Formal Specification of Non-functional Properties of Component-Based Software

Steffen Zschaler

Dresden University of Technology
steffen.zschaler@inf.tu-dresden.de

Abstract. Non-functional or extra-functional properties of a software system are at least as important as its somewhat more classical functional properties. They must be considered as early as possible in the development cycle in order to avoid costly failures. This is particularly true for a modern component-based approach to software development. In this paper we show a formal specification of timeliness properties of a component-based system, as an example for a formal approach to specifying non-functional properties. The specification is modular and allows reasoning about properties of the composed system.

1 Introduction

It is now widely recognized that the so-called non-functional or extra-functional properties of a software system are at least as important as its somewhat more classical functional properties and that, therefore, they must be considered as early as possible in the development cycle in order to avoid costly failures [1]. Operating systems research—especially research in the area of real-time systems—performance analysis and prediction research, and research in security have produced a wealth of results describing how to analyse, predict, or guarantee selected non-functional properties of applications (cf. e.g., [2, 3]). However, all these approaches work at a rather low level—for example, they use concepts like tasks, periods, memory pages, queuing networks, etc. These concepts on their own are not sufficient to model today's increasingly complex software systems.

Component-based software engineering [4] offers a way to partition complex systems into well-defined parts. The relevant properties of these parts are precisely specified, so that these parts in principle can be developed independently and assembled at a later time, and possibly even by a different person than the original developer. Functional issues of component-based software engineering have been well investigated (e.g., [4, 5]), but very little research has been performed concerning non-functional properties. For example, the higher-level component-based concepts still have to be mapped to the lower-level concepts from real-time system research.

In our work, we investigate this mapping by providing a semantic framework which explains how the different parts of a component-based system work together to deliver a certain service with certain non-functional properties. In a previous paper [6] we have defined the basic concepts of this framework. In this paper we show how these concepts

can be expressed formally, explaining important structuring techniques for specifications of non-functional properties. We use an example describing timeliness properties of a simple component system.

We first give a very short overview of the overall framework in Sect. 2, before the main section of this paper—Sect. 3—discusses the example in some detail. The paper closes with an overview of related work, a summary and an outlook to future work.

2 System Model

In this section we give an overview of the system model underlying the semantic concepts for non-functional specifications. This serves as an introduction to our approach, and is a very condensed version of [6].

Users view a system in terms of the *services* it provides. They do not care about how these services are implemented, whether by monolithic or component-structured software. A service is a causally closed part of the complete functionality provided by a system. *Components* provide implementations for services. As has been pointed out in the literature [7, 8] a component can provide implementations for multiple services. In addition, services can be implemented by networks of multiple cooperating components. In order to provide a service the system needs certain *resources*. A resource can essentially be anything in the system which is required by an application (see e.g., [3, 9], in our case the "application" includes both components and the container). The most important properties of a resource are that it can be allocated to, and used by, applications, and that each resource has a maximum capacity. We do not consider resources with unlimited availability, because they do not have any effect on the non-functional properties of an application. The components implementing a service require a runtime environment to be executed. We call this runtime environment the *container.* The container manages and uses the components, and requests resources, such that it can provide the services clients require.

We distinguish two types of non-functional properties:

1. *Intrinsic properties* describe component implementations. They can be meaningfully expressed without considering how the component is used.
2. *Extrinsic properties* are the result of using components with certain intrinsic properties. They express the user's view on a system, including effects of resource availability, container decisions, and so on.

3 A Formal Example

In this section we are going to examine in detail a formal specification of an example system. This is a very simple system, which consists of one component providing one service through one operation. The only relevant non-functional properties are response time of the service (required to be less than 50ms) and worst case execution time of the component (known to be 20ms); the only relevant resource is a CPU. The container simply attempts to create a single instance of the component and to use it to serve requests. The CPU available is scheduled by rate monotonic scheduling (RMS, [2]). We

use extended temporal logic of actions (TLA$^+$, [10]) to describe our system. The logic is a temporal logic where states are represented as values assigned to state variables. An expression containing primed variables (e.g., a') is an action and constrains a state transition. As usual $\Box$ means for all states. The operator $\stackrel{+}{\triangleright}$ is a special type of implication which is useful for rely–guarantee specifications [11]. $A \stackrel{+}{\triangleright} B$ essentially asserts that the system will guarantee B *at least as long as* the environment guarantees A. Because the specifications are pretty long already for this simple example we will only include the salient parts in this paper.[1]

We begin by modelling the response time property of the service. To be able to express our requirement we must first define *response time of a service* before we can use this term to express constraints on the response time. We define response time using two layers of specifications, so that the complete property will be expressed by three layers of specifications:

1. A specification defining a so-called *context model* for the subsequent response time definition. This essentially models the features of a "service" that are relevant to defining response time of a service. We call this the *context model layer.*
2. In a second specification we add history variables [12] to the context model specification to define the meaning of response time. Because we use history variables the original behaviour defined by the context model is not changed: We are merely adding probes which measure certain aspects of this behaviour. Separating this in a module of its own allows us to reuse context models for defining different extrinsic properties. For example, we can use different meanings of response time: the time from sending of a service request from the client to reception of the response by the client or the time from reception of a service request by the server to the sending of the corresponding response, resp. We have shown in another publication [13] that both interpretations can be formally specified using the same context model. We call this layer of specifications the *measurement layer.*
3. Finally we specify the actual system under consideration. This means, we write down concrete constraints over response time of the service, expressing our requirements on the system to be built. We call this the *system layer.*

Here, the first two layers together comprise the definition of response time. The third layer applies response time to a specific system, thus defining concrete constraints with concrete upper bounds for the response time of the service under consideration.

Figure 1 shows the formal specification of the context model for a service. This specification uses two variables $unhandledRequest$, a BOOLEAN indicating whether a request is in the system, and $inState$ a variable giving the current state of the service which can be either $Idle$—that is, the service is not doing anything—or $HandlingRequest$ if the service is currently working on a request. The remaining specification is very much a canonical TLA$^+$ specification. Note that it is an open specification of a service recognisable by the formula $Service \triangleq EnvSpec \stackrel{+}{\triangleright} ServiceSpec$ (line 34) stating that a $Service$ must only fulfil $ServiceSpec$ at least as long as the environment adheres to $EnvSpec$.

[1] Interested readers can download the complete specification from http://www.inf.tu-dresden.de/~sz9/publications/NfC04/

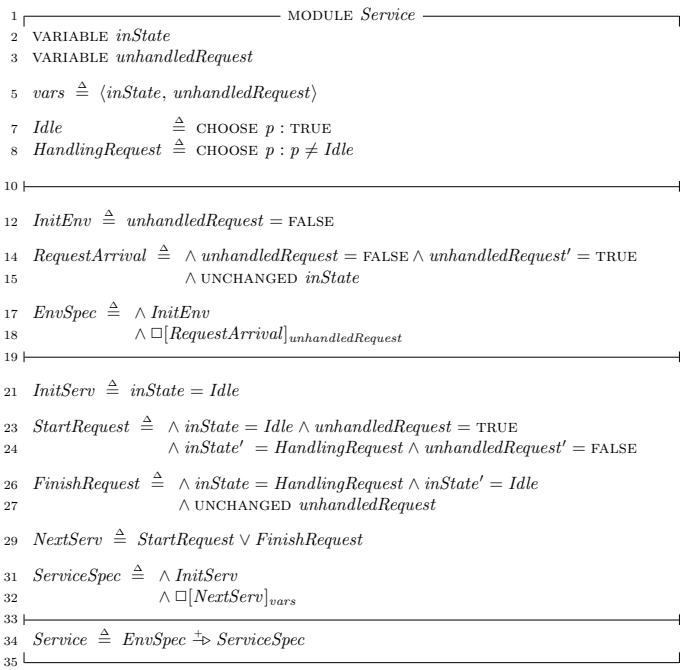

Fig. 1. Context model for a service offered by a system. The specifications have been abridged for space considerations.

The actual specification of the response time measurement is given in a separate module—shown in Fig. 2—instantiating the service definition from Fig. 1 as $Serv$ on line 6. In TLA$^+$, instantiation makes every symbol X defined in module $Service$ available as $Serv!X$ in module $ResponseTimeConstrainedService$. The specification is a canonical specification, however the individual actions have a special form and are combined by conjunction. Each action is written as an implication from an action of the original service specification to a conjunction of consequences affecting only the history variables $LastResponseTime$ (the response time measured for the last request serviced) and $Start$ (a helper variable). Finally this specification is conjoined to the original service specification, which in effect adds the consequences to the actions of $Service$ as can be verified by simple TLA$^+$ reasoning. The module $RealTime$ defines formula $RTnow$ and variable now, which is a representation of real time as described in [12].

Finally, the requirement on our actual service is expressed by the formula

$$\Box(LastResponseTime \leq 50)$$

assuming that we use now to measure time in milliseconds.

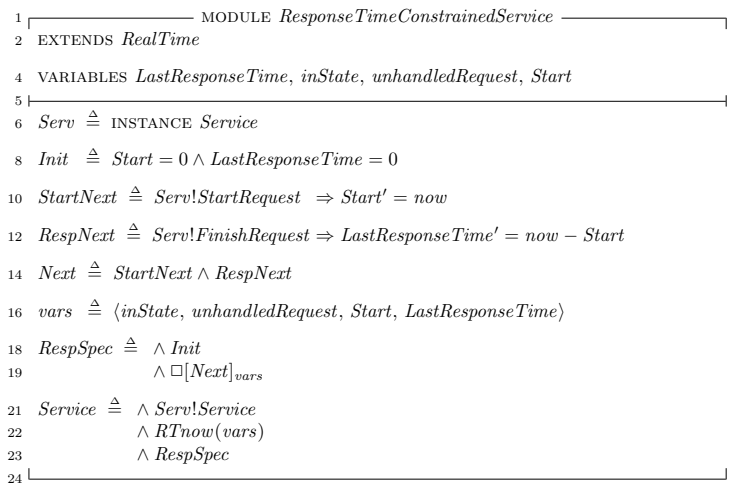

Fig. 2. Definition of response time

The specification of execution time of a component's operation mainly differs by providing an additional value for the $inState$ variable, namely $InEnvironment$, which signifies that the component would normally handle a request, but cannot do so, because it does not have access to all the resources it requires or has handed some subtask to another component. The definition of execution time takes this into account, counting only the time when the component is actually executing. Apart from this difference the structure of the component specification is similar to that of the service specification. We will therefore not discuss this specification in full detail in this paper.

The specification of the CPU is also very similar in structure. The context model layer describes what a CPU does by providing one variable $AssignedTo$ which in turn receives any value between 1 and $TaskCount$. The measurement layer then adds history variables which measure the amount of time per period allocated to each task scheduled on the CPU. Finally, there is a system layer specification which describes the specifics of a CPU scheduled by RMS. This part of the specification is a bit different from the specifications we have seen so far, because the specific constraints for a resource look different than those for components or services:

$$\Box Schedulable \stackrel{+}{\triangleright} \Box TimedCPUSched!ExecutionTimesOk$$

The specification consists of two parts asserting that as long as the schedulability criterion $Schedulable$ is fulfilled (i.e., the CPU's *capacity* is not exceeded), all tasks scheduled will meet their respective deadlines. $Schedulable$ is the well-known schedulability criterion for RMS scheduled CPUs (see [2] for further discussion).

The specification of the container is fundamentally different from the specifications we have seen before. A container specification is essentially a big implication, expressing that

If

1. certain components are available,
2. certain resources (e.g., CPU) enable certain aspects (e.g., execution of a given number of tasks with given execution times, periods, and deadlines), and
3. the system environment obeys certain rules

then the container will ensure that the system as a whole has a certain property.

We use a very simple container specification: The container expects a component with an execution time (given by the parameter $ExecutionTime$) of less than the expected response time (given by the parameter $ResponseTime$) to be available:

$$\begin{array}{l} \wedge\ ExecutionTime \leq ResponseTime \\ \wedge\ ComponentMaxExecTime(ExecutionTime) \end{array}$$

Furthermore it expects the CPU to be able to schedule exactly one task with a period (and deadline) equal to the required response time, and a worst case execution time equal to the one given as a parameter:

$$\begin{array}{l} CPUCanSchedule(1, \\ \qquad\qquad [n \in \{1\} \mapsto ResponseTime], \\ \qquad\qquad [n \in \{1\} \mapsto ExecutionTime]) \end{array}$$

Finally, it expects the environment to send requests with at least $ResponseTime$ time units between requests: $MinInterrequestTime(ResponseTime)$. These expectations are combined by conjunction into formula $ContainerPreCond$. If all these pre-conditions hold, the container can use exactly one instance of the component to provide the service. This is expressed in the conclusion of the specification:

$$\begin{array}{lll} ContainerPostCond & \triangleq & \wedge\ ServiceResponseTime(ResponseTime) \\ & & \wedge\ \Box\wedge\ TaskCount = 1 \\ & & \qquad \wedge\ Periods = [n \in \{1\} \mapsto ResponseTime] \\ & & \qquad \wedge\ Wcets = [n \in \{1\} \mapsto ExecutionTime] \\ & & \wedge\ \Box(CmpUnhandledRequest = EnvUnhandledRequest) \end{array}$$

$ComponentMaxExecTime$, $CPUCanSchedule$, $MinInterrequestTime$, and $ServiceResponseTime$ are defined in the container specification using the respective measurement layer specifications This means that the container does not assume concrete resources, components or services. The complete container specification is then $ContainerPreCond \stackrel{+}{\triangleright} ContainerPostCond$.

Now we have specified all the individual parts of our system and it only remains to put them together. Fortunately, this is very simple for a TLA$^+$ specification, because "composition is conjunction" [14]. Composing component, CPU, and container specification we arrive at a specification of our $System$. We want to show that our system

is *feasible*, that is, that it has sufficient resources and a component and container implementation available to provide services with a maximum response time of 50ms. Specifically, we expect our system to adhere to the following specification:

$$ExternalService \triangleq Environment(RequestPeriod) \stackrel{+}{\triangleright} Service(50)$$

that is: as long as the environment sends request with a minimum time distance of $RequestPeriod$, the system will provide services with a maximum response time of 50ms. We can formalise our proof obligation by

$$IsFeasible \triangleq System \Rightarrow ExternalService$$

and use the composition theorem from [14] to prove this property. For lack of space we cannot show the proof in this paper, but it is relatively straight forward. We need to pose one additional constraint on the $RequestPeriod$ parameter, namely that it is greater or equal the expected response time.

4 Related Work

In his thesis [15] Aagedal defines CQML, a specification language for non-functional properties of component-based systems. The definition remains largely at the syntactic level, semantic concepts are mainly explained in plain English without formal foundations. Staehli [16] describes a formal technique for specifying non-functional properties of multimedia presentations. As an extension and combination of these efforts, the two authors recently and independently of our research published a short paper on "QoS Semantics for Component-Based Systems" [17]. Their work is restricted to timeliness and data quality properties and does not cover resource demand at all. In contrast, we use more abstract definitions which cover any kind of measurement, including but not limited to timeliness and data quality. Also, resource demand and resource allocation is a central element of our semantic domain. The authors have so far presented only an intuitive explanation of their approach, but no formal specifications yet.

Hissam et al. [18] describe a prediction-enabled component technology (PECT). This work is very similar to our work in that it attempts to provide a framework in which specific analysis methods and specific component models can be combined. However, their work is somewhat more abstract. Also, they seem to be exclusively concerned with modularisation into components, whereas our work explicitly takes into account the container and resources as an important yet separate part of an overall system. Bertolino, Mirandola and Vincenzo [19, 20] presented work attempting to merge techniques from software performance engineering with component-based software engineering. They distinguish two model layers: the software model which represents the logical component structure of a system, and the machinery model which models properties relevant for performance analysis. In contrast to our work they are more interested in performance predictions than in feasibility analysis.

5 Conclusions

In this paper we have shown an example of formal specification of timeliness properties of a simple component-based system, based on the semantic framework defined in [6].

The main contribution of this paper is to show how these concepts can be expressed formally, using TLA$^+$. Our style of specifying allows us to reason about the response time of a system constructed from a specified resource, component and container. An important feature of these specifications is that they can be written independently and later be assembled to a specification of a complete system. This allows us to 'divide and conquer' the specification problem, potentially even by having different people write different parts of the specification. For example, middleware vendors can write container specifications, and component developers can write component specifications. Application assemblers finally compose all specification into a system specification. It is important to point out that timeliness is only used as an example, the approach is also appropriate for other properties.

Of course, there also remain some questions to be answered. There are three main directions in which we will extend this approach in our ongoing research:

1. We will provide support for services implemented by networks of components. We intend to solve this by providing a more complex container specification, which takes into consideration abstract descriptions of component networks, or *architectural constraints*—essentially providing a characterisation of classes of applications (resp. their topologies) the container can support. Based on this the container specification can then express how the components and resources are used to provide for the required non-functional properties.
2. We will provide support for specifying constraints on multiple non-functional properties. While this could already be done with the approach described in this paper, the challenge here is to do it in a modular way, so that every non-functional property can be specified independently and the specifications can be merged to provide a specification of the complete system.
3. We have only talked about abstract systems in this paper. In order to apply our approach to concrete systems with concrete components and services (e.g., a stock quoting application), we need to define mappings between context models and application models and to apply them to our specifications. The issue of mappings between context models has been treated in another paper together with Simone Röttger [13].

The formalism we introduced in this paper and in [6] merges component-based software engineering and specification of non-functional properties. It thus allows the non-functional properties of a component-based system to be considered and analysed as early as possible in the design process, and thereby helps avoid costly mistakes. Because the individual parts of the specification can be written independently, the approach also supports a component-off-the-shelf market, where components are developed and formally described by parties independent of the party who combines them to form an application.

Acknowledgements

I would like to thank Heinrich Hussmann, Sten Löcher, and the reviewers for their comments which helped me get my ideas and their presentation straight. This work was funded by the German Research Council as part of the COMQUAD project.

References

1. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. The Kluwer international series in software engineering. Kluwer Academic Publishers Group, Dordrecht, Netherlands (1999)
2. Liu, J.W.S.: Real-Time Systems. Prentice Hall, NJ (2000)
3. Tanenbaum, A.S.: Modern Operating Systems. 2nd edn. Prentice Hall (2002)
4. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Publishing Company (1997)
5. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison Wesley Longman, Inc. (2001)
6. Zschaler, S.: Towards a semantic framework for non-functional specifications of component-based systems. In Steinmetz, R., Mauthe, A., eds.: Proc. EUROMICRO Conf. 2004, Rennes, France, IEEE Computer Society (2004)
7. Krüger, I.H.: Service specification with MSCs and roles. In: Proc. IASTED Int'l Conf. on Software Engineering (IASTED SE'04), Innsbruck, Austria, IASTED, ACTA Press (2004)
8. Salzmann, C., Schätz, B.: Service-based software specification. In: Proc. Int'l Workshop on Test and Analysis of Component-Based Systems (TACOS) ETAPS 2003. Electronic Notes in Theoretical Computer Science, Warsaw, Poland, Elsevier (2003)
9. Gościński, A.: Distributed Operating Systems: The logical design. Addison-Wesley Publishers Ltd. (1991)
10. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
11. Jones, C.B.: Specification and design of (parallel) programs. In Manson, R.E.A., ed.: Proceedings of IFIP '83, IFIP, North-Holland (1983) 321–332
12. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM ToPLaS **16** (1994) 1543–1571
13. Röttger, S., Zschaler, S.: Model-driven development for non-functional properties: Refinement through model transformation. In: Proc. <<UML>> Conf. (2004) To appear.
14. Abadi, M., Lamport, L.: Conjoining specifications. ACM ToPLaS **17** (1995) 507–534
15. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
16. Staehli, R., Walpole, J., Maier, D.: Quality of service specification for multimedia presentations. Multimedia Systems **3** (1995)
17. Staehli, R., Eliassen, F., Aagedal, J.Ø., Blair, G.: Quality of service semantics for component-based systems. In: Middleware 2003 Companion, 2nd Int'l Workshop on Reflective and Adaptive Middleware Systems. (2003)
18. Hissam, S.A., Moreno, G.A., Stafford, J.A., Wallnau, K.C.: Packaging predictable assembly. In Bishop, J., ed.: Proc. IFIP/ACM Working Conf. on Component Deployment (CD 2002). Volume 2370 of LNCS., Berlin, Germany, Springer-Verlag (2002) 108–126
19. Bertolino, A., Mirandola, R.: Towards component based software performance engineering. In: Proc. 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction at ICSE 2003, ACM/IEEE (2003) 1–6
20. Grassi, V., Mirandola, R.: Towards automatic compositional performance analysis of component-based systems. In Dujmović, J., Almeida, V., Lea, D., eds.: Proc. 4th Int'l Workshop on Software and Performance WOSP 2004, California, USA, ACM Press (2004) 59–63