Model-Based Run-Time Monitoring of End-to-End Deadlines

Jaswinder Ahluwalia, Ingolf H. Krüger, Walter Phillips
University of California, San Diego
Department of Computer Science
9500 Gilman Dr., La Jolla, CA 92093, USA
{jas, ikrueger, whphilli}@cs.ucsd.edu

Michael Meisinger
Institut für Informatik
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
meisinge@in.tum.de

ABSTRACT

The correct interplay among components in a distributed, reactive system is a crucial development task, particularly for embedded systems such as those in the automotive domain. Model-based development is a promising means for capturing key structural and behavioral requirements *before* implementing code. Current development approaches focus on components as the central development entity, leaving component integration as a separate and error-prone task in later stages of the system development process. This approach is particularly problematic in the area of Quality-of-Service properties that are inherently end-to-end. We address this problem by using a model where system *functions*, not components implementing them, are central from the early phases of requirements capture through implementation. We develop a domain model for system functions (or services) based on interaction patterns; this model captures deadline specifications ranging from individual messages to entire services. Using a combination of modeling tools and code-generators for the RT CORBA platform, we provide an experimentation platform for monitoring these specified deadlines in executable specifications.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.11 [**Software Engineering**]: Software Architectures; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Design, Verification

Keywords

Components, Services, Service Engineering, Quality of Service, Code Generation, Runtime Monitoring, RT CORBA

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

1. INTRODUCTION

Distributed, reactive, embedded systems are notoriously difficult to develop. The shift from monolithic to highly networked, heterogeneous, interactive systems has led to a dramatic increase in both development and system complexity. At the same time the demands for safety, reliability, and other qualitative attributes have increased across application domains. This can be observed clearly in the automotive domain where the situation is aggravated further by demanding time-to-market requirements, short development cycles, and rapid change of technological infrastructures, customer demands and product lines.

The major challenge in developing such systems is to manage the complexity induced by the distribution and interaction of the corresponding components. Model-based development techniques and notations have emerged as an approach to dealing with this complexity, in particular during the analysis, specification and design phases of the development process; popular examples are the UML [48], SysML [45], ROOM [40] and SDL [9]. Each of these examples proposes managing the complexity of software development by separating the two major modeling concerns: system structure and system behavior. Other approaches work with generic models of a specific target application domain [29], and provide modeling environments and tools for model validation [4] and transformation [42] to add value beyond graphical modeling.

However, the center of concern in model-based design has so far been individual components rather than their interplay; yet many important properties, especially of distributed embedded systems, are non-local in nature: Quality-of-Service (QoS) properties such as throughput, execution time and memory constraints are examples of this observation. Typically, such properties are best associated with multiple components carrying out a specific task.

Contributions and Outline: In this paper, we demonstrate how to specify and monitor end-to-end QoS properties – in particular, interaction deadlines – using models that capture the cross-cutting interaction aspects of distributed, reactive systems. This places the *functions* (or features, services) provided by the system – rather than the components implementing them – in the center of the development process. We show how to create an infrastructure for simulation and validation based on RT CORBA [31]; this infrastructure includes a distributed monitoring facility that currently observes interaction deadlines during runtime. In this paper, we focus on monitoring deadline violations in ex-

ecutable specifications; this is an important capability used, for instance, to check QoS properties during conformance testing of supplier-provided components. Monitoring execution is an important complement to system testing and formal validation techniques, such as model checking and theorem proving.

The remainder of this document is structured as follows. In Section 2, we provide a domain model for interactions and QoS constraints for our interaction-based development approach. This domain model shows the relationships between the modeling entities underpinning our notion of system functions; it also serves as the basis for tool development. Section 3 shows how end-to-end interaction deadlines are modeled by using an excerpt of a central locking system as an automotive example. We describe the relationship between the interaction model and a corresponding prototypic tool chain, and how it can be used to generate RT CORBA prototypes where real-time constraints are monitored. In Section 4, we discuss our approach in the context of related work. Section 5 contains our conclusions and outlook.

2. INTERACTION-BASED DEVELOPMENT FOR QOS PROPERTIES

In this section we first give our rationale for the use of interactions as a key ingredient in the software development process of embedded reactive systems. Then, we discuss a domain model that clarifies the relationships between system functions, roles, interactions and components, and shows how real-time constraints can be modeled. This domain model is the basis for the tool chain we describe in detail in Section 3.3; it provides a structure for the data format for model interchange between the elements of the tool chain. In fact, our model can easily serve as a metamodel for tools such as GME [11] to exploit generic model transformations as described in [42].

2.1 Services in the Automotive Domain

The automotive domain is particularly attractive to explore the ideas presented in this text. Typical luxury cars have up to 80 distinct microprocessors (embedded within Electronic Control Units, ECUs) implementing hundreds to thousands of software-enabled vehicular functions. The high degree of dependencies and resulting interaction between these functions is observed to be a limiting factor in the development and implementation of new functionality [6, 12]. Triggered by the success of service-oriented development approaches in telecommunications and business systems (most notably under the keyword "web services"), car manufacturers are exploring service-oriented software design also for the vehicle [30]. The work we present here provides a basis for systematic development steps into this direction; it ties in with standardization efforts based on explicit or implicit feature or service notions such as AMI-C [2], EAST-EAA [8] and AutoSAR [5]. For further information on the emerging field of service-oriented development in the automotive domain, we refer the reader to [3].

2.2 Interaction-Based Development of End-to-End QoS Properties

Our approach is based on the observation that central aspects of system behavior involve the coordination or interaction of multiple components [25, 26, 22]. A system function (or service) in our approach, is defined as the collaboration among a set of components to accomplish a particular task. We have shown in [23] that this interaction-based notion is suitable for modeling of automotive software and how it can be supported by tools for interaction-based modeling, validation and code generation [22].

The principle development steps in interaction-based development are *service elicitation* and *architecture mapping*. During service elicitation we perform the following steps: (1) structure and determine the principal use cases of the system under consideration; (2) from the use cases, determine the key roles (also called *actors*); (3) determine the interactions among these roles required to execute the service – we use extended Message Sequence Charts (MSCs, [20]) to specify these interaction patterns. During architecture mapping we perform the following steps: (4) specify the target component configuration (deployment architecture), such as the layout of the ECUs and communication buses in the vehicle; (5) associate the roles elicited in step 2 with the deployment components. This approach supports general mappings of multiple roles to the same deployment component as needed, for instance, for model-driven architecture [33] and product-line architectures [36]. Often, however, models assume a one-to-one relationship between roles and deployment components, i.e. there is exactly one deployment component per role; for reasons of brevity we will work with this assumption in the remainder of this text as well. This development process is intended to be exercised in a highly iterative, incremental fashion.

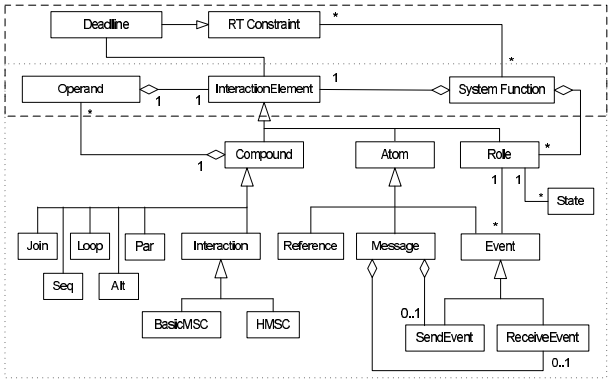

Figure 1: Interaction Domain Model

In [26] we have introduced an architecture definition language (ADL) supporting both steps of the interaction-based development process. This ADL offers system services and their defining interaction patterns as first-class modeling elements for software architectures. Instead of repeating the syntactic elements used in the ADL, we show the underlying domain model in Fig. 1. We use a class diagram as notation to define the main entities and their relationships that we need to model interactions. The domain model abstracts from the actual language syntax of the ADL given in [26]. The dotted area in Fig. 1 contains the model of system services (functions) defined as interaction patterns: a System Function consists of an interaction specification (an InteractionElement) among a set of Roles. An interaction specification is either simple (an Atom), or composed from an operator and its operands. Messages and References (to other interaction specifications) are examples of simple interac-

tion specifications. Composite interaction specifications are labeled with an operator and have a set of operands. Examples of operators are sequential and parallel composition, alternatives, loops, as well as joins. The *join* of two interaction specifications synchronizes its operands on shared interactions while interleaving all others. As we have described in [24], join supports the composition of *overlapping* system functions. Both Basic and High-Level MSCs are also interaction specifications.

The interaction-based system model described above provides an anchor-point for specifying QoS properties (dashed area in Fig. 1). With each system function, we associate a set of real-time constraints; each real-time constraint can relate any set of InteractionElements. As a special case in this paper, we consider deadlines associated with InteractionElements. This allows us to model deadlines for entire functions (services), but also for individual messages within interaction patterns.

An interaction-based development process supporting specification and validation of end-to-end QoS properties now emerges as follows from the one described above: Perform all development steps as described above; while determining the interactions among the roles required to execute a system function in step (3), identify QoS constraints and associate them with the appropriate InteractionElements. The corresponding model records the key interaction patterns *together* with their QoS constraints; the augmented notion of interactions we have introduced thus captures QoS constraints as cross-cutting system aspects.

In the following section we demonstrate the application of this idea to an automotive example; we also describe the tool support we have built to support the modeling and run-time validation of end-to-end deadlines using the model introduced here.

3. MODEL-BASED RUN-TIME MONITORING OF QOS PROPERTIES

The model developed in the previous section not only allows us to capture QoS properties; it provides us with the ability to *monitor* whether the specified properties are observed during runtime in actual or simulated deployments. This is important, for instance, for conformance testing of supplier-provided components in a complete system environment. It allows the supplier to test compliance of components with the rest of the system during development; it also provides the system integrator with an environment to integrate and test supplied, reengineered, or evolved components against the specification. Both capabilities are highly relevant and demanded in situations where time-to-market as well as outsourcing pieces of development dictate systems engineering, as currently observed in the automotive industry [3]. In this section we present an approach utilizing the RT CORBA infrastructure to provide a non-invasive approach to monitoring end-to-end RT constraints at runtime.

The basic idea is to distribute the QoS specifications associated with each system function to all the deployment components involved in executing the function. Each component monitors whether its functions continue to operate within the specified QoS properties; a flag is raised if any QoS property is violated. These flags are communicated for all subsequent interactions involved in a particular function. A central monitor component collects flagged interactions and alerts the developer about the violated properties.

Before we elaborate on this idea, we introduce the Central Locking System (CLS) as an example from the automotive domain to illustrate our interaction-based development approach.

3.1 Example: Central Locking System (CLS)

The Central Locking System (CLS) is a well-studied and documented example of a vehicle functionality. The CLS integrates a multitude of separate subsystems in the vehicle, ranging from safety critical ones (motor control and crash sensors) to comfort functions (automatic seat positioning and tuner presets in luxury vehicles). For reasons of brevity, we present a simplified and abstract adaptation of the CLS. We direct the reader to [30, 23] for a more comprehensive description. Here, we focus on two specific use cases during the *unlocking* of the vehicle: *operation of locks and signaling* and *transfer driver ID*.

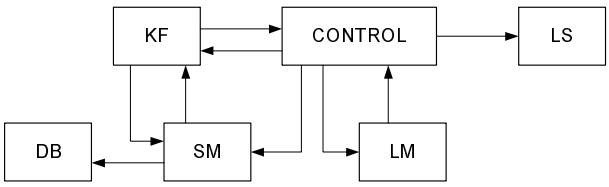

Figure 2: CLS Domain Model

Figure 2 describes the roles we have identified for our CLS. The CONTROL is the command center of the CLS. The Lock Manger (LM) abstracts away the details of performing the physical lock/unlock operations by providing a simplified interface. Similarly, a vehicle contains several interior and exterior lighting mechanisms, which are abstracted away by the entity Lighting System (LS). The Security Module (SM) handles security-related tasks of the vehicle, such as alarms, vehicle tracking and authentication. A Key Fob (KF) is the user interface for externally locking or unlocking the vehicle. Finally, the Database (DB) represents a persistent storage component, which logs identifications associated with particular users and/or key fobs.

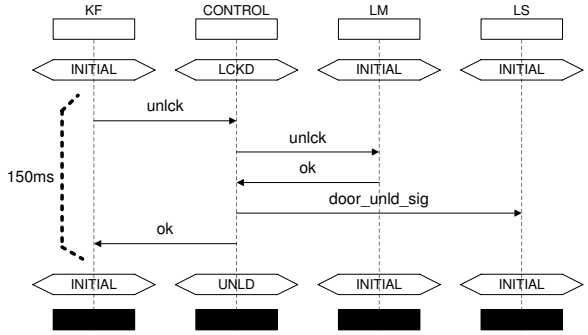

Figure 3: UNLK-1: Operation of Locks & Signaling

Figure 3 shows the *operation of locks and signaling* function. The graphical syntax we use is derived from MSCs as described in [20, 24]. Upon receipt of the *unlck* message from KF, CONTROL issues an *unlck* message to LM. Once LM acknowledges this with an *ok* message, CONTROL

requests signaling of the unlocking from LM by means of a *door_unld_sig* message. Once it has issued this message, CONTROL sends an *ok* message back to KF. The *transfer driver ID* function is also triggered by the *unlck* message from KF to CONTROL. The corresponding interaction pattern is shown in Fig. 4.

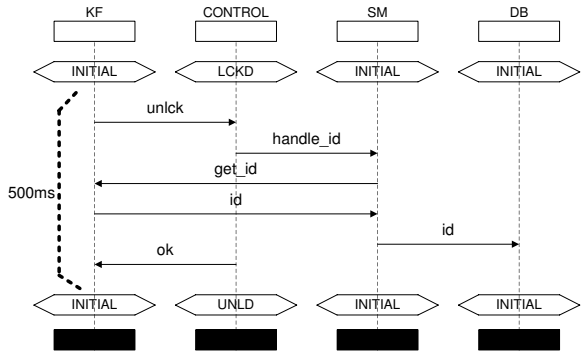

Figure 4: UNLK-2: Transfer Driver ID

The MSCs of Figures 3 and 4 are both augmented with interaction deadlines, indicated by means of a labeled dashed line. The *operation of locks and signaling* function has a deadline of 150 ms. This means that the vehicle must be unlocked and the signaling must have occurred within 150 ms according to the interaction specification. The *transfer driver ID* function has a deadline of 500 ms. These two operations are integral parts of the full unlocking use case; we can observe that they are overlapping (i.e. they share common messages).

3.2 Composition of QoS properties

Our interaction model allows us to compose overlapping functions by means of the join operator introduced in Sec. 2.2. The join operation combines two distinct, but overlapping, functions (represented as MSCs) into a single function. A combination of functions in our specification has to follow the constraints imposed by each separate function. In our example, the deadline for the joined function (called *unlocking*) will become 150 ms, because the common messages on which the two MSCs were joined upon were the start and end messages of both functions. Note also that these deadlines relate the otherwise independent *id* and *ok* messages seen in Fig. 4.

Our interaction model defines several operators to compose interactions (MSCs or parts thereof) to more complex interactions. Available operators are Seq, Par, Alt, Loop, and Join as introduced in [20]. We discussed the join operator above, where the result retains the minimum deadline of all operands. We currently consider only the common case where start and end messages are overlapping. A similar logic is used for parallel composition (Par). Sequential composition (Seq) simply adds the deadlines of the operands. For alternative interactions (Alt), the correct execution path will be chosen at runtime and we can retain the original deadlines. For Loops, the deadline is evaluated for each cycle of the loop separately. If the entire loop should have a fixed deadline independent of the number of cycles, this deadline must be specified manually in the composed function.

More precisely, we apply a bottom-up scheme for interaction composition. Deadlines can be applied to basic interactions. We can, for instance, define a deadline for a single message or a message sequence. For each composition operation, we apply defined rules that constrain the deadlines of the composite interactions. For instance, sequential composition leads to an addition of the operands' deadlines, loops to a multiplication, parallel and join composition to the selection of the minimum deadline. All deadlines can be tightened manually. A less restrictive composition alternative (for instance than applying the minimum constraint for join composition) would be to only consider a newly defined deadline for the composite. Doing so would allow the modeler to provide a different interpretation for the more complex composite function – it can be more than the sum of its parts. However, this may not yield a true refinement of the specification in the bottom-up sense, because the composite may not fulfill all QoS properties of the composed interactions anymore. Practical considerations would determine the concrete composition scheme used. We chose the composition variant that maintains all properties of basic interactions and allows for methodological refinement. We are aware that this is more restrictive to the modeler and requires more frequent modifications or refactorings of the specification.

In terms of methodology, we can also apply top-down refinement of deadlines, while still fulfilling all properties of bottom-up composition as described above. Starting from deadlines for entire functions, we allow the modeler to provide specific deadlines to parts of the interaction, as long as the overall deadlines are still satisfiable.

3.3 Tool Chain

In order to demonstrate the concept of interaction-based development, we have designed and implemented a prototypic tool chain. The primary purpose of this tool chain is to illustrate the complete development cycle, from the initial modeling phase to execution on a real system. As shown in Fig. 5, the tool chain consists of three main elements. First, the modeling and specification tool itself (M2Code) provides the means to specify roles, interactions patterns and system services. Second, we apply tools for model simulation, verification and testing of component configurations (AutoFocus/MSCCheck). Third, we have developed a tool for transforming the abstract specification model of M2Code or

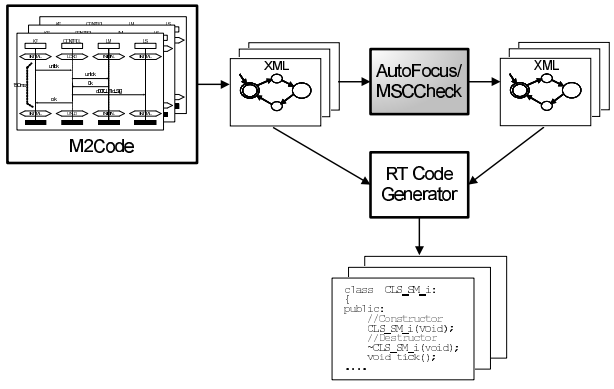

Figure 5: Tool Chain for Interaction-Based Development

AutoFocus into executable code for the RT CORBA middleware platform (Code Generator). We refer the reader to [22] for a more in depth look at the tool chain.

M2Code allows modelers to specify system functions (services) and QoS properties using MSCs, based on the domain model introduced in Sec. 2.2. The real-time constraints are specified on MSCs as shown in Figures 3 and 4. M2Code applies the logic described above to yield deadlines for composed interactions when applying MSC operators.

M2Code translates MSC specifications into component structures and state machines for the individual components [21, 22]. This abstract component specification is stored in an XML file, similar in format and content to the XML output of AutoFocus. This common XML specification file format allows data to move between the various tools. Fig. 6 shows how the XML output of M2Code is extended to record the QoS constraints for each system service using *unlocking* as an example:

```
<QoS>
 <services>
  <service name="UNLCK">
   <messages>
    <message source="KF" dest="CONTROL" deadline="150">UNLOCK</message>
    <message source="CONTROL" dest="LM" deadline="150">UNLOCK</message>
    <message source="LM" dest="CONTROL" deadline="150">OK</message>
    <message source="CONTROL" dest="LS" deadline="150">DOOR_UNLD_SIG</message>
    <message source="CONTROL" dest="KF" deadline="150">OK</message>
    <message source="CONTROL" dest="SM" deadline="150">HANDLE_ID</message>
    <message source="SM" dest="KF" deadline="150">GET_ID</message>
    <message source="KF" dest="SM" deadline="150">ID</message>
    <message source="SM" dest="DB" deadline="150">ID</message>
   </messages>
  </service>
 </services>
</QoS>
```

Figure 6: QoS Portion of XML Input

AutoFocus [4] and MSCCheck use the specification output of M2Code to simulate and (through model-checking algorithms) verify those behavioral aspects of the model that are *not* related to QoS.

To accommodate analysis of QoS properties, we utilize the infrastructure of RT CORBA [31] for simulation and validation. In contrast to the simulation tool found in AutoFocus, the RT CORBA based runtime system we have implemented provides monitoring and validation mechanisms for *both* logical flow *and* real-time property validation. To bridge the gap between the abstract XML specification and executable code, we have developed a code generator and a runtime system based on the RT CORBA platform. RT CORBA was chosen for its robust capabilities of specifying and implementing real-time properties. Our runtime system was intentionally kept simple and straightforward by using the Real-Time Event Service (RTES) messaging facility. The RTES provides the fundamental abstraction for asynchronous message passing, allowing each component to operate independently. We also make use of hooks the RTES provides to incorporate a real-time scheduler. The simplicity of a message-based communication system promotes future deployment to other message-based runtime systems such as a CAN (Controller Area Network) bus. Messaging and scheduling mechanisms would have to be reproduced, but fundamentally the system concept is portable.

The code generator supports two execution models: AutoFocus and asynchronous. Intuitively, the AutoFocus execution model operates in a time-synchronous mode; messages are exchanged via one-place buffers between components. Each component waits for inputs to arrive on *all* of its input buffers, then executes an enabled transition of its associated state machine, and finally writes to all of its output buffers; this scheme is further described in [13]. The AutoFocus model has the benefit that it is well-supported by the validation and verification tool itself, namely AutoFocus. This, however, results in "lock-step" executions of the components that have to be coordinated by an abundance of control messages on the communication medium.

To better support the treatment of automotive environments with QoS constraints we have developed an asynchronous execution model that is reactive in nature. In this model, upon receipt of a message, each component immediately executes an enabled transition of its automaton and sends output on *only* the appropriate ports. This model eliminates undesirable "waiting" as well as network flooding by control messages. The drawback of this model is that we lose AutoFocus as a validation tool. However, we are currently developing a tool called MSCCheck, which provides model-checking capabilities also under the asynchronous execution model.

The generated code employs RT CORBA's Time Service [32]. The Time Service provides a distributed, synchronized, global clock to all components in the system. Mechanisms for globally synchronized clocks exist on many embedded platforms and so the runtime system is, again, fundamentally portable in concept. This is critical for having components monitor QoS constraints in a distributed fashion without the need of an invasive entity like a central scheduler. We will discuss this further in the next section.

3.4 Deadline Monitoring

The broadcast/multicast nature of the Real-Time Event Service (RTES) allows for external monitoring of timing properties on the event channel. One approach to monitoring would be to register a single globally aware entity, a *central* monitor. This entity would serve a multitude of functions. It would include an internal timer to check for message and interaction deadlines; it would include internal state machines to associate incoming messages with the system functions, and it would act as a single point of user interaction. Although seemingly simple and obvious, this approach has an inherent flaw: in a heterogeneous, distributed environment with network delays between components, the monitor may receive messages *before* the specified recipient. Thus, from the monitor's perspective, a deadline may not have been violated; yet, from the component's perspective, the deadline may have passed before receiving the actual message.

We therefore introduce a fresh approach where parts of the functionality of the monitor are pushed out into the components. Specifically, all components contain a monitoring element, *component monitor (CMonitor)*, which checks whether or not interaction deadlines have occurred.

In our approach, each message sent via RTES contains interaction deadlines, calculated based on message and/or system function specifications. Upon receipt of a message, the CMonitor of the receiving component obtains the current time from the Time Service and checks it against the message's carried interaction deadlines. If such a deadline has been reached, the CMonitor sends the original message back on to the event channel with a flag raised. The *monitor* described below then displays this message, along with its

timing and system function information, to the user. The user is then able to verify which messages were not able to meet the deadline, by how much they were in excess of the deadline, and which functions the exceeded messages belonged to. Messages also carry function identifiers associated with the deadlines, to provide all necessary monitoring information to the monitor. This concept of local monitoring generalizes to other QoS properties, such as memory constraints and throughput, as well.

The remaining monitoring functionality, providing a point of user interaction and global system information, is established as a single entity, the *monitor*. The monitor registers itself with the RTES to receive copies of all messages placed on the Event Channel. This allows the monitor to observe the interactions occurring in the system from a global perspective. Alternatively, components register themselves with the RTES to receive copies of messages from only those components they are allowed to interact with, as specified by the system function requirements. In Fig. 7, we show a deployment picture of the CLS, which includes the monitor and all CMonitors. Note that the internal wiring among components remains the same as in Fig. 2 because of how components registered themselves with the RTES. Also note that since the monitor registers itself as a global entity, it has a global view of the entire event channel. Additionally, the monitor has the capability of injecting events on behalf of components into the event channel, which is useful for triggering a system function or a set of interactions. This is a helpful tool for testing how components react to specific messages.

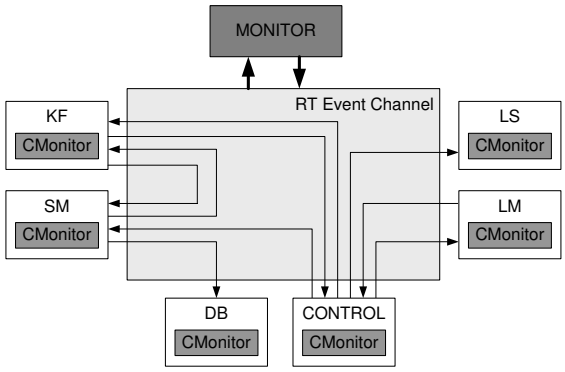

Figure 7: CLS Deployment Picture with Monitor

Let us examine a run-time execution instance of the *unlocking* function we have described in Sec. 3.1. We begin by utilizing the inject feature of the monitor. First, we choose a message to be injected into the system. In our case, we choose UNLCK. Next, source and destination components must be selected to let the system know which component the message should be sent to, and on whose behalf it should be sent. We choose to send the message to CONTROL on behalf of the KF. This simulates a user pressing the unlock button on a key fob device. Finally, we press the inject button in the monitor and observe the interactions among components as well as a list of incoming *alarms*. The interactions adhere to those specified in Fig. 3 and 4. The alarms are messages that let us know which messages did not meet their deadlines and by how much they were off. In this specific execution instance, the final three messages of the function exceeded the 150 ms deadline. The first message exceeded it by 21 ms, the second by 37 ms, and the final by 53 ms. In total, the function took 203 ms to complete. This entire process can be observed by viewing the unlock function as captured by our monitor in Fig. 8. As system designers, we now know that the function has to be refactored, (whether from a modeling, implementation, or a hardware perspective) in order to meet the 150 ms deadline. This information is invaluable because we have determined such needs well before an actual deployment.

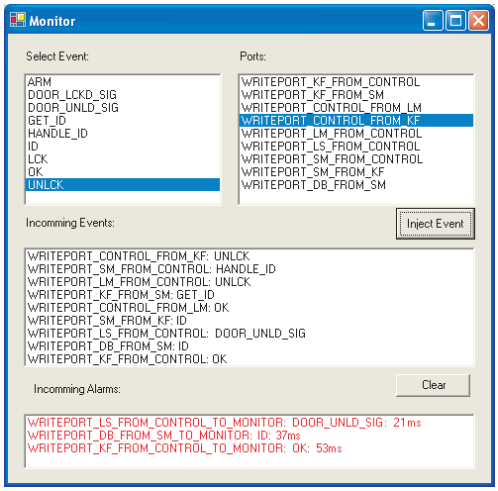

Figure 8: Monitor During Unlock Execution

3.5 Towards Enforcing QoS Properties

Our approach of monitoring interaction deadlines provides developers and integrators of embedded systems with a powerful means to check QoS properties using executable specifications against interaction-based system specifications. The quality of the results – in terms of repeatability and determinism – depends on the properties of the deployment environment and configuration of the middleware.

RT CORBA provides several mechanisms and policies that allow us to consider end-to-end deadlines and adjust runtime parameters to create an environment which can be similar to a specific target environment. We utilize the RT Event Service; it provides us with hooks into a RT Scheduler which we have incorporated. We employ a dynamic scheduling policy for individual messages known as Earliest-Deadline-First (EDF). All messages in a sequence receive the same deadlines as the functions they are a part of. For example, each message in the *unlock* service receives a deadline of 150 ms. This scheme ensures that all messages within a sequence will be treated equivalently; it enables us to deal with message sequences, not just individual messages. Thus, since functions are composed of message sequences, we provide means to enforce end-to-end deadlines.

At system startup, each component obtains a reference instance to the RT Scheduler. Before messages are sent out, the deadlines are looked up in a hash table for the functions that the message belongs to. The deadlines are then sent along with the message and the RT Scheduler dispatches messages according to the EDF policy described above.

If we can show that our executable specification meets all specified QoS requirements in an environment with guaran-

teed real-time properties such as message delivery time and scheduling policy, we know that the realized system running on a real-time operating system with the same properties will meet the QoS requirements as well.

3.6 Code Generator Design

We described an initial version of the RT CORBA code generator RTCGen, using a central scheduler for interaction coordination, in [22, 28]. To accommodate the multiple different execution models for the generated code, as well as to provision flexible integration of QoS monitoring and enforcement in the code generation process, we have redesigned RTCGen in the following way.

RTCGen takes in two forms of inputs to produce executable code for the RT CORBA middleware: a set of *templates* and an XML file obtained from M2Code; this XML file contains the system structure, state machines for component behavior as well as QoS constraints on a per-function basis. The templates are a set of files which are used by RTCGen in conjunction with data pulled and manipulated from the XML file to produce code. A template essentially contains three types of content: First, code that is independent from the model and will be copied unchanged into the output – this subsumes infrastructure tasks such as setup and configuration of the naming service and RTES. Second, XML-style *basic-tags* which represent information that is dependent on the model (such as the project name) – tags are dynamically replaced by the code generator with model specific content. Third, *loop-tags*, which contain basic-tags and iterate over certain entities (such as the list of components in the XML file). Using templates allows for great flexibility because they provide a means for easy manipulation of target code. For example, swapping out middlewares can be accomplished by altering existing templates or creating a new set of templates.

We have also developed a *framework* and a *common library* that may be utilized by generated components. The framework isolates from a large portion of the RT CORBA middleware. Communication mechanisms, real-time scheduling and timing instruments are handled within the framework. Providing an abstraction from the often intricate RT CORBA code decouples templates and generated code from the details of the underlying middleware. This results in cleaner and easier to maintain code. The common library contains project-specific code, declarations, utilities and data structures that are common to all components of a project. The modularity of the library keeps component templates and the code they produce as simple as possible.

4. DISCUSSION & RELATED WORK

In the following paragraphs we put the interaction-based development approach presented here into perspective by discussing it in the context of related work.

Services and Roles. Triggered by its success in the telecommunications domain [35, 37] the term service has become quite prevalent in the literature, especially in the context of "web services" [41]. So far, however, services have been used mainly as an *implementation* concept, not as a first-class *modeling* entity. Consequently, existing definitions for the term service capture only syntactic lists of operations upon which a client can call. These definitions are inadequate for a systematic treatment of services throughout the development process. This is especially true also for the UML [48] or SysML [45], which do not recognize services as separate modeling entities. In our approach, the interaction based service notion emerges as a cross-cutting modeling element regarding both system structure and behavior. In particular, we have established a decoupling between services (functions) as modeling elements and implementation infrastructures on top of which services can be implemented. We use a generalized notion of a system service (or feature, function) in our interaction-based modeling approach. In [25, 26, 23] we present our service modeling approach based on the modeling of role interactions [24]. It is related to the role concept introduced in [34] and the activities of [19]. While our service concept is based on interaction patterns, stressing the cross-cutting nature of services, the roles of [34] describe projections of such patterns onto individual components; to yield the overall picture the latter have to be recomposed into a global interaction specification. Activities of [19] capture global interaction properties as we do in our service definition; in contrast to our approach, however, [19] views activities as classes and roles as extensions to these classes.

Service vs. Component-Based Development. The component-based development approach [46] certainly has many advantages, including support for encapsulation, modularity, defining a unit of deployment, fault-containment, and many more. However, it falls short for cross-cutting aspects [15] including interaction patterns. In contrast, by establishing a decoupling between service modeling and deployment of the resulting services on top of component architectures, we allow for "late binding" between functionality and components. Services provide a level of abstraction higher than components because services hide the components that implement them. This induces a choice regarding the architecture on top of which the services are implemented.

Our approach allows for the system to be understood at the granularity of individual features instead of components. The ability to gracefully deal with faults, both predictable and unpredictable, is an important property of embedded systems. Although we have not elaborated on this topic here, our approach allows for a better understanding of failures that emerge from the interplay of multiple components; the component-based approach accounts for faults localized to individual components.

Quality of Service. Defining system functions (services) based on the notion of interaction patterns gives us an immediate handle at defining Quality-of-Service properties similar to the approach proposed in the UML Profile for Schedulability, Performance, and Time (UPSPT) [47]; in fact, our service definition supports more general QoS specifications than what is possible in UPSPT. The General Resource Model (GRM) consists of the following fundamental components: Resource, ResourceService, QoSCharacteristic, and Scenario. Resources can be described as components in a system that will provide services. Each Resource provides a set of ResourceServices. QoSCharacteristics are the Quality of Service requirements which can be applied to both Resources and ResourceServices. This limits specification of QoS properties on a rather fine level of granularity; basically they have to be broken down to the level of individual functions of executable components. Often, especially in the early stages of development, it is helpful to specify QoS properties that span multiple components or resources.

Our approach provides means to specify QoS properties of interaction-based scenarios on all levels of granularity, from an entire service down to a single interaction.

Specification and verification of QoS Properties. Many suggestions for expressing Quality-of-Service attributes of distributed, reactive, embedded systems (DREs) exist in the literature. Some are based on formal specification languages such as timed automata [1]. Timed automata are a successful approach for the specification and verification of real-time systems; they extend traditional labeled transition systems with clocks, i.e. real-valued variables that record the passage of time and influence how the system evolves. The work in [7] provides a real-time extension of the process algebra CSP inspired by timed automata to enable automatic verification of refinement relations between processes. Several model-checking approaches exist to verify real-time properties in system specifications using timed automata, cf. [43, 49].

The work in [27] introduces a composition analysis model for component-based embedded systems. QoS requirements are considered in analyzing the solution space using an evolutionary algorithm and to provide component selections and parameter settings for the system as a result. Similarly, [18] shows a model-driven development technique for middleware configurations on end-to-end real-time and embedded system quality of service. It helps to select middleware configuration parameters that satisfy key functional and QoS requirements of DREs. The work in [16] shows how to compute worst-case execution times (WCET) for functionality-oriented Matlab/Simulink models of DREs; it is possible to make use of WCET analysis while modeling on a higher level of abstraction. The authors make use of the code-generation capabilities of Matlab/Simulink to perform the WCET analysis on assembly/object code levels. The work in [38] presents an approach of run-time monitoring of formal specifications. This provides a capability to distribute the monitoring of specifications on multi-processor hardware platforms to meet practical time constraints. [38] describes techniques for distributing checks onto different processors and discusses error reporting and recovery in a multi-processor environment.

Our approach is based on a formal model of interactions that enables model-based verification techniques such as model-checking and WCET analysis. In this paper, however, we describe an approach to test generated executable components for the RT CORBA middleware using run-time monitoring. This is, for instance, very useful when performing compliance testing of supplier-provided components in a fully integrated environment as specified; it also demonstrates the flexibility of our approach to target different execution environments with RT CORBA being one example. The selection of the test environment certainly determines the significance of the results. The more guarantees a test environment provides and the more it resembles the real execution environment, the more significant the test results will be. We chose RT CORBA because it provides for us a useful trade-off between flexibility of use and quality of the results for our case studies and industry collaborations.

In this paper, we focused on timing deadlines as one example of Quality of Service properties that can be effectively be represented in interaction-based service specifications. Other QoS properties include, for instance, throughput, memory consumption, CPU usage, I/O load, energy consumption, transmission reliability etc. Specific QoS properties might require specific approaches to verify and test them; not all can be checked using run-time monitoring. Our model-based approach provides the basis for the specification of cross-cutting QoS properties on the system level, but also on the level of individual interactions or component activities. Verification approaches can be flexibly built upon these specifications. We provided an interaction and QoS specification domain model as a basis for efficient tool support of our methodology.

Metamodels and Domain Specific Modeling Languages. Metamodels are described as formal models or specifications of domain-specific modeling languages [29]. A metamodel permits the creation of models using only the semantics prescribed by the metamodel. This is the basis to build a generic tool suite that will eventually create a product-line solution [14]. It is argued by [29] that this reduces development costs because one can reuse existing model components in the design process. One example toolset that complements this strategy is known as the Generic Modeling Environment (GME) [14, 11]. The domain model we have presented for our interaction-based approach can be viewed as a metamodel in this sense. This metamodel could serve as an input to tools such as GME and be used to construct domain-specific models. This would be an endeavor that awards further exploration. However, we have chosen a more general approach using MS Visio. We feel that with the tool's pervasiveness and its graphical capabilities, coupled with our modifications, it serves as an excellent modeling tool for service- and interaction-oriented design.

Model-driven Development. Model-Driven Development (MDD) techniques and tools help to specify, analyze, optimize and verify distributed real-time and embedded systems. [39] shows that MDD techniques can significantly improve quality and productivity for such systems, in particular in deployment scenarios using QoS-enabled component-based middleware platforms. To that end, [39] introduces a platform independent component modeling language (PICML). AIRES [17] and VEST [44] are MDD analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. The tools enable the annotation of components from a pre-defined library with real-time properties and a subsequent mapping to hardware platforms where the analysis is performed.

Our service-based approach targets similar systems and adds the concepts of services and interactions as first-class modeling elements; furthermore, it remains independent of a certain component infrastructure or target hardware environment. Our model-driven approach is related to Model-Driven Architecture (MDA) [33], Model-Integrated Computing [14], aspect-oriented modeling (AOM) [10] and architecture centric software development (ACD) [48]; similar to MDA and ACD we also separate the software architecture into abstract and concrete models, as for instance shown in [26]. In contrast to the cited model-driven development approaches, however, we consider services and their defining interaction patterns as first-class modeling elements of all our models throughout the different development phases.

5. CONCLUSIONS AND OUTLOOK

The development of high-quality, distributed, reactive, embedded systems is still a daunting, error-prone task. As we have discussed in this text, this is especially true in the automotive domain, where the complexity induced by an ever increasing amount of software functions and corresponding interactions between these, has become a limiting factor to innovation.

In this text, we have presented an *interaction-based* development process as an alternative to traditional approaches with components at the center of the development process. Here, the *defining elements* for system functions are interaction patterns. This allows us to deal adequately with interactions among sets of distributed components, as well as with QoS properties associated with these interactions.

As foundation for systematic, model-based development of interaction-based systems we have introduced a domain model capturing the relationships between key entities of our approach: system functions, roles, interactions, components and QoS properties. System functions capture the interaction patterns among sets of roles. Roles (and the functions they are associated with) are mapped to component configurations to obtain an architecture for the system under consideration. QoS properties are associated with interaction specifications; this supports associating QoS with entire functions or individual messages of an interaction pattern.

We have also shown how to exploit this interaction model in monitoring the QoS properties (specifically, end-to-end interaction deadlines) during runtime within the RT CORBA middleware infrastructure. To that end we have presented a tool chain covering all phases of the development process we have introduced. The generated RT CORBA code is very useful as an executable specification that can be handed over to suppliers for runtime analysis of subsystems within context. Because of the flexible design of the code generator we can easily switch from one target platform (such as RT CORBA) to another (such as a CAN-BUS/OSGi-based platform for deployment). Together, these elements enable interaction-based service-oriented specifications of complex, distributed, reactive systems together with their QoS properties. The monitoring facility provides run-time analysis of the QoS properties.

We have shown an application of our approach by means of the Central Locking System (CLS), an example extracted from the automotive domain. This example demonstrates our interaction-based development process from the earliest stages (identifying use cases) to the later ones (developing deployment architectures) to run-time property monitoring. Finally, we have provided a discussion on several key issues in this realm.

Opportunities for Future work include the extension of the code generator to provide monitoring of QoS properties of different type. Refining the interaction/QoS domain model to include more detailed QoS properties, such as worst case execution times of local actions, or QoS levels is an example. This would allow us to apply more sophisticated scheduling and resource allocation algorithms towards an even better enforcement of QoS properties, and provide the basis for model-checking certain QoS properties. Interesting areas for future research are also to evaluate opportunities for service and architecture refinement provided by the domain model and development process we have presented here.

6. ACKNOWLEDGMENTS

Our work was partially supported by the UC Discovery Grant and the Industry-University Cooperative Research Program, as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2). Further funds were provided by the Deutsche Forschungsgemeinschaft (DFG) within the project *InServe*. We are grateful to Sabine Rittmann and Diwaker Gupta and to the anonymous reviewers for insightful comments.

7. REFERENCES

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

[2] AMI-C. `http://www.ami-c.org/publicspecrelease.asp`.

[3] Automotive Software Workshop San Diego 2004. `http://tharkun.ucsd.edu/aswsd/index.html`.

[4] AutoFocus. `http://autofocus.informatik.tu-muenchen.de/index-e.html`.

[5] AutoSAR. `http://www.autosar.org`.

[6] M. Broy. Automotive software engineering. In *Proceedings of the 25th international conference on Software engineering*, pages 719–720. IEEE Computer Society, 2003.

[7] S. Cattani and M. Kwiatkowska. A refinement-based process algebra for timed automata. *Formal Aspects of Computing*, 2005.

[8] EAST-EEA. `http://www.east-eea.net`.

[9] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL. Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1998.

[10] G. Georg, R. France, and I. Ray. Composing Aspect Models. In *Proceedings of the 4th AOSD Modeling With UML Workshop*, 2003.

[11] Generic Modeling Environment. `http://www.isis.vanderbilt.edu/projects/gme/`.

[12] K. Grimm. Software technology in an automotive company: major challenges. In *Proceedings of the 25th international conference on Software engineering*, pages 498–503. IEEE Computer Society, 2003.

[13] F. Huber and B. Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch*, pages 343–352. GMD Verlag (St. Augustin), 1997.

[14] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. In *Proceedings of IEEE January 2003*, 2003.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer Verlag.

[16] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, pages 31–41, 2002.

[17] S. Kodase, S. Wang, Z. Gu, and K. G. Shin.

Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-time Systems using Shared Buffers. In *Proceedings of the 9th IEEE Real-time/Embedded Technology and Applications Symposium (RTAS)*, 2003.

[18] A. S. Krishna, E. Turkay, A. Gokhale, and D. C. Schmidt. Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*, pages 180–189, 2005.

[19] B. Kristensen and D. May. Activities: Abstractions for Collective Behavior. In *ECOOP'96*, volume 1098 of *LNCS*, pages 472–501. Springer, 1996.

[20] I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[21] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[22] I. Krüger, D. Gupta, R. Mathew, P. Moorthy, W. Phillips, S. Rittmann, and J. Ahluwalia. Towards a Process and Tool-Chain for Service-Oriented Automotive Software Engineering. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS)*, 2004.

[23] I. Krüger, E. C. Nelson, and V. Prasad. Service-based Software Development for Automotive Applications. In *CONVERGENCE 2004*, 2004.

[24] I. H. Krüger. Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In M. Pezzè, editor, *FASE 2003*, volume 2621 of *LNCS*, pages 387–402. Springer, 2003.

[25] I. H. Krüger. Service Specification with MSCs and Roles. In *Proceedings of the IASTED International Conference on Software Engineering*, 2004.

[26] I. H. Krüger and R. Mathew. Systematic development and exploration of service-oriented software architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2004.

[27] H. Ma, D. Wang, F. Bastani, I.-L. Yen, and K. Cooper. A Model and Methodology for Composition QoS Analysis of Embedded Systems. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*, pages 56–65, 2005.

[28] O. Müller. Generating RT-CORBA Components from Service Specifications. Master's thesis, Technische Universität München, 2004.

[29] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-Based Design-Space Exploration and Model Synthesis. In *EMSOFT 2003*, pages 290–305, 2003.

[30] E. C. Nelson and K. V. Prasaad. Automotive Infotronics: An emerging domain for Service-Based Architecture. In I. H. Krüger, B. Schätz, M. Broy, and H. Hussmann, editors, *SBSE'03 Service-Based Software Engineering, Proceedings of the FM2003 Workshop*, Technical Report TUM-I0315, pages 3–14. Technische Universität München, 2003.

[31] Object Management Group: Real-time CORBA specification, 2002. `http://www.omg.org/technology/documents/index.htm`.

[32] Object Management Group: Time Service Specification. `http://www.telelogic.com/products/tau/index.cfm`.

[33] OMG Model Driven Architecture. `http://www.omg.org/mda`.

[34] B. Paech. A framework for interaction description with roles. Technical Report TUM-I9731, Technische Universität München, 1997.

[35] Parlay 3.0. `http://www.parlay.org/specs/index.asp`.

[36] K. Pohl and A. Reuys. Considering Variabilities during Component Selection in Product Family Development. In *Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4)*, volume 2290 of *LNCS*, pages 22–37. Springer, 2002.

[37] Sector Abbreviations and Definitions for a Telecommunication Thesaurus Oriented Database. Definition for the term "software service" at `http://www7.itu.int/sancho/detailsdefE.idc?id=11702`.

[38] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *Computer*, 26(3):32–41, 1993.

[39] D. C. Schmidt, K. Balasubramanian, A. S. Krishna, E. Turkay, and A. Gokhale. Model Driven Development for Distributed Real-time and Embedded Systems. In S. Gerard, J. Champea, and J.-P. Babau, editors, *Model Driven Development for Distributed Real-time and Embedded Systems*. Hermes, 2005.

[40] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[41] J. Snell, D. Tidwell, and P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.

[42] J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, and G. Karsai. Domain Model Translation Using Graph Transformations. In *ECBS 2003*, pages 159–167, 2003.

[43] J. Sproston. Model checking for probabilistic timed systems. In B. C, H. B, H. Hermanns, J.-P. Katoen, M. Siegle, and F. Vaandrager, editors, *Validation of Stochastic Systems: A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science (Tutorial Volume)*, pages 189–229. Springer, 2004.

[44] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An Aspect-based Composition Tool for Real-time Systems. In *Proceedings of the IEEE Real-time Applications Symposium*, 2003.

[45] Systems Modeling Language (SysML). `http://www.sysml.org/`.

[46] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[47] UML Profile for Schedulability, Performance, and Time. ptc/2003-03-02 OMG Specification.

[48] UML 2.0. `http://www.omg.org/uml`.

[49] S. Yovine. Model checking timed automata. In *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems*, pages 114–152, London, UK, 1998. Springer-Verlag.