Using Model-Driven Engineering to generate QoS Monitors from a formal specification

Sébastien Saudrais, Olivier Barais, Laurence Duchien

Using Model-Driven Engineering to generate QoS Monitors from a formal specification

Sébastien Saudrais
IRISA France, Triskell Project[1]
ssaudrai@irisa.fr

Olivier Barais
IRISA France, Triskell Project
barais@irisa.fr

Laurence Duchien
INRIA France, Jacquard Project
duchien@lifl.fr

Abstract

In the domain of soft real-time application design, the gap between component-specification models and the implementations often implies that the implementations cannot fully take advantage of the specification models. To limit this gap, this paper proposes an approach to generate a QoS monitor from the timed behavior specification. To support this approach, we rely on two different component models: one focused on formal description and the other on practical implementation. Those models are interconnected by model transformation, using a Model-Driven Engineering *style.*

1 Introduction

Recently, hopes that modeling could take an important role in the software engineering process have been refuelled by so-called MDE (Model-Driven Engineering) initiatives, most prominently advanced by IBM with EMF, the OMG (Object Management Group) with the MDA or by Microsoft with Software Factories. The underlying idea is to promote models to be the primary artifacts of software development, making executable code a pure derivative. According to this development paradigm, software is generated with the aid of suitable transformations from a compact description (the model) that is more easily read and maintained by humans than any other form of software specification in use today.

In the soft-real time domain, the industry is interested in abstract component models to build systems. Such models improve the reusability of software modules because they provide three main features [7] for designing soft real time applications: (1) a composition model that provides operators able to compose existent libraries of components, (2) an abstraction level for defining components and connectors with only precise and yet abstract properties of the components, (3) a set of analysis tools to validate architectural descriptions. To enable an architectural analysis, the specification activity must add a time information within the component interface specification. Nevertheless, even though the real-time system community and the software engineering community use the component paradigm, the details are not necessarily the same. Consequently, although standards such as AUTOSAR [3] and sysML [**?**], for real-time systems, or UML 2.0 [13], for software engineering, promote the concept of component, there is not currently any component model designed to specify a real-time application by assembling components with a clear semantic and a clear mapping with a real-time framework such as Giotto [8] or Simulink [6].

Our work is motivated by the need to provide a bridge between the two communities to take the best of the different approaches: indeed software engineering provides standards and tools for the design of system and real time system engineering community provides semantic and tools for analysis. Consequently, we aim at preserving the correctness verification techniques of real-time components, while supporting component-based software architecture. Our approach aims at applying formal composition of specifications while supporting conventional source-code-based implementations. In this way, our paper proposes a Model-Driven Engineering process to generate a QoS monitor of the component system from timed-behavior specifications as illustrated in Figure 1.

The rest of this paper is organized as follows. Section 2 provides details on the languages and metamodels used in our approach. Section 3 details the component model and the real-time framework used for the implementation layer and explains the transformation process. Finally, Section 4 describes some related work and Section 5 concludes and discusses some future work.

[1]This work was funded by ARTIST2, the Network of Excellence on Embedded Systems Design

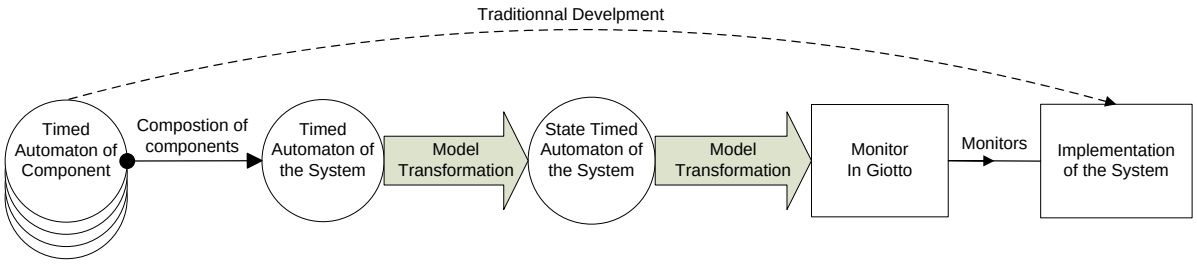

Figure 1. Overview of the approach

2 Analysis and design model

Several works in different domains converge on the use of components, ports, and connectors to describe a software architecture [11]. Our approach selects a suitable subset of UML 2.0 with a special emphasis on component-based architecture design with time-related features.

Furthermore, in our approach, a specification of a system consists in defining its architecture. This architecture is an abstract system specification consisting primarily of components described in terms of their behaviors, their temporal specification, their interfaces and the component assembly. This section presents the structural concepts used to define the architecture and the formalisms used to define the behavioral and the temporal properties of components.

2.1 Structural elements of the component model

The structural part of our component model is heavily inspired from the UML 2.0 architecture diagram. Nevertheless, contrary to UML 2.0, we define an abstract model with fewer concepts to limit the complexity of the language that the architect has to manipulate and to remove all the semantic variation points existing in UML 2.0.

Consequently, in our component model, a *component* provides *services* and may require some services from other components. Services can only be accessed through explicitly declared ports. A *port* is a binding point on a component that defines two sets of *interfaces*: *provided* and *required*.

Our component model distinguishes two kinds of components: primitives which will contain the code, and composites which are only used as a mechanism to deal with a group of components as a whole, while potentially hiding some of the features of the subcomponents. A primitive component can be seen as a basic building block in the component assembly. Our component model does not impose any limit on the levels of composition. The model thus provides two mechanisms to define the architecture of an application: *connector* between ports of components, and encapsulation of a group of components into a composite. A connector associates a component's port with a port located on another component. Two ports can be bound with each other only if the interfaces required by one port are provided by the other and vice versa. The services provided and required by the child components of a composite component are accessible through *delegated ports*, which are the only entry points of a composite component. A delegated port of a composite component is connected to only one child component port.

2.2 The behavioral part

With the interface and method definitions, a component declares structural elements about provided and required services. the behavioral part of the component model adds information about the behavior of a component. The behavior specification defines the component's interactions with its environment. This behavior is declared by a timed automaton [2] describing the sequences of messages that may be exchanged between the component and its environment with timed properties.

A timed automaton is an automaton extended with clocks, which are a set of variables increasing uniformly with time. Formally a timed automaton is defined as followed :

Definition 1. *(Timed Automaton)*
A timed automaton is a tuple $A =< S, X, L, T, \iota >$ *where :*

- S *is a finite set of locations,*
- X *is a finite set of clocks. To each clock, we assign a valuation* $v \in V$, $v(x) \in R^+$ *for each* $x \in X$.
- L *is a finite state of labels,*
- T *is a finite state of edges. Each edge* t *is a tuple* $< s, l, \psi, , s' >$ *where* $s, s' \in S$, $l \in L$, $\psi \in \Psi_X$ *is the enabling condition.* Ψ_X *is the set or predicates on* X *defined as* $x \sim c$ *or* $x - y \sim c$ *where* $x, y \in X$ *and* $\sim \in \{<, \leq, =\}$ *and* c *an integer.*

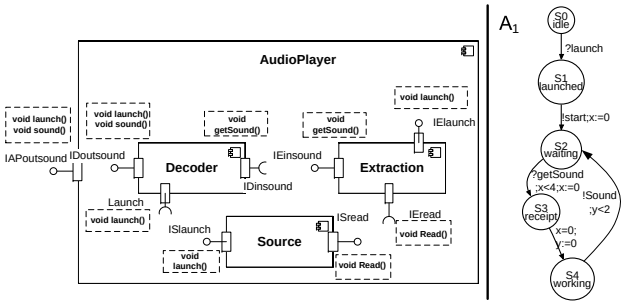

Figure 2. Example of an audio player component

- *ι is the invariant of A. $\iota \in \Phi_X$ where Φ_X is the set of functions $\phi : S \rightarrow \Psi_X$ mapping each location s to a predicate ψ.*

A state of an automaton is a location and a valuation of clocks who satisfies the invariant of the location. We can change of state by two types of transition : discrete transition and timed transition.

The timed automaton of composite is the composition of the timed automata of the components of the assembly. This timed automaton is the expected behaviour of the assembly with respect of timed QoS. The timed properties in the timed automaton refer to QoS properties. For example, at the implementation level, if the QoS wants to have a response in a specified time, the behaviour is correct if the response arrives in time. If the response is too late, the component does not stop but the QoS is not good and the user must be inform of this violation. We will transform automatically the timed behaviour to a monitor which can check the correct execution of the components.

2.3 Example

Fig. 2 illustrates the model with an example of component `AudioPlayer`. The `AudioPlayer` component provides an `IAPoutsound` interface that contains methods `launch` and `sound`. It is composed of 3 components: `Decoder`, `Extraction` and `Source`. The left side shows the structural representation of the component in UML 2.0. The right side of Fig. 2 shows an timed automaton A_1 describing all possible behaviors of the `Decoder`[1]. In this automaton A_1, we have two clocks: x and y. The first one is used for representing the response time of `?getSound` who has to be received less than each 4 units of time. The second clock is used for modelling the execution time of the transformation of `?getSound` into `!sound` which takes less than 2 units of time.

[1] In Fig. 2, in order to simplify the automaton, we only represent the receipt of message for a method call and the send of message for a method receipt.

3 A model oriented approach for code generation

From the component-based software architecture representation, our approach generates a QoS monitor based on the Giotto framework [8]. This section presents the Giotto framework. We also discuss the choice of a model transformation approach to generate the code from the specification to the implementation. Finally, we provide details on the transformation of an architecture specification with time constraints to the Giotto Framework.

3.1 The Giotto abstractions

Giotto is a real-time framework for embedded control systems running on possibly distributed platforms. A Giotto program explicitly specifies the exact real-time interaction of software components with the physical world. The Giotto compiler automatically generates timing code that ensures the specified behavior on a given platform. The Giotto model is based on four main concepts:

- ports,
- tasks,
- drivers,
- and modes.

In Giotto, all communication are performed through *ports*. Giotto defines five kinds of ports. Two kinds of port (*Sensor - Actuator*) manage the input and the output interactions with the hardware layer. Two others kinds of port (*Input - Output*) manage the interactions with the software layer. They are used to exchange data between concurrent tasks. Finally, the *private* ports represent the state of a task. They are inaccessible outside the task in which they are defined.

In Giotto, a *task* has a set of inputs and outputs ports, a set of private ports and a function which infers the outputs from the input ports. This function is implemented by a sequential program and is written with a common programming language. For each function, the Giotto framework has to know the worst-case execution time of the function on each available CPU.

The third type of elements in Giotto is the *driver*. A *driver* is a function that converts the value of sensor ports or outputs ports of the current *mode* to values for the input ports. Driver are guarded: this gard is a predicate on a sensors and output ports of a mode.

The main concept of Giotto is the *mode*. A *mode* consists of a period, a set of output ports for the mode and a set of `freq`. A freq defines the frequency of an action during the period. This action can be an actuator update

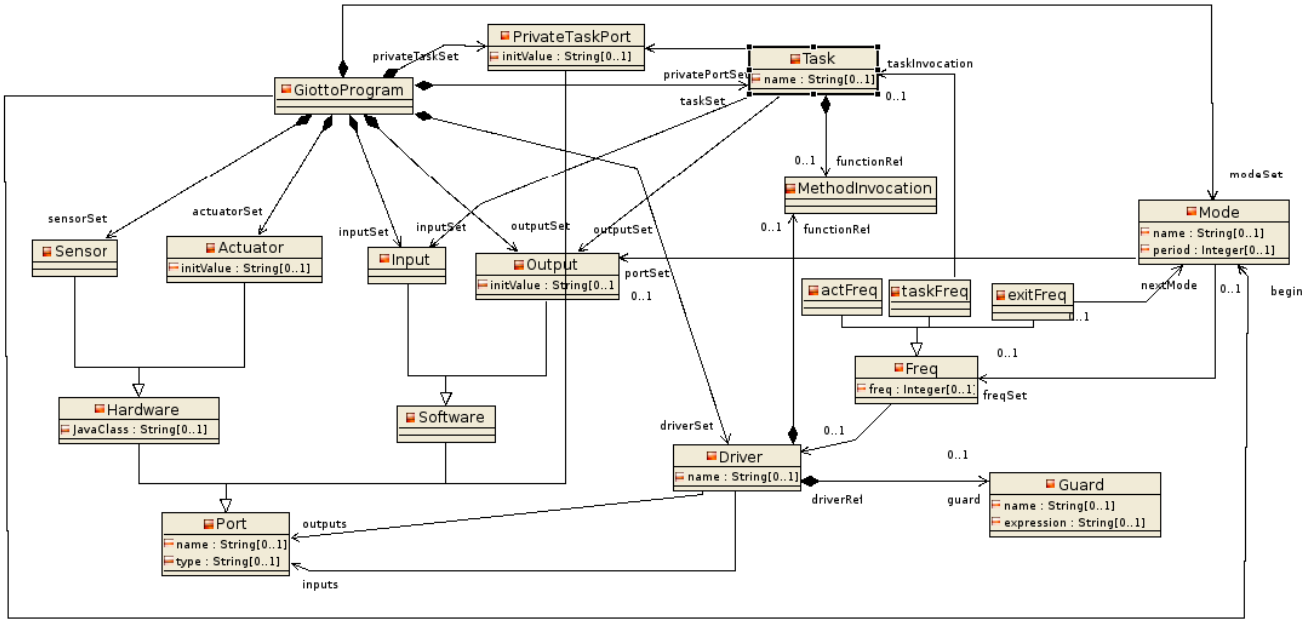

Figure 3. Giotto meta-model

(`actFreq`), a task invocation ((`taskFreq`) or a switch to another mode (`ExitFreq`). A mode switch defines a transition from one mode to another mode. For this purpose, a mode switch specifies a target mode and a driver. The guard of the driver is called the *exit* condition, as it determines whether the switch occurs. The Giotto meta-model is presented in Fig. 3.

3.2 From the specification to the implementation

From the specification of a component, we generate the skeleton of the business component and the configuration descriptors. From the timed automata, we generate the Giotto layer implementation that controls the respect of the time constraint in the architecture of the architecture. Indeed, Giotto separates the system's behaviour from its implementation. Then we have three levels in the implementation architecture: functional, time interaction and platform. In the functional level, we find business components generated from the specification of services and the abstract implementation. In the time interaction level, we find the Giotto layer generated from the timed automata and the time constraints. Finally, in the platform level, we find the specification of the platforms as the topology of CPUs and networks and the performance. Choosing a MDE approach has two main benefits for the QoS. The time interaction is decoupled from the functionalities. The framework improves the separation of concerns. Moreover, the generative approach improves the productivity of the development process. To define a MDE approach, we use Kermeta a model oriented language. It allows the design of the different meta-model of the generative process and the implementation of the transformation itself.

Kermeta: a model oriented language Kermeta[2] is an open source meta-modeling language developed by the Triskell team at IRISA. It has been designed as an extension to the EMOF [14]. Kermeta extends EMOF with an action language that allows specifying semantics and behavior of meta-models. The action language is imperative and object-oriented. It is used to provide an implementation of operations defined in meta-models. A more detailed description of the language is presented in [12]. The Kermeta action language has been specifically designed to process models. It includes both Object-Oriented (OO) features and model-specific features. Kermeta includes traditional OO static typing, multiple inheritance and behavior redefinition/selection with a late binding semantics.

To implement the transformation process between our component model and the Giotto, we have chosen Kermeta for four reasons. First, it gives a graphical and textual representation for designing the different meta-models identified in the process. Second, the language allows implementing a composition semantic in the component model by adding the algorithm in the body of the operations defined in the component metamodel. Third, Kermeta is suitable for model processing. It also includes specific concepts

[2]http://www.kermeta.org

such as opposite properties (i.e. associations) and handling of object containment. In addition to this, convenient constructions of the Object Constraint Language (OCL), such as closures (e.g. each, collect, select), are also available in Kermeta. Finally, Kermeta tools are compatible with the Eclipse Modeling Framework (EMF) which allows us to use Eclipse tools to edit, store, and visualize models. This second argument is more technical than scientific, but it is very interesting to tool quickly the different meta-model defined in the approach.

Generating the Giotto layer The assembly of components at the specification level gives a timed automaton describing the behaviour of the complete system. We will transform this automaton to Giotto to monitor the implementation of the components. If a component does not have a correct behaviour, Giotto can inform the user that the level of QoS is no longer correct. The real components are developed by traditionnal methods and must only inform Giotto of the arrival of messages.

The first step of the transformation is to transform the timed automaton. From the automaton $A1$, we will create the automaton $A2$ as illustrated in Fig. 4. The second automaton represents the states of the first automaton with discrete and time transitions. It can be viewed as a simulation automaton because each state represents the system at a given time. For the example, locations $s0$ and $s1$ have only discrete transitions. The two clocks are reinitialized before being used so no timed transitions are used before their initialization. Each timed transition increases the time unit by 1 so for the state $wait$, which must hold no more than four units of time, it is transformed to four states.

The second step of the transformation is to produce the Giotto code. This step is made with the help of MDE. A model transformation helps us to create the Giotto model. A pretty printer was created for the Giotto meta-model. This generates the textual representation used as input to the Giotto compiler as illustrated in Fig. 6. The meta-model of timed automata with states is represented in figure Fig. 5. The main idea of the transformation is to create one mode for each states of the timed automata and mode switches for transitions. The code produced for the example is:

The time unit used for our timed automata is second wheras for Giotto it is millisecond. For example, the state $S2\ Waiting\ x\ =\ 1$ has 2 transitions: one discrete $?getSound$ and 1 time transition so the corresponding mode $waitingone()$ has 2 mode switches. The discrete transition is transformed to a mode switch $exitfreq\ 1\ do\ workingone(CGET)$ where $CGET$ verifies if the message $?getSound$ has arrived. The timed transition is transformed to a mode switch $exitfreq\ 1\ do\ waitingtwo(True)$ which means if nothing happen during the period the automaton changes of state with a time transition. The line $taskfreq\ 1\ do\ Idle(getMessages)$ updates the arrival of messages.

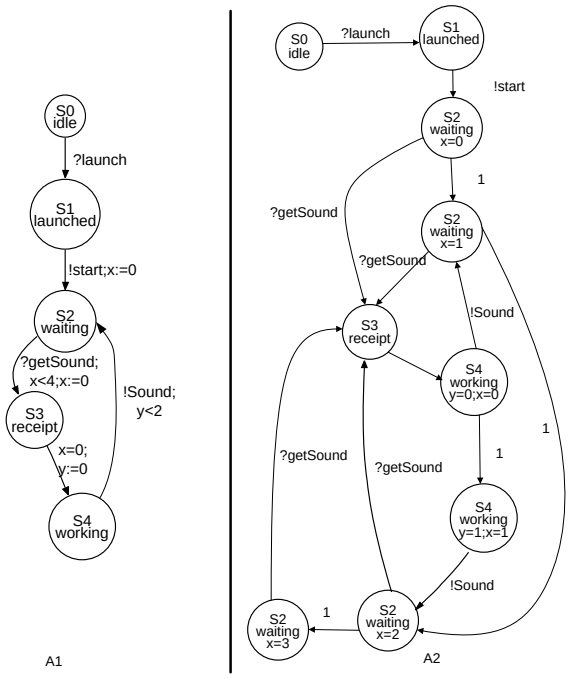

Figure 4. Transformation of automata

The addressed domain is QoS so the program will not stop if a message is not received. For the example, we introduce a single mode error. In reality, different modes will be introduced depending of the policy of QoS: allowing five kinds of error and enabling the reconfiguration of the assembly for example.

3.3 Concrete implementation consistency

Our approach aims at removing the gap between the techniques used by the developers to implement the applications and the model used by the designer/the architect to specify and analyze their system. The use of model transformation techniques ensures that the concrete implementation has the same time constraints than the specification and the abstract implementation. At the concrete implementation level, the respect of these constraints is checked by the addition of a real time controller on the component to interact with the QoS monitor. Besides, the use of Giotto as a concrete implementation target allows the architect to check if the specification of the platform is constrained enough to obey the time constraints.

The main interest of our approach consists in generating the concrete implementation time consistency checking from the specification. The Giotto real time framework guarantee the time correctness. Consequently, the implementation of the adaptation policy in the case of QoS con-

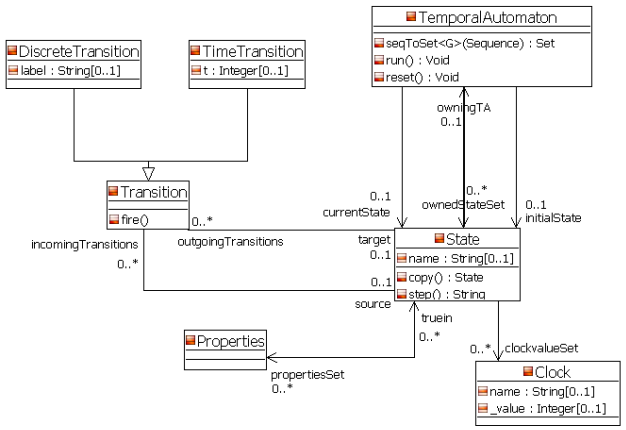

Figure 5. Meta-model of timed automata with state

tract violation does not tangle the functional components. For the moment, the main limitation of the approach is the risk of state explosion of the timed automata increased by the discretization of the different clocks in the transformation process. This risk is limited with the calculation of the highest discretization step for each clock.

4 Related work

Several research results have shown the usefulness of specific languages to describe the software architecture. Thanks to the precise semantics of such languages, tools suites have been developed to analyze the consistency of a software architecture and to prototype it. For example, SOFA [9] provides a specific language that extends the OMG IDL to specify the architecture of component based software. It also provides a process algebra to specify the external behavior of component. However, using SOFA the architect cannot describe the required and provided QoS of components. The AADL standard [15] is one of the first ADL that provides mechanism to specify the QoS level of component interface [4]. However, AADL is a low abstraction model, strongly connected with the implementation. Besides, AADL is not yet connected with tools that use the QoS information to analyze the consistency of the architecture.

At the validation level, the OMEGA project [1] provides formal methods to check the consistency of UML 2.0 models. The OMEGA approach deals with the specification level only. It does not provide any global development process that includes source code development. Uppaal [10] is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. Their results are only on the model level and not linked to implementation. Consequently, the

```
start idle{
  mode idle() period 1000 {
    actfreq 1 do motion(Move);
    exitfreq 1 do launched(CLAU);
    taskfreq 1 do Idle(getMessages);
  }
  mode launched() period 1000 {
    actfreq 1 do motion(Move);
    exitfreq 1 do waitingone(CSTA);
    taskfreq 1 do Idle(getMessages);
  }
  mode waitingone() period 1000 {
    actfreq 1 do motion(Move);
    exitfreq 1 do workingone(CGET);
    exitfreq 1 do waitingtwo(True);
    taskfreq 1 do Wait(getMessages);
  }
  mode waitingtwo() period 1000 {
    actfreq 1 do motion(Move);
    exitfreq 1 do workingone(CGET);
    exitfreq 1 do waitingthree(True);
    taskfreq 1 do Wait(getMessages);
  }
  mode waitingthree() period 1000 {
    actfreq 1 do motion(Move);
    exitfreq 1 do workingone(CGET);
    exitfreq 1 do waitingfour(True);
    taskfreq 1 do Wait(getMessages);
  }
  mode waitingfour() period 1000 {
    actfreq 1 do motion(Move);
    exitfreq 1 do workingone(CGET);
    exitfreq 1 do error(True);
    taskfreq 1 do Wait(getMessages);
  }
  mode workingone() period 1000{
    actfreq 1 do motion(Move);
    exitfreq 1 do waitingtwo(CSOU);
    exitfreq 1 do workingwo(True);
    taskfreq 1 do Working(getMessages);
  }
  mode workingtwo() period 1000{
    actfreq 1 do motion(Move);
    exitfreq 1 do waitingthree(CSOU);
    exitfreq 1 do error(True);
    taskfreq 1 do Working(getMessages);
  }
  mode error() period 1 {
    actfreq 1 do motion(Move);
    taskfreq 1 do Error(getMessages);
  }
}
```

Figure 6. generated code

OMEGA project is complementary to our approach.

At the implementation level, Qinna [17] is a component-based QoS architecture for open system. They integrate QoS on their architecture but they don't integrate QoS specification in their model. Chan et al. proposed a model-oriented framework for monitoring at runtime extra-functional properties[5]. They address probabilistic temporal properties. Their monitoring is made at runtime by checking constraints written in PCTL. They also make a .NET-based implementation of their framework. The SeCSE[16] project aim to create methods, tools and techniques for systems integrators and service providers. It will integrate tools and techniques to provide a SeCSE development environment. Their approach is service-based and they take care of QoS but they target only web-services.

5 Conclusion and perspectives

Correctly designing and implementing a real-time system is usually an error-prone task because of the gap between the specification model and the implementation model. This paper is a step toward bridging this gap. It proposes a unified approach to design and to implement component based systems. This approach aims at assisting architects in the design and in the implementation of soft-real-time systems by providing a set of tools that generate the QoS monitors from the specification of those systems using a Model Driven Engineering style. This approach is based on an extended UML 2.0 standard to design the services provided by component, to specify the component and to give a first abstract implementation of the systems. It clearly separates the functional level, the timing interaction level at the implementation level.

We are currently working on a proof of correctness for the transformation process. This proof must ensure that the composition mechanism, at the concrete implementation level, is valid with respect to the composition mechanism at the abstract level. This is needed to preserve the results gained by validation at the abstract implementation phase.

Finally, we intend to test our approach in the context of the HRC componeent model provided in the SPEEDS project [18].

References

[1] Webpage of the OMEGA IST project. http://www-omega.imag.fr/.

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

[3] AUTOSAR partners. AUTomotive Open System ARchitecture, August 2005. Version 1.5 light version.

[4] A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[5] K. Chan, I. Poernomo, H. W. Schmidt, and J. Jayaputera. A model-oriented framework for runtime monitoring of nonfunctional properties. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker, and Patrick J. Schroeder, editors, *QoSA/SOQUA*, volume 3712 of *Lecture Notes in Computer Science*, pages 38–52. Springer, 2005.

[6] J. B. Dabney and T. L. Harman. *Mastering SIMULINK*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[7] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[8] T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, January 2003.

[9] T. Kalibera and P. Tuma. Distributed component system based on architecture description: The sofa experience. In *On the Move to Meaningful Internet Systems - DOA, CoopIS and ODBASE*, pages 981–994, London, UK, October 2002. Springer-Verlag. *ISBN: 3-540-00106-9*.

[10] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[11] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, volume 26, page 23, January 2000.

[12] P-A. Muller, F. Fleurey, and J-M. Jézéquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

[13] Object Management Group OMG. *Unified Modeling Language: Superstructure*, August 2003. Version 2.0.

[14] Object Management Group OMG. Meta-Object Facility (MOF) Specification, 2005. Version 2.0.

[15] As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506, November 2004.

[16] Walkerdine J. Sommerville I. Sawyer P., Hutchison J. Faceted service specification. In *Proceedings of Workshop on Service-Oriented Computing Requirements (SOCCER)*, August 2005.

[17] J.C. Tournier, J.P. Babau, and V. Olive. Qinna, a component-based QoS architecture. In G.T. Heineman, I.Crnkovic, H.W. Schmidt, J.A. Stafford, C.A. Szyperski, and K.C. Wallnau, editors, *CBSE*, volume 3489 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2005.

[18] A. Metzner B. Josko T. Peikenkamp E. Bde W. Damm, A. Votintseva. Boosting re-use of embedded automotive applications through rich components. In *FIT'05 Foundations of Interface Technologies*. Elsevier Science, August 2005.