A Framework for Formal Component-Based Software Architecting

M.R.V. Chaudron, E.M. Eskenazi, A.V. Fioukov, D.K. Hammer

Department of Mathematics and Computing Science, Technische Universiteit Eindhoven,

Postbox 513, 5600 MB Eindhoven, The Netherlands

+31 (0)40 – 247 4416

{m.r.v.chaudron, e.m.eskenazi, a.v.fioukov, d.k.hammer}@tue.nl

ABSTRACT

The assessment of quality attributes of software-intensive systems is a challenging issue. This paper outlines a method aimed at quantitative evaluation of certain quality attributes that are important for embedded systems: timeliness, performance and memory consumption.

The paper sketches out the key principles for building a formal model for evaluating quality attributes: (1) Dependability constraints are specified in an end-to-end fashion; (2) Components are attributed with resource demands; (3) Specification of component interaction is separated from specification of component behavior.

The method is aimed to be applicable in practice. Therefore we investigate combining widely used software modeling notations with existing formal methods. In particular, the proposed approach combines Message Sequence Charts and Timed Automata. We illustrate the approach with an example.

Categories and Subject Descriptors

D.2.11 Software Architectures; D.2.4 Software / Program Verification

General Terms

Performance, Design, Reliability, Verification.

Keywords

Component-based software, software architecture, quality attributes, architecture evaluation, timeliness, memory consumption, formal methods.

1. INTRODUCTION

Nowadays, more and more "intelligent" devices contain sophisticated embedded software to fulfill a broad scope of functions. As more devices are developed, the scope of functions to be implemented must be broader. This growing complexity complicates the development of embedded software. Thus, new approaches for software development are heavily needed.

To reduce development cost and development time, the reuse of existing solutions is vital. For this purpose, the construction of software with reusable components is highly desirable. This method of software development requires techniques for assessing the composability of components. This assessment can be done reasonably well for the functional aspects of components, but no adequate techniques exists for analyzing composability of non-functional ones.

The possibility to estimate relevant quality attributes in the early development phase is crucial. These kinds of predictions will reduce the risk of developing non-competitive, infeasible or flawed products. Thus, for quantitative software architecting a formal mathematical basis is needed.

Most contemporary architecting approaches deal only with the functional aspects of software. But, there are also non-functional quality attributes, a typical example being dependability quality attributes: performance, timeliness, reliability, availability, safety, and security. These attributes emerge as a result of collaborative functioning of all parts, which a system is built from, and they can make a significant impact on the entire architecture. Consequently, one of the most important issues in software architecting is dealing with quality attributes.

Many modern techniques for software evaluation use expert-based approaches (e.g. ATAM [4], SAAM [1]) which evaluate overall quality of the architecture. The quantitative methods used in these approaches are applied in an ad-hoc manner and, in many cases, are too context-oriented to be generalized. As a consequence, it is difficult to make the architecting process *reliable*, *predictable*, *and repeatable*.

For the time being, architecting is still more an art than an engineering discipline. Substantial efforts have to be invested in making the architecting process more *rationalized and precise*. One of the ways to make the architecting process more precise is the use of formal methods. However, formal methods are not a silver bullet; they have their own drawbacks. One should be aware that the application of formal methods usually requires precise and complete specification. In many cases, an architect does not have this information. He or she may not even be interested in a very detailed design at the early stages of design, but prefer to postpone design decisions to later stages. Thus, a balance between the architect's freedom of design and the precision of analysis and specification should be found. Note, accuracy of a specification is not always a drawback, as it also stimulates one to think more accurately and systematically.

Support by automatic tools could be very helpful for architects. However, tooling usually requires strict non-ambiguous semantics of the models being processed; thus, formal methods are also essential here.

There exist no general approaches for evaluating non-functional properties of a system at the architectural level. The interesting approach combining structural description of architecture (Darwin Architecture Description Language) with behavioral description of components (Labeled Transition System) was proposed in [8]. However, this approach only allows checking safety and liveness properties, but it does not model timing aspects which are needed for the analysis of quality attributes like timeliness and performance. Also, the existing methods for quantitative evaluation focus on one quality attribute only.

We aim to integrally model multiple quality attributes–*timeliness*, *performance*, and *memory consumption*–to support the making of architectural design trade-offs.

1.1 Requirements on the Method

The aim is to develop a method for supporting software architects during early stages of component-based software architecting of resource-constrained systems. It is necessary to find suitable techniques for architecture description and methods for quality attributes evaluation and to merge them into an integrated framework.

The requirements on the method are the following:

1. Compositionality of resource-constraint systems. The methodology should be *compositional*. This means that quality attributes of a composite system can be calculated from the constituents components and composition mechanisms. In particular, the method should focus on predicting quality attributes based on resource consumption of the components.

Even if the functional interfaces and the interaction mechanisms of components are precisely specified, component composition may not function properly because non-functional properties of the entire system were not considered beforehand. Typical instances of this problem are resource conflicts (e.g., race conditions, conflicts on access to shared resources etc). These conflicts make it difficult to ensure the proper and predictable behavior of a system in advance.

2. Dependability assessment during the early development phases. The dependability attributes (timeliness, performance, reliability, availability, safety and security) of component-based systems cannot be evaluated by the current approaches. So, the method must help architects to estimate the dependability attributes and ensure a certain level of *estimation accuracy*. As a simple example, the results can be described in terms of worst-case and best-case estimations.

Also, there are non-technical requirements on the method. In order to be easily comprehensible and easy to learn by software engineers, the method should be based on widely accepted software specification and design techniques. These techniques should make engineer's work more efficient after the engineer has gained some experience with them.

2. KEY PRINCIPLES FOR COMPONENT-BASED ARCHITECTING

Component-based architecting (e.g., see [15] and [21]) is one of the most promising approaches for managing complexity and boosting reuse. However, current component-based approaches do not address the behavioral and non-functional aspects of software. Therefore, we propose the following extensions.

Explicit specification of component behavior and interaction. A formal specification of the dynamic aspects of components and their interaction is necessary for reasoning about the behavior of their composition and its non-functional properties.

Support of hierarchical component description. A component can be either atomic or compound. The atomic component cannot be further subdivided, but the compound component can consist of atomic and(or) other compound components. As a result, one has more flexibility in choosing the unit of the reuse: either a single atomic component or an entire package. Another advantage is the possibility to apply compositional design approach at the different levels of hierarchy: a system is composed from subsystems; subsystems are composed from compound components etc.

Separation of component interaction from component behavior. The description of behavioral aspects is structured in separate parts. There are specifications of *component behavior* and specifications of *component interaction*. A rationale for independent specification of the interaction relationships is presented in [2]. We identified the following additional reasons for this separation:

1. Genericity/Tailorability: The interaction specification may be used to tailor the behavior of generic components to particular context. This helps to avoid coding of context-driven aspects within components and, hence, allows more general component designs.
2. Specifying constraints end-to-end: Dependability constraints are often concerned with the end-to-end interaction between components. Having a separate specification of the interaction constitutes a better means for structuring the specification than the alternatives: (1) placing constraints at one of the components involved or (2) dividing up an end-to-end timing constraint over multiple components.
3. Loose coupling: In existing component models the way that a component is intended to interact with other components is programmed into a component (endogenous binding). This has to change if the behavior of other components changes. Hence, it constitutes a dependency on the behavior of other components. By specifying interaction separately (exogenously), this dependency is avoided.

Explicit specification of the resource requirements of components. The dependability of a system is related to the amount of resources consumed by the components and provided by the execution platform. There are three types of resources: computation resources, communication resources and storage. The definition of component resource requirements in a platform-independent way broadens the scope of component application.

Specification of dependability constraints in an end-to-end fashion. The basic idea is that timing and dependability

constraints should not be component attributes, because this would jeopardize reusability. They are rather considered as constraints on the dynamics of the system, i.e. on the component interaction.

Distinguish resource constraints and resource consumption. The former are described at the overall system level in an end-to-end fashion, and the latter is associated with a component description. This separation enables the development of reusable components and gives designers freedom in the satisfaction of the resource constraints.

All the aforementioned aspects have to be properly elaborated in order to constitute a practical method.

3. ARCHITECTING ENVIRONMENT

To reconcile the goals of using accepted software engineering notations and automated analysis tools, we aim for a framework that consists of three parts (see Figure 1).

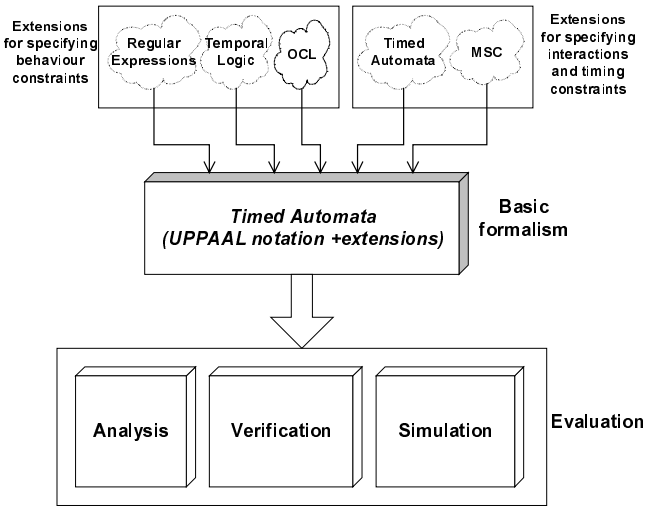

Figure 1. Architecting environment

The architecting environment provides engineers the possibility to use a combination of notations for describing architectures. For the time being, we focus on Message Sequence Charts (MSC) [20] and Timed Automata (TA) [3]. Also, the architecting environment is to be extendible with other notations, such as the Object Constraint Language (OCL) [24] or temporal logic.

For analyzing architectural designs, the different notations need to be related. To this end, we devise mappings of notations onto a *basic formalism*. In our approach, Timed Automata are used as basic formalism. We do not expect to find mappings of all constructs of all modeling notations onto a single basic modeling formalism. In co-operation with engineers, we have to select subsets of the notations that comprise the most important modeling constructs, yet also provide the information needed for automatic analysis.

To support different types of analyses we envisage a *collection of analysis tools* such as schedulability-, simulation- or verification-tools. These tools operate on the representation of the architectural design in terms of the basic formalism.

4. ARCHITECTURE DESCRIPTION TECHNIQUES

This section enumerates the requirements on architecture description techniques and outlines the framework. Also, it contains an example to illustrate the method proposed.

4.1 Requirements for description techniques

For specifying component behavior, a number of formal description techniques were inspected and compared. Before the actual comparison, essential requirements on the description techniques were identified. These requirements and their rationale follow below. The requirements are marked as *compulsory (C)* or *optional (O)*.

1. The description techniques should support quantitative models for timeliness analysis. *(C)*

   Rationale: to enable timeliness assessment at the early architecting phase, before system implementation.

2. The description techniques should support quantitative evaluation of memory consumption. *(C)*

   Rationale: to enable memory consumption estimation at the early architecting phase.

3. The description techniques should support specification of timing constraints in an end-to-end fashion. *(C)*

   Rationale: to avoid unnecessary reduction of design space (caused by artificial subdivision of the initial deadlines).

4. The description techniques should support the possibility to specify interaction behavior exogenously *(C)*.

   Rationale: to increase reusability of the components and to build flexible architectures.

5. The description techniques should allow one to reason about the properties of a component composition, based on the properties of the components. *(C)*

   Rationale: To enable effective (automated) formal reasoning.

6. The description techniques should support the specification of resource requirements (processing, storage and communication. *(C)*

   Rationale: to enable analysis of effects of resource conflicts.

7. The specifications must be comprehensible for engineers *(C)*.

   Rationale: reduce efforts for education of engineers.

8. Support for automatic code generation *(O)*.

   A description technique should enable creating of tools that can generate code for a given specification.

   Rationale: to enable efficient development.

9. Use of existing design, simulation and verification tools *(O)*. It is preferable to use the existing tools instead of developing new ones.

   Rationale: to design the software quickly and easily.

4.2 Basic formal framework

In this section we explain the architecting approach by listing the models that should be constructed for describing an architecture. We motivate the choices of the formal description techniques for these models.

4.2.1 General view

A general overview of the approach to modeling architectures is given in Figure 2. Three essential architecting models are considered.

The *"Structural model"* represents the static configuration of a system through the dependencies and connections between components.

The *"Behavioral Model"* is used to describe the dynamic aspects of the components, component interaction and resource constraints (e.g. end-to-end deadlines). A component description specifies resource requirements in terms of the "Resource Model".

The *"Resource model"* describes the available resources. This model also defines a sharing strategy for each resource.

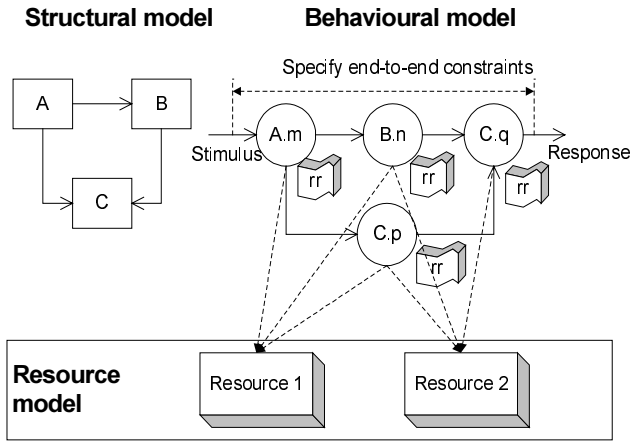

Figure 2. Overview of the approach

4.2.2 Choice of appropriate formalisms

For structural description we consider one of the existing component models supporting the notions of provided and required interfaces (e.g. Koala [19] and Darwin [18]). Since we aim to the evaluation of timing properties, the proper behavioral description formalisms are to be found.

After comparing several formalisms, the extended notion of State Machines (Timed Automata, [3]) was chosen for the specification of component behavior. Basic Message Sequence Charts [20] were chosen for specifying component interaction. This section motivates this choice.

4.2.2.1 Timed Automata

Timed Automata are supported by a wide scope of existing tools for modeling and simulation (e.g. UPPAAL, for details see [6]). Furthermore, their graphical description makes them comprehensible for engineers.

The theory on Timed Automata describes constructions for obtaining an automaton that describes the behavior of the parallel composition of timed automata. For the simplest cases, it is possible to use the Cartesian product.

However, when using timed automata, certain principles must be followed in order to be able to reason compositionally. For example, one should not use global clock variables to define constrains on the behavior of multiple components.

4.2.2.2 Message Sequence Charts

As mentioned before, component interaction is specified separately from component behavior. For that, a specification language is required that can address the following issues:

- It should enable restricting the behavior of generic components
- It should support the specification of timing constraints in an end-to-end fashion.

The MSC notation allows one to vividly express timing constraints between stimulus and response events at a single place in a specification. This is in contrast to timed automata, where timing constraints are specified by means of two (or more) constraints on shared clock variables that are distributed over separate states or transitions of the model (this will be illustrated later by an example). This reduces the intelligibility of a specification.

MSC are a well-accepted software notation that is easy to learn and understand. Also, they have formal semantics in terms of automata (see e.g. [14]) that makes it possible to relate them to timed automata (which we use as basic formalism).

Because of the above advantage, MSC were preferred to Timed Automata for specifying the timing constraints.

4.3 Example

To give a flavor of our approach, we will demonstrate some of the description principles (analysis is not included) with an example of an "Automatic Teller Machine".

The structural model of the architecture is depicted in Figure 3.

Figure 3. Key components

The system consists of the following components: User, ATM (modeling a cash dispenser), and Bank (modeling some aspects of bank operation). User and Bank interact only with ATM, but not with each other.

The behavioral model consists of a specification of the behavior of the individual components using UPPAAL and a specification of their interaction using MSCs.

We briefly explain the UPPAAL notation [6]. In UPPAAL, time is modeled using clock variables: timing constrains are expressions over clock variables. These constraints can be attached both to transitions and states. A condition on a state is an *invariant*; the system is allowed to be in a state only if its invariant holds. A condition on a transition is a *guard*; the transition can only be taken if the guard holds.

The labels on transitions denote events. Labels with a question mark "?" define input events; labels with an exclamation mark "!" define output events.

Returning to the example, Figure 4 describes the behavior of User.

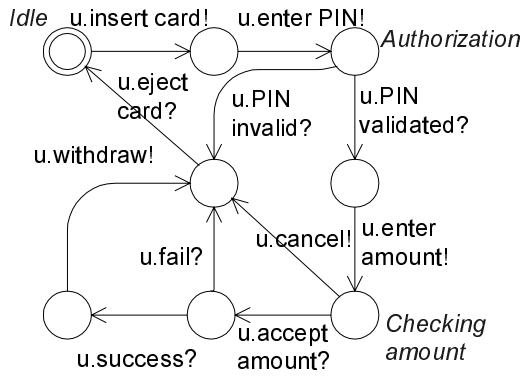

Figure 4. Behavior of User

The specification of User describes the behavior of an individual who wants to withdraw cash from an ATM. The automaton defines an order on the events for the withdrawal process.

The behavior of ATM and Bank is illustrated in Figure 5 and Figure 6, respectively.

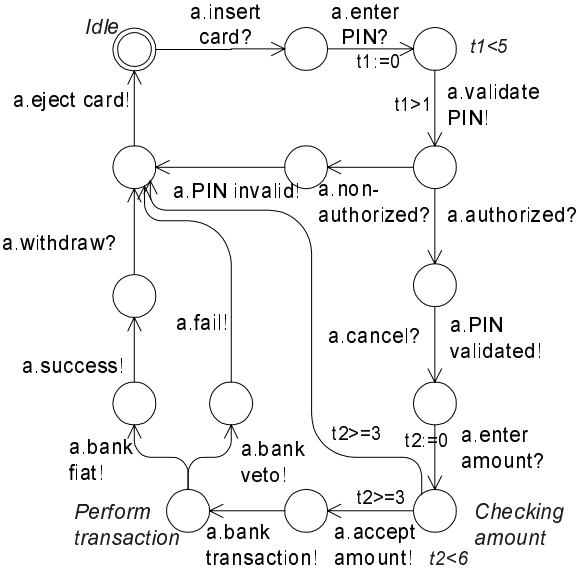

Figure 5. Behavior of ATM

For the specification of Bank we use two automata. The semantics of this is that they operate in parallel. This allows further enhancing of the specifications to support more than one User and one ATM: the unnecessary serialization of the authorization and transaction requests from different cash dispensers, which would be enforced by modeling the behavior as a single automata, is avoided.

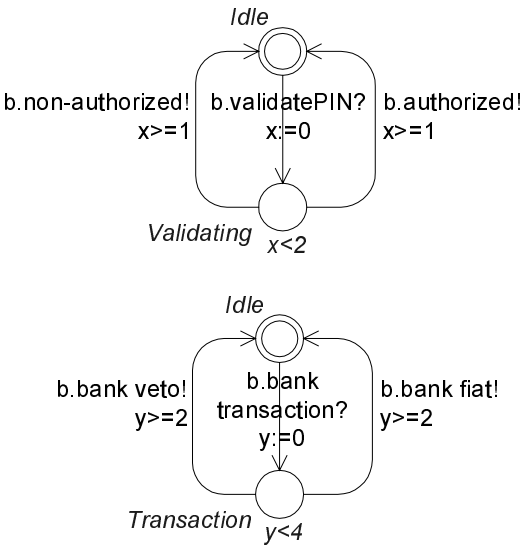

Figure 6. Behavior of Bank

We use clock variables x and y to specify the resource consumption of Bank. The specification states that the time of processing authorization requests (by a hypothetical CPU) is at least one time unit and at most two time units. The former is indicated with the guards ($x>=1$) on both transitions, and the latter is specified with the invariant ($x<2$) of the *"Validating"* state. Likewise, the time necessary for performing a transaction is at least two time units but at most four time units. Similarly, one can specify consumption of CPU capacity for other components.

Finally, we demonstrate the specification of end-to-end timing constraints and the interaction between the components User, ATM, and Bank with the Message Sequence Chart in Figure 7.

On the one hand, the message sequence chart in Figure 7 specifies through which transitions all the three automata interact. For example, to indicate that the actions "u.insert card" and "a.insert card" of User and ATM, respectively, need to synchronize, we use an ampersand symbol "&". Likewise, the other labels of all the three automata are bound. This type of specification technique allows one to bind components in an exogenous manner.

Additionally, this message sequence chart indicates that the time between the occurrence of "enter PIN" and the occurrence of "PIN validated" must not exceed five time units. At the same time, it indicates that time between "enter amount" and "transaction success" must not exceed six time units. In a similar way, message sequence charts can be used to specify other dependability constraints in an end-to-end manner for relevant execution scenarios.

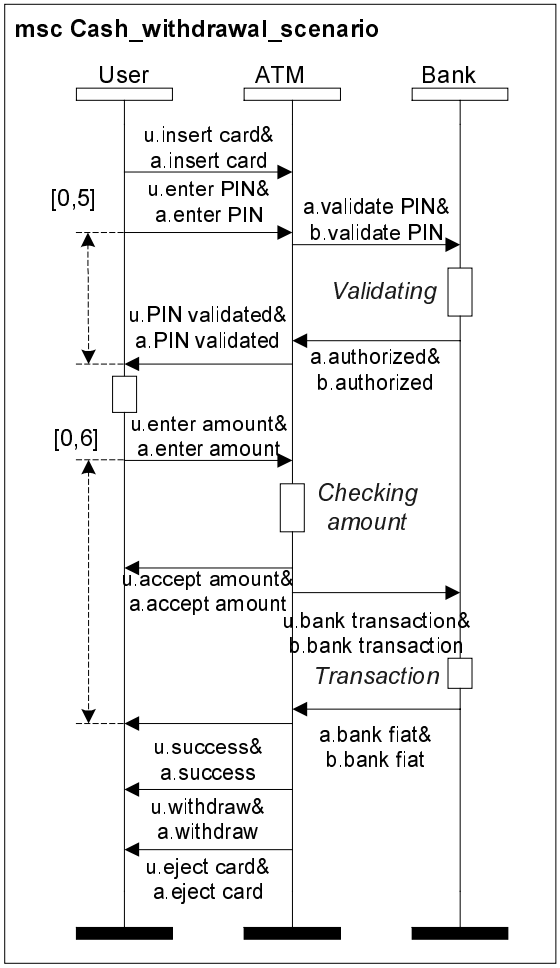

Figure 7. Specification of deadlines and component interaction

5. TECHNIQUES FOR EVALUATION OF QUALITY ATTRIBUTES

This section summarizes our evaluation of methods for the assessment of timeliness and memory consumption and interprets their use for the software architecting.

5.1 Timeliness evaluation

As already mentioned, timeliness is an important quality attribute. Timeliness can be reasoned about either (analytically) through a schedulability test, a typical example being Rate Monotonic Analysis (RMA) (see [17], [16], [11], and [12]), or through the construction of an explicit schedule (e.g. in [23]).

Both approaches model the scheduling policy adopted for the system. Depending on whether the priority assignment strategy of the scheduling policy is fixed or dynamic, different models have to be used.

For the fixed-priority scheduling policy, RMA is usually applied. This method is based on the analysis of a so-called critical instant, when all tasks in the task set are released simultaneously. It is proven that the worst-case response time appears for each task during the critical instant. The RMA method calculates response time for each task (for a given real-time situation describing a set of tasks being analyzed [12]), based on the specified worst-case execution time and deadline. In addition, if the tasks share some resources, the blocking time induced by the ones with lower priority should be given. Finally, the periodicity information (for a simple case, in a form of task inter-arrival periods) has to be provided to enable the application of RMA.

However, RMA can only be used with some, rather strict, assumptions on the tasks of the system. The main restriction is the assumption that the arrival pattern of a stimulus has some form of *periodicity*. Early versions of RMA have dealt only with strictly periodic events; however, later extensions have incorporated a-periodic tasks and sporadic servers. Another drawback of RMA is that it does not allow *interaction* between the tasks. The only allowed interaction is mutual access to shared resources.

New schedulability modeling approaches have emerged as a result of the substantial progress in the development of model checking techniques both for ordinary and hybrid timed automata. These approaches partially address the drawbacks of RMA-like techniques, as they take interaction between the tasks into account.

The main principle on which these techniques are built is the replacement of the initial schedulability problem with the reachability problem for a timed automaton encompassing all peculiarities of a concrete schedulability policy. The automaton modeling the scheduler, combined with automata modeling inter-task communication, is analyzed for reachability of the state corresponding to a non-schedulable situation [23].

In general, automata-based methods cover a broader scope of possible scheduling policies, than RMA-like methods do, as they also take into account task interactions. However, the automata-based methods have two disadvantages. The first is that these methods only indicate whether a task set is schedulable or not: they do not provide the response time of the tasks, which would be very useful for architects. The second drawback is that these methods, being based on the construction of an automaton modeling the scheduler, suffer from the state explosion problem. Fortunately, during the last three years, a number of successful accounts about the application of automata-based techniques have been published [5], [7], and [13]. Because of the flexibility of these kinds of techniques, their application is feasible at the architecting level, especially in the cases when standard techniques like RMA are not applicable.

Both types of techniques require information about the worst-case execution time of tasks. Unfortunately, most contemporary methods for the estimation of worst-case execution time cannot be directly applied to architecting, as they are based on already existing code. But at the architectural level, some estimations are often needed before the code is written. Furthermore, there are two problems with traditional approaches:

- Predictions are over-pessimistic due to excluding effects of the acceleration facilities of modern CPU's (pipelines, branch prediction blocks, caches etc). These effects are excluded from the analysis because of unpredictable behavior of the acceleration facilities.
- Variation in behavior due to different input parameters is difficult to account for. This requires analysis of all possible paths of control flow, which is not possible for many situations without providing additional information

describing the relation between input data and program behavior.

It is foreseen that these problems of traditional approaches must also be solved for performing worst-case execution time estimation at the architectural level.

5.2 Memory consumption evaluation.

In many cases, analysis of memory consumption is needed to reason about the feasibility of an embedded system. The allocation of memory can be dynamic or static. Static memory allocation is performed at compile- or load-time, while dynamic memory allocation is performed during run-time. For most real-time operating systems, the memory layout of an application can be presented as follows:

1. Statically allocated memory: the image of program code, static data, stack, and heap
2. Dynamically allocated memory: stacks (for different threads), data objects allocated in the heap.

Analysis of memory availability for the static allocation mechanism is trivial in most cases. It is enough just to summate the sizes of all memory blocks needed for all tasks and compare the result with the amount of available system memory. However, this holds only for binary components.

The situation worsens when dealing with the mechanisms for dynamic memory allocation. In this case, the phenomenon of fragmentation can be observed. Usually, the fragmentation is caused by interleaved sequence of memory block allocations and de-allocations with greatly varying block size. It is rather difficult to evaluate the impact on memory allocation induced by the fragmentation. Moreover, having the memory shared between different tasks results in additional interference that makes the memory behavior even less predictable.

The most common practice for hard real-time systems is to avoid the use of dynamic memory management to increase the predictability and efficiency. Instead, data is allocated statically. Nevertheless, some research on the evaluation of dynamic memory allocation has been done, e.g. in [25] by Zorn et al. Their method employs synthetic allocation traces: the allocation trace of an actual program is modeled with a stochastic process. This method is reported to provide results with 80% accuracy. Thus, it might be applicable for the early analysis of worst-case dynamic memory consumption, as more precise estimations are not needed during the architecting phase. Another approach is based on the abstract interpretation theory; this method automatically transforms a high-level language program into a function calculating the worst-case usage of stack and heap space (see [22]). This approach might also be applicable during the architecting.

6. CONCLUSION

The foreseen framework for component-based software architecting is supposed to address the following: (1) reasoning about *composability of behavior*, (2) the early *assessment* of quality attributes.

For the specification of component *behavior,* timed automata are suggested, while the specification of component interaction is described with Message Sequence Charts (MSC) which allow to vividly represent timing constraints in the end-to-end manner.

For the *assessment* of timeliness, two alternative classes of techniques were considered: analytic and constructive techniques. The applicability of these techniques in the context of component-based software architecture is being validated with industrial case studies.

Currently, we are working on the integration of the proposed architecture description techniques with the evaluation techniques for the analysis of timeliness and memory consumption. Here, timed-automata-based techniques are especially promising, as they have the same formal basis both for the evaluation of quality attributes and the description of component behavior.

There are a number of challenging topics for further research:

- To find an appropriate level of abstraction for component behavior description. There should be a balance between the accuracy of the description (relevant parts of behavior are not omitted) and the complexity of the evaluation
- To elaborate a method for the specification of resource consumption. It is important to be able to integrate the specification of resource consumption in the description of components to enable the evaluation of quality attributes
- To develop a formal methodology for the evaluation of worst-case memory consumption.
- To develop a method for relating MSC-based descriptions to timed-automata-based ones.

7. REFERENCES

[1] G. Abowd, L. Bass, R. Kazman, M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architecture," in Proceedings of the 16th International Conference on Software Engineering, Italy, May 1994

[2] R. Allen and D. *Garlan: A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology, 6(3):213---249, July 1997.

[3] R. Alur, D.L. Dill. *A Theory of Timed Automata*. in: Theoretical Computer Science Vol. 126, No. 2, April 1994, pp. 183-236.

[4] M. Barbacci, S. J. Carriere, P. Feiler, R. Kazman, M. Klein, H. Lipson, T. Longstaff, and C. Weinstock, "Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis", Technical Report CMU/SEI-97-TR-029, 1998

[5] A. Burns. *How to Verify a Safe Real-Time System*. The Application of Model Checking and a Timed Automata to the Production Cell Case Study. Technical report, Real-Time System Research Group, Department of Computer Science, University of York, 1998

[6] A. David, UPPALL 2k: Small Tutorial, http://www.docs.uu.se/docs/rtmv/uppaal/tutorial.pdf

[7] A. Fehnker. *Scheduling a steel plant with timed automata*. In Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA99), pages 280-286. IEEE Computer Society, 1999

[8] D. Giannakopoulou, J. Kramer and S. Cheung, Analysing the Behaviour of Distributed Systems using Tracta. Journal of Automated Software Engineering, special issue on

Automated Analysis of Software. Vol. 6(1) pp. 7-35., January 1999

[9] D. K. Hammer and M.R.V. Chaudron, Component Models for Resource-Constraint Systems: What are the Needs?, Proc. 6th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), Rome, January 2001.

[10] D.K. Hammer, Component-based architecting for distributed real-time systems: How to achieve composability?, Int. Symposium on Software Architectures and Component Technology (SACT), Enschede, Netherlands, January 2000, to be published by Kluwer.

[11] D. I. Katcher, S. S. Sathaye, J. K. Strosnider. Fixed priority scheduling with limited priority levels. IEEE Transactions on Computers, 44(9):1140--1144, 1995

[12] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, H. Gonzalez, *A Practitioners Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer Academic Publishers, 1993

[13] K. J. Kristoffersen, K. G. Larsen, P. Pettersson, and C. Weise, Experimental Batch Plant - VHS Case Study 1 Using Timed Automata and UPPAAL, *Deliverable of EPRIT-LTR Project 26270 VHS* (Verification of Hybrid Systems), 1999

[14] P.B. Ladkin, S. Leue, Interpreting message flow graphs, Formal Aspects of Computing, 7(5):473-509, 1995

[15] G.T. Leavens, M. Sitaraman, Foundations of component-based systems, Cambridge University Press, 2000.

[16] J. Lehoczky, L. Sha, Y. Ding, "*The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*," IEEE Real Time Systems symposium, 1989

[17] C. Liu, J. Layland, "*Scheduling Algorithms for Multiprogramming in Hard Real Time Environment*", JACM, 1973

[18] J. Magee, N. Dulay, S. Eisenbach, J. Kramer. *Specifying Distributed Software Architectures*. In Proceedings of 5th European Software Engineering Conference, Spain, 1994

[19] R. van Ommering, F. van der Linden and J. Kramer and J. Magee, The Koala Component Model for Consumer Electronics Software. Computer 33, 3 (2000), pp 33-85, 2000

[20] M.A. Reniers, Message Sequence Chart, Syntax and Semantics, Ph.D. thesis, TUE, 1999

[21] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[22] L. Unnikrishnan, S. D. Stoller, Y. A. Liu. *Automatic accurate stack space and heap space analysis for high-level languages*. Technical Report TR 538, Computer Science Department, Indiana University, Feb. 2000

[23] A. Wall, C. Ericsson, and W. Yi. *Timed Automata as Task Models for Event-Driven systems*. In Proceedings of RTSCA 99. IEEE Press, 1999.

[24] J. Warner, A. Kleppe, "The Object Constraint Language", Addison Wesley, 1999.

[25] B. Zorn, D. Grunwald. *Evaluating models of memory allocation*. ACM Transactions on Modeling and Computer Simulation, 1(4):107--131, 1994.