPPAAL [31], IF [32], Bogor [33], and SPIN [34] each have their particular features and restrictions. For example, among these four tools, timed automata models are supported only by UPPAAL and IF, whereas only Bogor supports object-oriented and concurrent constructs explicitly. UPPAAL uses a rendezvous model of communication whereas in IF communication is asynchronous. Because our models must capture time, concurrency, and asynchronous interactions between system elements that can be added and removed dynamically, we selected IF as the most suitable model checking environment for our needs in that work.

Figure 3 shows our model architecture, which is implemented using the IF tool set [32,35,36]. We specify our fine-grained models as IF *processes* that run in parallel and interact through shared variables and asynchronous signals. The behavior of these processes is represented formally in IF as *timed automata with urgency* [37] and the semantics of a system modeled in IF is the Labeled Transition System (LTS) obtained by interleaving the executions of its processes.

Our models are divided into three layers: (1) models of network and OS level abstractions such as channels for interprocess communication; (2) models of semantically rich reusable software building blocks like reactors; and (3) models of the application functionality implemented in the form of event handlers. Although Figure 3 shows a static view of our models, the models themselves are executable in the IF environment and can be checked against system property specifications. The unshaded rectangular boxes shown in Figure 3 are modeled using timed finite state automata specified using the IF language. The shaded rectangular boxes shown in Figure 3 are data structures that are shared by the different automata in the models. Automata with timed transitions (transitions that are guarded with conditions based on clock variables) are indicated in Figure 3 by timer icons. [38] and [39] provide detailed explanation of these models.

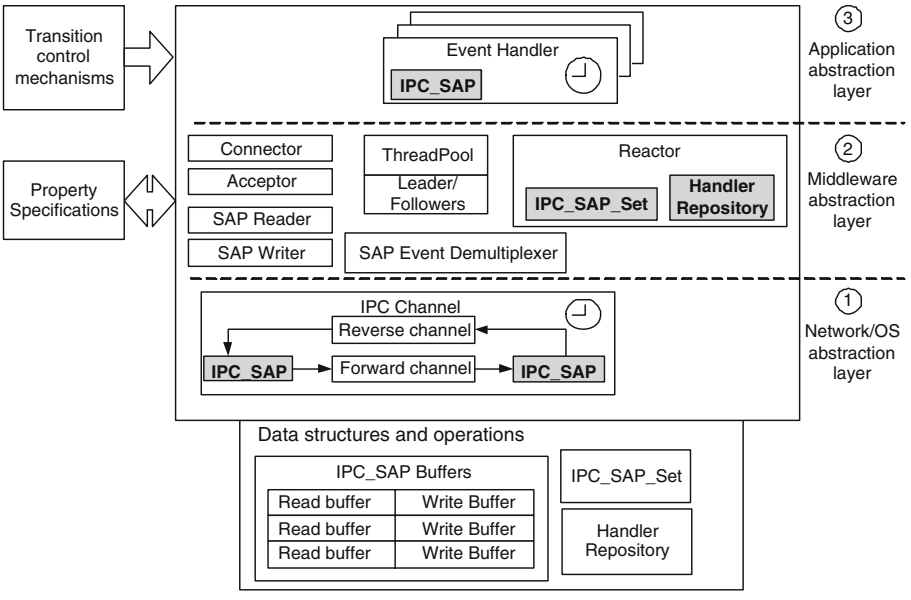

Fig. 3. Model Architecture

5 Integrated Verification and Validation

We now summarize the results of a verification and validation study we conducted to evaluate the fidelity of the reusable middleware models developed in our previous research [5] by instrumenting both the models and the software they represent, and comparing the execution traces produced by that instrumentation in both cases. To do this, we recorded events via instrumentation points in the kernel, middleware, and application layers using data streams [40]. We also added output to timed automata transitions corresponding to the middleware instrumentation points. We then collected traces from the execution of the models and of the software, and post-processed those traces to generate time-lines for comparison. We compared the two timelines in terms of (1) the sequence of events that occurred in each case, and (2) the time at which each event occurred. The models were realized and executed using IF 2.0 (with bug-fixes) on a 2.8GHz Pentium 4 with 2GB RAM running Enterprise Linux with a 2.6.9-22 kernel. All validation experiments were run on a 1.4GHz Pentium 3 with 1GB RAM and running Fedora 2 with a LibeRTOS [41] 2.6.12 kernel.

Figure 4 shows a short extract from the sequence of events generated by post-processing traces from model and actual executions of the example scenario described in Section 4.1. In that scenario, two clients each send a request to the same server and the server hosts two event handlers each processing the requests from one client. We logged the following events along with their time stamps - (1) a client sending a request, (2) the request arriving at the socket buffer on the server, (3) the upcall to the event handler, and (4) the receipt of reply from the event handler by the client. The sequence of events

```
0: BEFORE_CLIENT_SEND_REQUEST(2)
0: EVENT_SOCK_DEF_READABLE(4)
0: BEFORE_CLIENT_SEND_REQUEST(1)
0: EVENT_SOCK_DEF_READABLE(2)
0: HANDLE_INPUT_BEGIN(2)
25: EVENT_SOCK_DEF_READABLE(1)
25: AFTER_CLIENT_RECV_REPLY(1)
25: HANDLE_INPUT_BEGIN(4)
50: EVENT_SOCK_DEF_READABLE(3)
50: AFTER_CLIENT_RECV_REPLY(2)
50: BEFORE_CLIENT_SEND_REQUEST(2)
50: EVENT_SOCK_DEF_READABLE(4)
50: BEFORE_CLIENT_SEND_REQUEST(1)
50: EVENT_SOCK_DEF_READABLE(2)
50: HANDLE_INPUT_BEGIN(2)
75: EVENT_SOCK_DEF_READABLE(1)
75: AFTER_CLIENT_RECV_REPLY(1)
75: HANDLE_INPUT_BEGIN(4)
100: EVENT_SOCK_DEF_READABLE(3)
100: AFTER_CLIENT_RECV_REPLY(2)
```

```
0 : BEFORE_CLIENT_SEND_REQUEST(2)
0 : EVENT_SOCK_DEF_READABLE(4)
0 : BEFORE_CLIENT_SEND_REQUEST(1)
0 : EVENT_SOCK_DEF_READABLE(2)
0 : HANDLE_INPUT_BEGIN(2)
25 : EVENT_SOCK_DEF_READABLE(1)
25 : AFTER_CLIENT_RECV_REPLY(1)
25 : HANDLE_INPUT_BEGIN(4)
51 : EVENT_SOCK_DEF_READABLE(3)
51 : AFTER_CLIENT_RECV_REPLY(2)
51 : BEFORE_CLIENT_SEND_REQUEST(2)
51 : EVENT_SOCK_DEF_READABLE(4)
51 : BEFORE_CLIENT_SEND_REQUEST(1)
51 : EVENT_SOCK_DEF_READABLE(2)
51 : HANDLE_INPUT_BEGIN(2)
76 : EVENT_SOCK_DEF_READABLE(1)
76 : AFTER_CLIENT_RECV_REPLY(1)
77 : HANDLE_INPUT_BEGIN(4)
102 : EVENT_SOCK_DEF_READABLE(3)
102 : AFTER_CLIENT_RECV_REPLY(2)
```

Fig. 4. Comparison of timelines between model (left) and actual (right) executions

shows that that the model and actual executions are reasonably close both in terms of the order of events and the time at which they occur. However, one key difference between the model and actual execution traces is the execution time of event handler processing. During model execution, the progress of time is controlled by the model checker and unless it is specified explicitly (as we show later), there is no execution jitter. However, during actual software execution we recorded the execution jitter shown in Figure 5.

As part of our experiments, we ran 25 iterations of the above flow - *i.e.*, with the client sending a request and the event handler responding with a reply - in both model and actual execution. Based on the generated timeline traces, we then plotted the events generated against their timestamps for both model and actual executions to obtain the

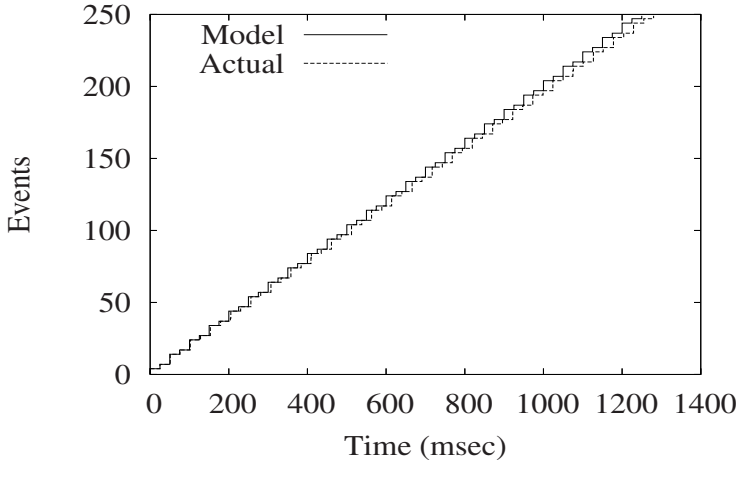

Fig. 5. Timeline comparison between model and actual implementation

graph shown in Figure 5. This figure shows visually how close the model is to the actual execution in terms of the times at which various events occur.

We also observed that the cumulative effect of jitter during actual execution becomes more pronounced as time progresses, which suggests that modeling the execution jitter as random noise would be a reasonable first approximation. To make the composed verification model reflect the actual system more closely, we added a jitter interval of 1ms to the event handler execution time in our model, which caused a significant increase in state space explored by the model checker. For exhaustive simulation, the model without jitter produced 2325 states, 2380 transitions, and took 1 second, whereas with jitter the model produced 217885 states, 225130 transitions, and took 181 seconds to explore exhaustively. The exhaustive simulation with jitter produced 288 traces of possible executions of the example scenario described in Section 4.1. To illustrate the range of variability added to the model by the jitter interval, we generated timelines for each of the 288 verification traces from the model with jitter, and then superimposed them in a new graph with the event numbers on the y-axis and timestamps on the x-axis. The result, shown in Figure 6, confirms that the model checker explores various combinations of execution jitter as it moves from state to state during exhaustive simulation.

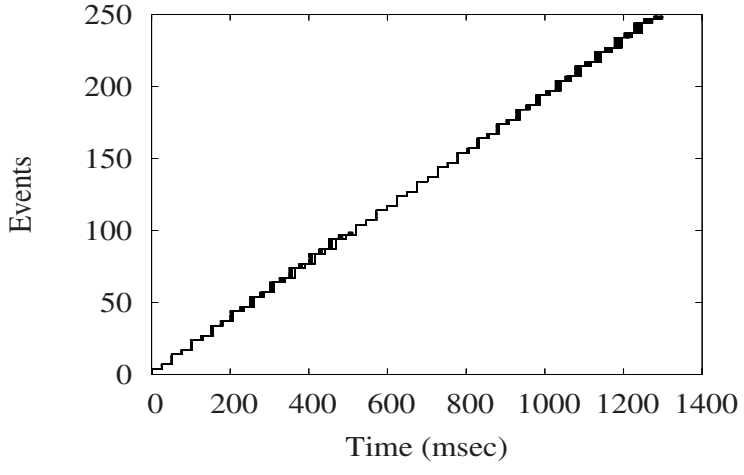

Fig. 6. Timeline with execution jitter for all paths explored by the model checker

6 Concluding Remarks

Observations and Lessons Learned. These results shown in Section 5 highlight several important principles for co-design of verification and validation in middleware models and software. First, the choice of instrumentation points is essential: discrepancies between where events are recorded in the software and in the models may skew the relationships between the timelines. While fixing this may be trivial in some cases, in other cases it may be necessary to expand a single model state into a more nuanced automaton to capture software semantics in more detail.

Second, observational equivalence must be evaluated subject to the constraints being checked. Relevant non-determinism in the software must be reflected in the model to ensure that the model is not over-constrained (and thus fails to check important cases) but irrelevant non-determinism should be eliminated from the model to reduce model checking complexity and to increase correspondence between the timeline representations for the model and software traces. Because the same model elements may be reused for a variety of software configurations, doing this in practice can be aided greatly by (1) the ability to turn model and software instrumentation points on and off together, through a set of common configuration descriptors, (2) post-processing predicates for both time-lines that filter out irrelevant variations while detecting relevant ones, and (3) classifiers that annotate the timelines to indicate regions where event timing and ordering correspond between them, and regions where event timing or ordering differ.

Third, the ability to conduct scientifically valid evaluations of the correspondence between verification and validation results, especially in the face of concurrency [42], requires that (1) the sources of non-determinism in the model be the same as those in the software, (2) the equivalence classification of traces from the model be the same as the equivalence classification of traces from the software, (3) for every specific trace generated from the model, a trace in the same equivalence class must be generated by the software experiments, and vice versa, and (4) the likelihood of generating a particular trace from the model or the software should be appropriate even if the generation of every trace is impractical or intractable. The last two points highlight a very important relationship between (1) exerting more control (e.g., through scheduling [2]) over when events occur and in what order to (re)produce specific scenarios, and (2) allowing a wider variety of scenarios to be explored to avoid repeated verification or validation of essentially equivalent scenarios.

Open Problems and Future Work. Several open problems will shape our future work on model-based verification and validation of reusable software frameworks. First, the need for automated instrumentation of both models and software *with respect to particular constraints to be checked* motivates the development of automated analysis and aspect weaving techniques. Second, the need for round-trip co-design and engineering of models and software is demonstrated by our results in Section 5, which emphasizes the need to develop integrated tools for software design, implementation, verification, and validation, into which different sets of reusable software building blocks and fine grained formal models for those building blocks can both be incorporated. Third, the ability to configure schedulers and other means of controlling software execution at run-time must also be integrated within these software tools, in terms of both models and software implementations.

References

1. Subramonian, V., Xing, G., Gill, C., Lu, C., Cytron, R.: Middleware specialization for memory-constrained networked embedded systems. In: Proceedings of 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS) (2004)
2. Aswathanarayana, T., Subramonian, V., Niehaus, D., Gill, C.: Design and performance of configurable endsystem scheduling mechanisms. In: Proceedings of 11th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS) (2005)

3. Sanchez, C., Sipma, H.B., Manna, Z., Subramonian, V., Gill, C.: On Efficient Distributed Deadlock Avoidance for Real-time and Embedded Systems. In: 20^{th} IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006) (2006)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
5. Subramonian, V., Gill, C., Sanchez, C., Sipma, H.B.: Composable Models for Timing and Liveness Analysis in Distributed Real-time Embedded Systems Middleware. Technical Report WUCSE-2005-54, Computer Science and Engineering Department, Washington University in St.Louis (2005)
6. Madl, G., Abdelwahed, S., Schmidt, D.C.: Verifying distributed real-time properties of embedded systems via graph transformations and model checking. International Journal of Time-Critical Computing Systems (2005)
7. Madl, G., Abdelwahed, S.: Model-based analysis of distributed real-time embedded system composition. In: EMSOFT 2005: Proceedings of the 5th ACM international conference on Embedded software, pp. 371–374. ACM Press, New York (2005)
8. Madl, G., Abdelwahed, S., Karsai, G.: Automatic Verification of Component-Based Real-time CORBA Applications. In: The 25th IEEE Real-time Systems Symposium (RTSS 2004), Lisbon, Portugal (2004)
9. Karsai, G., Neema, S., Bakay, A., Ledeczi, A., Shi, F., Gokhale, A.: A Model-based Front-end to ACE/TAO: The Embedded System Modeling Language. In: Proceedings of the Second Annual TAO Workshop, Arlington, VA (2002)
10. Marques, R.S., Simonot-Lion, F.: Guidelines for the development of a communication middleware for automotive applications. In: Proceedings of the 3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER3 2005) (2005)
11. Marques, R.S., Simonot-Lion, F.: Design-Patterns based development of an automotive middleware. In: Proceedings of the 6th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2005) (2005)
12. Gu, Z., Wang, S., Shin, K.G.: Issues in Mapping from UML Real-Time Profile to OSEK API. In: Proc. Workshop on Specification and Validation of UML models for Real-Time and Embedded Systems (SVERTS 2003) (2003)
13. Object Management Group: UML Profile for Schedulability. Final Draft OMG Document ptc/03-03-02 edn. (2003)
14. AUTomotive Open System ARchitecture: AUTOSAR (2005), `www.autosar.org`
15. Institute for Software Integrated Systems: The ADAPTIVE Communication Environment (ACE) (Vanderbilt University), `www.dre.vanderbilt.edu/ACE/`
16. Gill, C.D., Levine, D.L., Schmidt, D.C.: The Design and Performance of a Real-time CORBA Scheduling Service. Real-time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-time Middleware 20(2) (2001)
17. Institute for Software Integrated Systems: The ACE ORB (TAO) (Vanderbilt University), `www.dre.vanderbilt.edu/TAO/`
18. Group, D.: nORB - Special Purpose Middleware for Networked Embedded Systems (2005), `deuce.doc.wustl.edu/nORB/`
19. Subramonian, V., Xing, G., Gill, C., Lu, C., Cytron, R.: Middleware Specialization for Memory-Constrained Networked Embedded Systems. In: Proceedings of the 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), Toronto, Canada, IEEE, Los Alamitos (2004)
20. Subramonian, V., Gill, C.: Middleware Design and Implementation for Networked Embedded Systems. In: Zurawski, R. (ed.) Embedded Systems Handbook, CRC Press, Boca Raton (2006)
21. Institute for Software Integrated Systems: Component-Integrated ACE ORB (CIAO) (Vanderbilt University), `www.dre.vanderbilt.edu/CIAO/`

22. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, pp. 93–104. ACM Press, New York (2000)
23. OSEK Consortium: OSEK/VDX communication specification (2004), http://www.osek-vdx.org
24. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, New York, vol. 2. Wiley & Sons, Chichester (2000)
25. Subramonian, V., Gill, C., Sanchez, C., Sipma, H.B.: Composable timed automata models for real-time embedded systems middleware. Technical Report WUCSE-2005-29, Computer Science and Engineering Department, Washington University in St.Louis (2005)
26. Wang, N., Gill, C., Schmidt, D.C., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: OTM 2004. LNCS, vol. 3291, pp. 1520–1537. Springer-Verlag, Heidelberg (2004)
27. Sharp, D.C., Roll, W.C.: Model-Based Integration of Reusable Component-Based Avionics System. In: Proc. of the Workshop on Model-Driven Embedded Systems in RTAS (2003)
28. Object Management Group: The Common Object Request Broker: Architecture and Specification. 3.0.2 edn. (2002)
29. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. In: Proceedings of the 11th Real-time Technology and Application Symposium (RTAS 2005), San Francisco, CA, pp. 190–199. IEEE, Los Alamitos (2005)
30. Sanchez, C., Sipma, H.B., Subramonian, V., Gill, C., Manna, Z.: Thread Allocation Protocols for Distributed Real-Time and Embedded Systems. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 159–173. Springer, Heidelberg (2005)
31. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
32. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF Toolset. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004)
33. Robby, Dwyer, M., Hatcliff, J.: Bogor: An Extensible and Highly-Modular Model Checking Framework. In: In the Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), Helsinki, Finland, ACM, New York (2003)
34. Holtzman, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
35. Bozga, M., Graf, S., Ober, I., Mounier, L.: IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)
36. Bozga, M., Fernandez, J.C., Ghirvu, L., Graf, S., Krimm, J.P., Mounier, L.: IF: A Validation Environment for Timed Asynchronous Systems. In: Proceedings of CAV 2000 (2000)
37. Bornot, S., Sifakis, J., Tripakis, S.: Modeling Urgency in Timed Systems. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 103–129. Springer, Heidelberg (1998)
38. Subramonian, V.: Timed Automata Models for Principled Composition of Middleware. PhD thesis, Washington University in St. Louis, Computer Science and Engineering Department Technical Report WUCSE-2006-23 (2006)
39. Subramonian, V., Gill, C., Sánchez, C., Sipma, H.B.: Reusable models for timing and liveness analysis of middleware for distributed real-time and embedded systems. In: Sixth ACM/IEEE International Conference on Embedded Software (EMSOFT 2006), pp. 252–261 (2006)

40. Buchanan, B., Niehaus, D., Dhandapani, D., Menon, R., Sheth, S., Wijata, Y., House, S.: The data stream kernel interface. Technical Report ITTC-FY98-TR11510-04, Information and Telecommunication Technology Center, University of Kansas (1998)
41. Linutronix: LibeRTOS (2004), http://www.linutronix.de/linutronix/e/libertos.html
42. Niehaus, D., James, J., Gill, C.: Closing the Programmer's Universe: A Pattern Language for Reproducibility in Concurrent Programming Environments. In: Pattern Languages of Programs Conference, Allerton Park, IL (2003)