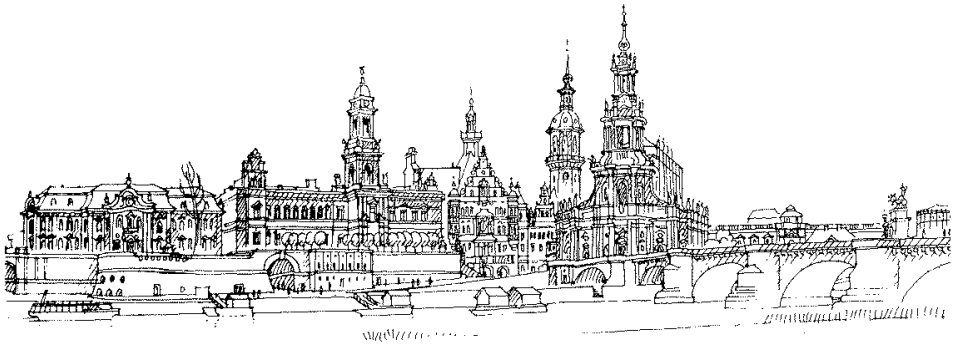
TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Technische Berichte
Technical Reports
ISSN 1430-211X

TUD-FI04-12-Sept.2004
Jean-Michel Bruel, Geri Georg, Heinrich Hussmann, Ileana Ober, Christoph Pohl, Jon Whittle, Steffen Zschaler (eds.)

Proceedings of the 1st International Workshop on Models for Non-functional Aspects of Component-Based Software



Technische Universität Dresden
Fakultät Informatik
D−01062 Dresden
Germany
URL: http://www.inf.tu−dresden.de/

Preface

Jean-Michel Bruel[1], Geri Georg[2], Heinrich Hussmann[3], Ileana Ober[4], Christoph Pohl[5], Jon Whittle[6], and Steffen Zschaler[5]

[1] Laboratoire d'Informatique, University of Pau, B.P. 1155, F-64013 Pau, France,
bruel@univ-pau.fr
[2] Computer Science Department, Colorado State University, Fort Collins CO 80523, USA,
georg@cs.colostate.edu
[3] Institut für Informatik, Universität München, Amalienstraße 17, 80333 München, Germany,
hussmann@informatik.uni-muenchen.de
[4] UMR Verimag, Centre Equation, 2, avenue de Vignate, 38610 Gi-Ares, France,
ileana.ober@imag.fr
[5] Fakultät Informatik, Technische Universität Dresden, 01062 Dresden, Germany,
christoph.pohl|steffen.zschaler@inf.tu-dresden.de
[6] NASA Ames Research Center, MS 269-2, Moffett Field, CA 94035, USA,
jonathw@email.arc.nasa.gov

After the successful "Workshop on Quality of Service in Component-Based Software Engineering" [1] in Toulouse in 2003, we decided to continue organising workshops on this theme—and we were richly rewarded by interesting papers and good discussions. These proceedings collect the papers accepted for the 2004 workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04), which was co-located with the 7th UML conference in Lisbon, Portugal. The programme committee selected four papers out of five submissions, which gave us sufficient time for a real workshop atmosphere, fostering lots of interesting discussions. In this preface to the workshop proceedings we want both to give a short overview of the papers you can expect to find in the proceedings and to summarise the discussions that took place on site.

1 Overview of Papers Presented

In this section, we give summaries of each accepted paper. The corresponding presentation slides for all papers have been made available from the workshop's website www.comquad.org/nfc04. All of the papers presented looked at the workshop's theme from very different angles. We will use this motivation for a first classification of the papers in the following.

1.1 Non-functional Aspects Management for Craft-Oriented Design

The authors of this paper discuss a technology to combine different view-points used by different teams (or 'crafts') developing different parts of a complex application. The key idea is in the introduction of a so-called 'pivot'-element, which serves as an interface between the different models of the different teams. Each team together with project management decides which part of its models is to be public and how these models are to be represented. Thus, teams can work largely independent of each other, while still

enabling project management gain a global overview of the system and ensure overall consistency. The major motivation for this work lies in the development process, namely in the support for diverse teams cooperating in a large project.

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

This paper presents a formal specification of timeliness properties of component-based system, as an example for a formal, measurement-based approach to specifying non-functional properties. The approach is motivated by the separation of roles in component-based software development and uses separate specifications for components, containers, system services and resources. The specification is modular and allows reasoning about properties of the composed system. As the previous paper, this paper's main motivation is in the development process, however, it is more driven by the different needs of roles such as the component developer and the application assembler.

1.3 A Model-driven Approach to Predictive Non Functional Analysis of Component-based Systems

This paper discusses an idea to use model transformation to refine a platform-independent model (PIM) into an analysis model in addition to the platform-specific model (PSM) driving development towards implementation. The authors discuss relevant relationships between the two different refinement paths (PIM to PSM and PIM to analysis model), and describe in more detail a refinement leading to a queueing network model for average-case compositional performance analysis. A major motivation for this paper is in the requirement to analyse non-functional properties of the system under development based on models of it.

1.4 Tailor-Made Containers: Modelling Non-functional Middleware Services

This paper discusses the generation of tailor-made application servers from specifications of the non-functional properties to be supported. The authors suggest to model application servers as a collection of core services and aspects implementing support for individual non-functional properties. The idea is then to generate application servers from these parts, depending on the non-functional specifications of the components or applications to be executed. This paper is mostly motivated by experiences the authors made when providing runtime support for realtime properties.

1.5 OMG Deployment and Configuration of Distributed Component-Based Applications

Another important motivation for modelling non-functional properties of component-based applications lies in the actual deployment of such applications, which may necessitate reconfiguration of individual components or the complete application. As this important area had not been covered by any of the submissions, we invited Francis

Bordeleau—who is one of the co-authors of the OMG deployment and configuration specification—to give a presentation on the concepts in this specification and how these relate to models of non-functional properties of component-based software. The talk was very well received. Its main message was that although the current specification does not specifically address non-functional properties, these must be considered an important ingredients in particular to reconfiguration decisions for specific target platforms or user groups.

2 Workshop Results

In this section we are going to summarise, and report on, the discussion which where inspired by the papers presented. There are several ways that separation of concerns surfaced in the papers: The paper on craft-oriented design separated the concerns of different specialists working together towards a common goal. Achieving independence of different roles in the development process was one motivation behind the paper on formal specification of timeliness properties, while the paper on a model-driven approach to prediction separated concerns of construction vs analysis of systems. Finally, the paper on tailor-made containers separated different non-functional aspects.

A number of questions have been discussed in the afternoon sessions:

- Domain limits: This question was about defining the domain of research. In particular, we discussed terminology issues, such as what is a non-functional property?
- Where should we put support for non-functional properties? And, when in the development process do we need to consider them?
- Is there a difference between the concept of component and the concept of resource? Is there such a thing as a resource component?
- What are the different kinds of composition that are of relevance?

We'll summarize the outcome of the discussions in the following subsections.

2.1 Domain limits

After much discussion—especially on the definition of non-functional properties—we arrived at the following understandings.

- By starting from the requirements down to the code, functional properties are those identified first.
- A property is not functional or non-functional by itself. It depends on the point of view, or intent of the system. The functionality label is dependent on the client of this property. For example, the security feature of a communication line can be functional for a certain system, but not for another one.
- Clearly there is still a wide view of what a non-functional issue is.

2.2 Support for non-functional properties

The consensus here was that

- Non-functional properties need to be considered throughout the entire development process.
- For verification purposes, many non-functional properties require separate analysis models to be constructed (information comes both from the development models and non-functional properties expertise) and analysed. The results of such analyses need to feed back into the development process. Most of the time, the properties are verified at several steps of the development process and thus at several levels of abstraction. Analysis results and models can in general not be maintained over functional refinement steps, but rather must be recreated at each different level of abstraction.
- Representation of non-functional properties changes with the stage of development process. For example, properties are expressed very explicitly in the requirements, but can be represented by certain structures in the architecture, or by a middleware configuration. This requires sophisticated notions of refinement and traceability.

2.3 Resources vs Components

Some of the talks where considering resources as components, while others separated the two concepts. This started a discussion on these notions. The general conclusion was that there is no formal difference between components and resources. However, it is practical to distinguish them for hiding implementation details and complexity. They can be distinguished based on usage, where resources represent an encapsulation of lower-level components.

2.4 Composition

We identified four different kinds of composition, but left this for further discussion:

1. What are the semantics of composition for models vs components executing in a container?
2. How do the properties of individual components contribute to the properties exhibited by the composition?
3. Is there a semantic difference between composing components for an application vs middleware components (containers)?
4. What are the semantics of composition when the constituent elements are not orthogonal? For example, how can we compose models of one component dealing with security properties of this component and with response time properties of the same component?

3 Conclusion

While non-functional properties especially of component-based software is still an open field, we did reach consensus on some points. This encourages us to continue the series of workshops.

There is still work to do:

- Industry is still waiting for an easy way to annotate models with one generally accepted and known notation for non-functional properties.
- The role of standardizing committees in this process needs to be understood. Industry and academia are expecting them to provide a framework for thinking about non-functional properties in research and application. Concrete case studies are needed to evaluate the usefulness of standards. Finally, standardization may need to be more domain-specific.

We hope to continue this series of workshops in the future.

Acknowledgements

This work has been partially funded by the French National Science Fund (FNS/ACI/JC-9067) and the German Research Council (DFG FOR 428/COMQUAD).

References

1. Jean-Michel Bruel, editor. *Proc. 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*. Cépaduès-Éditions, June 2003.

Non-functional aspects management for craft-oriented design

and its application to embedded systems

Francois Mekerke, Wolfgang Theurer, and Joel Champeau

ENSIETA
2 rue Francois Verny
29806 BREST Cedex 9
{mekerkfr,theurewo,champejo }@ensieta.fr

Abstract. The most important phase in the development of large-scale system is integration. At model level, some techniques allow us to weave models of different parts of the system into one another so as to obtain a complete model. The problem is that these methods don't guarantee that the properties added to the system by a first aspect won't be invalidated by another one.
We present a technique that enables property checking in the framework of the craft-oriented partition that is the norm in industrial development processes.
In order to validate the consistency of a model distributed depending on the specialities of the teams that have to realise it, we suggest to use an abstract view of the system we call the "pivot" through which craft-oriented "facets" communicate. Since it is the only link among facets, the pivot can check if the properties woven with previous aspects are still valid or not. It can therefore help manage aspects' interference.
This technique provides a good compromise between flexibility and rigour, which allows for free development while validating choices all along.

1 Introduction

As technological changes happen always faster and in an always more fragmented manner, systems' complexity tend to explode in all phases of their life cycle.

The difficulty of dealing with this problem lies not so much in the technologies themselves than in the organisation of the teams that master them, whose number and specialisation increase.

The majority of the troubles encountered by the project manager to follow the state of advance of his system are related to the management of the consistency between the different subsystems.

He must be able to form his own abstract image of the system from the partial data sets given by his partners.

We will place ourselves here in the framework of system modelling, which can help manage the complexity, at least by providing an eagle-eye view of the system. However, consistency management into a set of models remains a full-scale problem.

Partitioning approaches, especially in the MDA framework, provide solutions for the management of "Separation of Concerns" (see [1],[2]). Aspect-Oriented Design (see [3], inspired by Aspect-Oriented Programming (see [4]), or the Composition Filters (see [5]) approach, allow us to separately model functionalities on the one hand, and non-functional concerns on the other hand, then to weave the latter into the firsts.

We introduce here the concept of partitioning by "craft-oriented facets", each dealing with specific functionalities, which responds to the industrial problematics that consists in studying a same system under different lights, with different teams. Indeed, the industrial practise is such that contracting authorities partition the system into subsystems depending on the specialities of their subcontractors (internal or external). We present a methodology enabling the simultaneous management of these two partitions, through a "pivot" that controls exchanges and checks that the properties introduced by the different aspects are always validated.

2 Model management

2.1 Context

During the development of a system, numerous models are created, which each corresponds to a concern or the point of view of a stakeholder. These stakeholders work generally in an autonomous manner, independently from one another, each having its own point of view (models, requirements, specifications, hypotheses…) on the system (see [6]).
However, these points of view can be very far from one another, having very local validities. The question is then for the project manager: how to obtain and moreover maintain consistency between the different models of a system, provided by the stakeholders.
Over this first problematics comes the problematics of the management of cross-cutting concerns, for example fault tolerance or real-time performance, which aspect technologies can help to overcome. The project manager must find means to ensure that aspects are actually taken into account in the distributed model of the functionalities.
This leads to the following problem: we may have to weave every given aspect into the models of several subsystems, since they tackle a cross-cutting concern. However, to do so, we need a distributed aspect whose different parts each operate on a single subsystem.
Another solution would be to merge the local models into a complete model of the system in order to weave the aspects, but this option seems unrealistic because the global model of any system, even small, can become huge, therefore difficult to manage.
In addition to this initial problem, we have to ensure that it is possible to check the properties woven into the system. It is thus necessary to define a set of invariants that will show the persistence of certain required qualities.

2.2 Distributed aspects and invariants

Let us consider the case of an avionics company that want to obtain for each of the commands of its air-plane a fault ratio under 10^{-6} fault per hour of flight. We will also consider that there are two teams working on a command, one for software and the other for hardware.
In addition to their proper problematics (description of the command depending on other data for the firsts, dimensioning of the calculators and integration into the network for the latter), the teams must respect this constraint and coordinate a number of their choices.
If the solution chosen to respect the objective is to apply n-redundancy to functions and calculators, we have to ask the firsts to use the design pattern Main-Rescue (see [7] for the concept and a catalogue of patterns) for the functions, the latter to foresee an adequate number of computational resources, without using two functions linked by this pattern on the same processor. This implies that we give the knowledge of the functions' structure to the "hardware" team at some point of the development process.
However, as it seems counter-productive to provide them with the complete (therefore complex) model of the "software" team, we have to generate an intermediate model, which provides them with the pertinent data and nothing more.
The introduction of an aspect into the system will therefore generate (1) a data flow from a team to the other and (2) a set of invariants to respect (here for example FonctionMain.CPU != FonctionRescue.CPU). These latter can be seen as the expression of a long-term "contract" (see [8]), which will have to be respected from there on.

3 A balanced approach

So as to be able to answer the double requirement induced by the couple separation of concerns / craft-oriented partitioning, we have chosen to keep the craft-oriented partitioning structure and to provide the suitable mechanisms to implement the aspects, which implies guaranteeing their invariants over the whole life cycle.
To summarise, the result we obtain, in terms of structure, is the following: each "craft-oriented facet" presents data to a "pivot", which processes them in order to provide data to other facets or check the properties introduced at weaving time.

3.1 Facets

From now on, we will call a ”facet” both the requirements related to a craft and the models developed in their framework, as well as the physical team in charge of the domain.
Example: *In the case of the development of an air-plane, we could find the following facets: hydraulics, electronics, software, aerodynamics ...*
The facets are the heart of the system, since they are the ones that provide the development effort, and finally realise the product. They are each given their own meta-model, which allow them to provide models to the pivot, and also to manage their own development.
These meta-models manipulate two kinds of objects: (1) internal objects: developed inside the facet, all their data is available, and the facet is in charge of providing their pertinent elements to the pivot; (2) external objects: simple abstractions of objects developed in other facets, the knowledge about them comes from the pivot, they are never refined in the facet.
To realise its own part, a facet is free to use the modelling technology of its choice (if any), but one of its models has to be public. This model, that we call “Public Model”, is the interface of the facet with the pivot. The Public Model is an instantiation of a subset of the facet’s meta-model that has a double functionality: (1) it allows us to define the functional and structural specifications of the facet, and (2) it allows the facet to present its principal characteristics to the pivot whenever necessary. The meta-model of the Public Model is chosen in collaboration with the project manager. This is the only way for him to specify the data he wants to receive and add the necessary consistence between the facets. In the framework of software-intensive systems development, several facets will deal with different software parts. In this case we can consider that each facet will develop a component whose ports’ information will present in the Public Model. The Public Modelof the facet will contain (without being restricted to) the interface of the component.

3.2 Pivot

The pivot is the entity in charge of maintaining the consistency between the facets. On the one hand , some facets’ outputs have to be rewritten to be given as inputs to others, in order to allow these to coordinate each other (see 2.2). On the other hand, the pivot checks that the invariants introduced by the aspects are not violated.
Therefore it contains two kinds of information: (1) those connected to data management (source, processings, destination) and (2) those related to properties checking. The first one implies model transformations techniques, whereas the second requires skills in algebraic computation or formal methods, depending on the situation.
Invariants management is quite a hard task because of the numerous different formalisms we can use, depending on what has to be checked. Our solution to fix the problem is to introduce a set of “layers”, each one containing the information related to a particular aspect, in terms of data flow as well as in terms of invariants. The pivot is thus composed of those layers, which are pushed each time an aspect is added, and poped later in the development process, as soon as they are considered useless.
The pivot meta-model has to lie at the intersection of the facets’ ones in order to make it easier to map the elements of the pivot on those belonging to the facets.
As shown fig 1, the pivot synthesises the relationships among Public Models. It thus enables the uniform usage of heterogeneous data without having to change anything in the Public Models. The most noteworthy property of this construction is that the pivot is the unique link between the Public Models.

3.3 Layers

The layers are the cement that holds the pivot and the facets together. They are the only entities to have the knowledge of both the Public Models and the pivot. Through the abstract representation of their information, the distributed architecture of the system is transparent for them. They can simply work on some elements of the models as if they were parts of a unique homogeneous model.
A layer has the knowledge of all the elements it has to monitor. It computes either the value of abstract pivot attributes from attributes of facets’ internal objects, or the values of facets’ external objects from attributes in the pivot. Thanks to this symmetric organisation, two facets never have to (and must not) exchange information directly.
Once pivot’s attributes are computed, constraints can be checked on them.
Figure 2 shows an example of how a layer can be defined using the UML syntax. We define two main types of constraints: (1) local constraints that guarantee the consistency between facet elements and

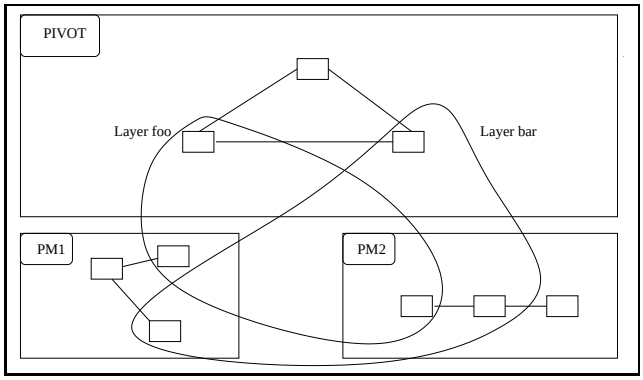


Fig. 1. Pivot general organisation

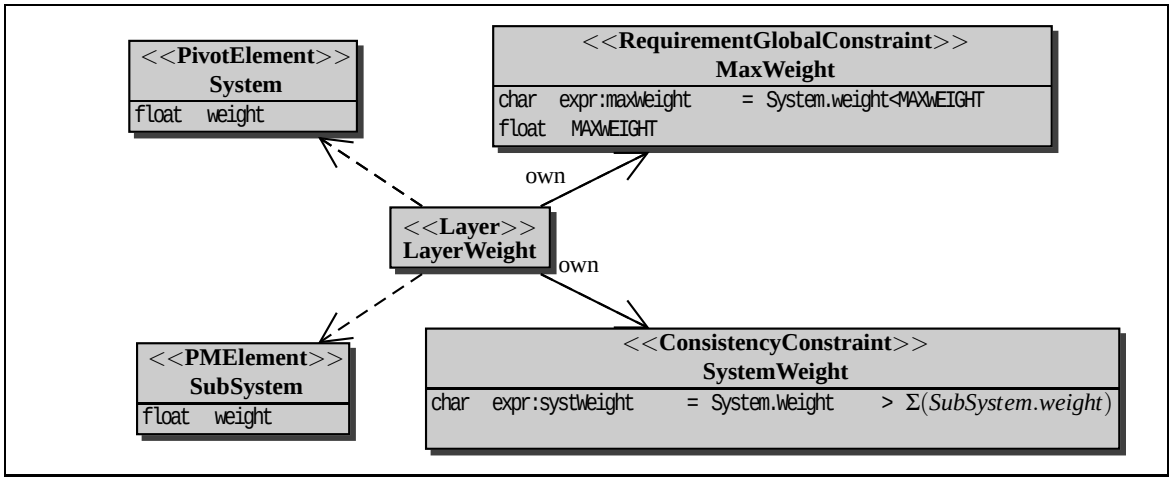


Fig. 2. Layer: example

their representation in the pivot, they also provide a way to compute the values in the pivot from those of the facets, and (2) global constraints that express global system requirements.
With this organisation, we emulate the weaving of distributed aspects, and we add the continuous checking of the sets of constraints that characterise them.
When developing collaborating components, layers will be dedicated to checking that the inputs' and outputs' formats of the components are compatible.

4 Development process

4.1 Two phases

The process induced by such a structure for managing models and their relationship can be split in two main stages: first, a phase of negotiation and then a phase we called "stable models' phase". This behaviour is close to the industrial usage, where the choice of the partners then the technological choices and, finally the tasks' allocation are negotiated.

- The first stage includes drafting a consensus on the choice of the meta-models to be used by the different stake-holders to communicate (those of the Public Models), then on the determination of the pivot meta-model. Follows a phase where the pivot own structure is defined, through analyse of the preliminary solution elements provided by the facets.
- The second stage consists in an enrichment of this structure, *i.e.* an improvement of the pivot attributes precision and a convergence towards the solution model. Whenever we have to add a new facet at this stage (which means we have forgotten it at the requirement analysis stage) we have to go back in negotiation phase. We then may have to first rebuild the pivot adding the new facet object, and add force the existent facets to change their Public Modelin order to provide the information relevant to the new one.

4.2 Two applications

Consistency The main usage of the pivot is to dynamically check the consistency of the data provided by the different facets through their Public Models. Each time a value is modified in a Public Model, all the layers related to this element are evaluated again. Whenever a constraint is violated by the new value, it is rejected, and the faulty facet is notified as well as the project manager.
Example: *If we consider two facets "software" and "hardware", the pivot can check that the CPU power provided by the "hardware" team will allow the processings defined in the "software" team to respect their real-time requirements.*
More generally, it can easily detect inconsistencies between quality of service requested and quality of service provided.
Basically, in this case, the pivot can be seen as a checker which verifies that the data stored in the Public Models satisfy the constraints' system composed of the layers.

Dimensioning As a practical application, the pivot can also be used the other way round, as a methodological tool for system dimensioning. In this case, we don't just check that a set of values satisfies the system: given values for a subset of the system's variables, we solve the system in order to obtain a set of solutions for the rest of them.
Actually, by implementing processings in the layers other than those imposed by the aspects, it is possible to add a reflexive flavour to the pivot, which could then provide elements of solution by itself. For example, by first determining the CPU power needed by a set of functions to respect their real-time specifications, the pivot could determine the appropriate set of on-the-shelf CPUs able to execute them properly.

5 Real-time embedded software

Although our approach aims at being as general as possible, and applicable to all sorts of systems, thus to an undefined number of facets of different kinds, we mainly focused on software-intensive embedded systems composed of a "software" facet and a "hardware" facet. Our choice of these two particular facets can be argued as follow:

First, most embedded (software intensive) systems, are built upon a “software+hardware” core (sensor/control/actuator chain), closed enough to be studied as a completely independent system.
It seems also interesting to show how the hardware platform can be integrated in software development.
Finally, the real-time performance aspect is a cross-cutting concern for the two facets.

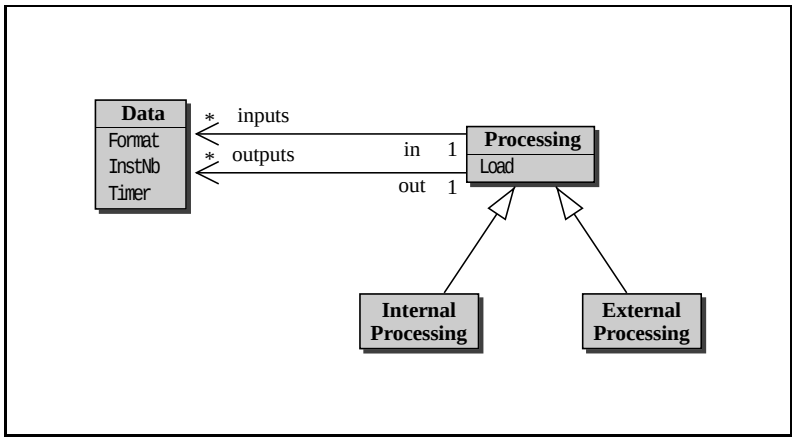


Fig. 3. “Software” facet concept model

The “software” facet is in charge of developing the whole high level software (drivers, schedulers and low level protocols are considered to be part of the hardware). Its meta-model, shown figure 3, is composed of processings, which exchange data. Each processing is viewed as a sequential unit by the rest of the system. That sets the Public Model granularity, fixed from the first stage till the end of the development.

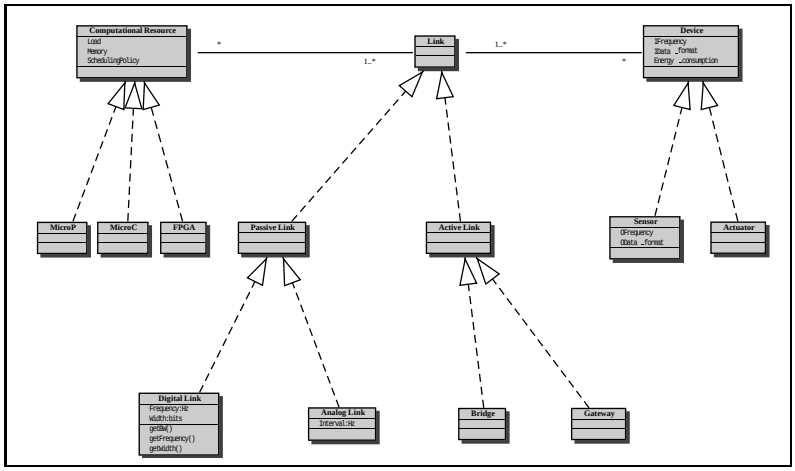


Fig. 4. “Hardware” facet concept model

The “hardware” facet describes the platform which runs the processings. The meta-model of this facet, shown 4, includes computational resources which communicate with each other or with devices via communication links. The pivot is composed of functions which exchange data on communication links.

The pivot elements describe the same entities than the facets ones, but from a different point of view.

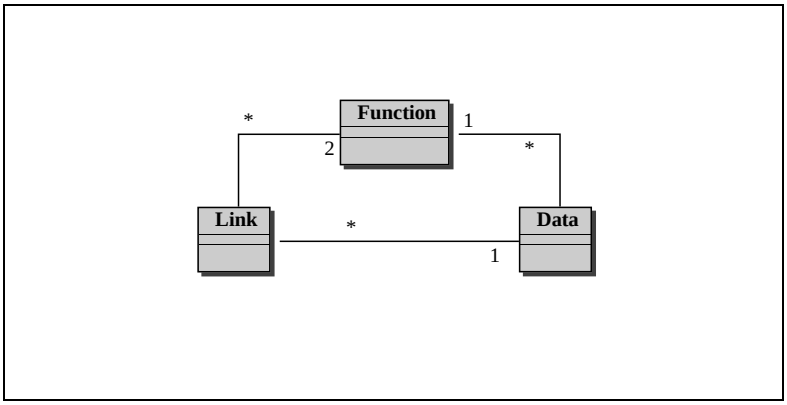


Fig. 5. Pivot concept model

5.1 Formal property validation

One of the major problems that arises in the scope of distributed modelling, is the real-time behavioural validation of the system. Actually, despite the existence of numerous formal techniques (model-checking, formal test, etc see [9]) for modelling and validating real-time softwares, evaluating the impact of the hardware platform on software is very laborious. Taking this impact into account needs a wide human intervention (because of undecidability problems see [10], [11]). These troubles are generally bypassed by introducing timing information into the models of process (like timed automata see [12]), *i.e.* building a unique model containing simultaneously behavioural (functional) information and executive information. Even if these methods have already proved themselves, they don't get along with our wish to separate models by crafts. We thus propose creating a validation layer which builds on the fly a set of communicating timed automata from information provided by the "hardware" and the "software" facets. The previously mentioned techniques can be applied on this set. This layer is based on an abstract syntax associated with an operational semantics and a translation algorithm to a chosen target formalism (UPPAAL's timed automata, for instance). The Public Model of the "software" facet has to provide its processings' behavioural models, the "hardware" facet as for it, has to give the pivot the values related to its computational resources and links (power, bandwidth ...).

The abstract syntax related to this layer is defined as the parallel composition of functions sets:

$S := \mathcal{F}$

$\mathcal{F} := \mathcal{F} \parallel_F F \mid F$

$F := < I, O, gs, ge, gd, \Delta, A, T >$

A function is a tuple $< I, O, gs, ge, gd, \Delta, A, T >$ where:

- I: the set of inputs
- O: the set of outputs
- $gs : O \rightarrow 2^{Q^+}$ (set of parts of Q^+, set of positives rational numbers) mapping that associates to any given data from the set of outputs, an interval for its emission delay (expressed in cycles from the effective launch of the function).
- $ge : I \rightarrow 2^{Q^+}$ mapping that associates to any given data from the set of inputs, an interval for its reception delay (expressed in cycles from the effective launch of the function).
- $gd \in 2^{Q^+}$: interval for the function launch delay.
- $\Delta \subseteq 2^{Q^+}$ interval for the function clock offset δ, set at system launch .
- A: untimed automata describing formally the function behaviour.
- T: the function activation period.

All delays are computed in the pivot from the quantitative information provided by the "software" facet (code size, number of instructions needed to produce a data, etc.) and the qualitative information extracted from the "hardware" facet (power, rate, etc.). The communication delays are included in the functions' input/output delays.

The operational semantics associated to this syntax, is defined as a timed labelled transitions system (TLTS) expressed by rules of the following form:

$$\frac{\text{premise}}{\text{conclusion}}$$

where premise is a conjunction of time assertions and assertions specifying the state of the automaton of the considered function.
The operational semantics, as well as the translation algorithm are not provided is this article, for readability (and size) reasons.
The set of invariants associated to this layer are formal real-time behavioral properties (deadlock-freeness, safety, liveness, etc) expressed in a temporal timed logic (such as TCTL). The set of communicating timed automata built by the translation algorithm (a process we may implement soon) has to satisfy all invariants. To check this, we will need an external model-checker for the specified formalism: we consider using UPPAAL (see [13]) for this purpose.

6 Conclusion and prospects

We have expounded a structure for managing the developments of systems involving craft-distributed models. We also explained how to make sure that cross-cutting concerns are taken into account, through a set of layers. These layers contain information on the identity of the elements they have to deal with, and the set of constraints to apply on these elements. Thus, this method allows us to maintain consistency over the different facets, through guaranteeing the validity of cross-cutting properties all along a system's life cycle.
We have already applied this method to a simple dimentioning application with some success. We now study a bigger application: we work on the modelling of an autonomous underwater "glider" developed at ENSIETA[1], we defined three facets, "hardware", "software" and "control", and a number of simple layers (QoS, mass, volume, ...).
Through this experience, our goal is to develop, in the near future, tool support for this specific application.

References

1. Hürsch, W.L., Lopes, C.V.: Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA (1995)
2. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the hyperspace approach. In: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer (2000)
3. Suzuki, J., Yamamoto, Y.: Extending uml with aspects: Aspect support in the design phase. In: Proceedings of the third ECOOP Aspect-Oriented Programming Workshop. (1999)
4. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
5. Aksit, M., Tekinerdogan, B.: Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters (1998)
6. Hilliard, R.: Using the UML for Architectural Description. In France, R., Rumpe, B., eds.: UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings. Volume 1723 of LNCS., Springer (1999) 32–48
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994) ISBN: 0-201633-61-2.
8. Meyer, B.: Applying design by contracts. IEEE Computer **25** (1992) 40–51
9. Laroussini, F.: Automates temporisés et hybrides. In: École d'Été Temps Réel (ETR2003), Toulouse (2003) 155–166
10. J. Ermont, F.B.: La vérification des systèmes temps réel soumis la préemption de processus est indécidable. In: MSR2003 (Modélisation des Systèmes Réactifs). (2003)
11. Ermont, J.: Une algèbre de processus pour la modélisation et la v érification des systèmes temps-réel avec préemption. PhD thesis, ENSAE (2002)
12. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
13. Larsen, K., Pettersson, P.: Timed and Hybrid Systems in UPPAAL2k, ("http://www.cs.auc.dk/ paupet/talks/MOVEP2k.html/")

[1] http://www.ensieta.fr/dtn/automatique/glider/gliders.htm

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/

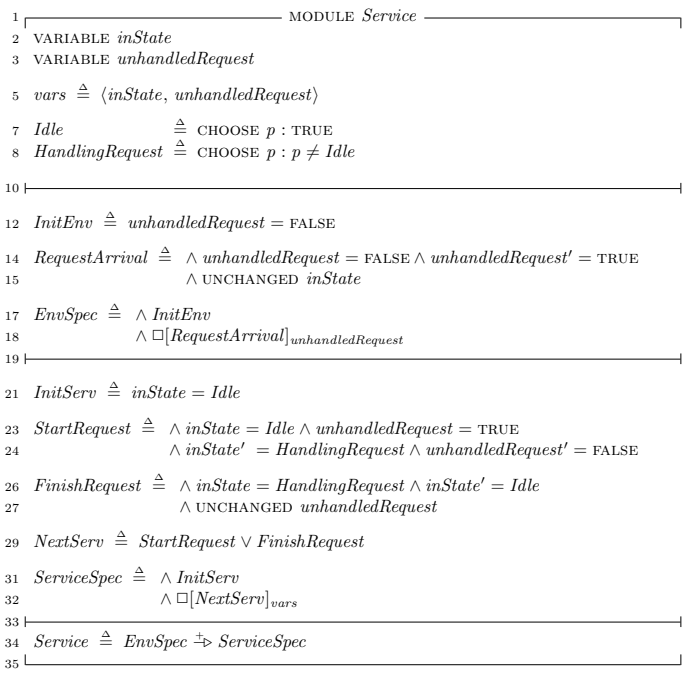


Fig. 1. Context model for a service offered by a system. The specifications have been abridged for space considerations.

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.

```
MODULE ResponseTimeConstrainedService
EXTENDS RealTime
VARIABLES LastResponseTime, inState, unhandledRequest, Start
Serv ≜ INSTANCE Service
Init ≜ Start = 0 ∧ LastResponseTime = 0
StartNext ≜ Serv!StartRequest ⇒ Start′ = now
RespNext ≜ Serv!FinishRequest ⇒ LastResponseTime′ = now − Start
Next ≜ StartNext ∧ RespNext
vars ≜ ⟨inState, unhandledRequest, Start, LastResponseTime⟩
RespSpec ≜ ∧ Init
           ∧ □[Next]_vars
Service ≜ ∧ Serv!Service
          ∧ RTnow(vars)
          ∧ RespSpec
```

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:

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

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:

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

$ComponentMaxExecTime$, $CPUCanSchedule$, $MinInterrequestTime$, 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:

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

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

A Model-driven Approach to Predictive Non Functional Analysis of Component-based Systems

Vincenzo Grassi Raffaela Mirandola
Università di Roma "Tor Vergata", Italy
{vgrassi, mirandola}@info.uniroma2.it

Abstract. We present an approach to the predictive analysis of non functional properties of component-based software systems. According to a model-driven perspective, the construction of a model that supports some specific analysis methodology is seen as the result of a sequence of refinement steps, where earlier steps can be generally shared among different analysis methodologies. We focus in particular on a path leading to the construction of a stochastic model for the compositional performance analysis, but we also outline some relationships with different refinement paths.

1. Introduction

In this paper we present a model-driven development (MDD) [2] approach to the predictive analysis of non functional properties of component-based software systems. Given our goal of supporting predictive analysis, our focus is on the construction as a "final product" of a suitable system model that lends itself to the application of some analysis methodology.

According to MDD, we look at the definition of this model as a sequence of *refinement steps*, where each step basically specializes and enriches a more "abstract" model defined at the previous step, in a suitable way for the needs and goals of some specific analysis domain. In our opinion, this approach serves multiple purposes. First, it allows to better isolate and understand the basic concepts that must be modeled, and their interdependencies. Moreover, at each step different refinements can be, in general, devised, where each of them can be seen as the definition of a different and more specialized view of the same system. As a consequence, the construction of a final model (view) as a sequence of refinements allows to possibly share some preliminary refinement steps with other final views. Finally, since different views are generally not independent, but some kind of relationship exists among them, this approach can also provide some support to check possible inconsistencies among different views. Figure 1 shows our vision of possible refinement steps, that start from a "root model" expressing basic concepts. We remark that Fig. 1 shows a reasonable sequence of refinements steps that does not intend to be normative.

In the rest of this paper, we illustrate in Sect. 2 the basic concepts expressed in the root model. Then, we give in Sect. 3 some details about possible refinement paths, focusing in particular on the path leading to the definition of a model suitable for stochastic performance analysis (surrounded by a dashed line in Fig. 1); however, to better illustrate our approach, we will also discuss some concepts that belong to different refinements paths. In Sect. 4 we present an example of "instantiation" of our approach, to better illustrate its general concepts through their application to the incremental construction of a model for the performance analysis of a simple component-based application. Finally, we discuss future work in Sect. 5.

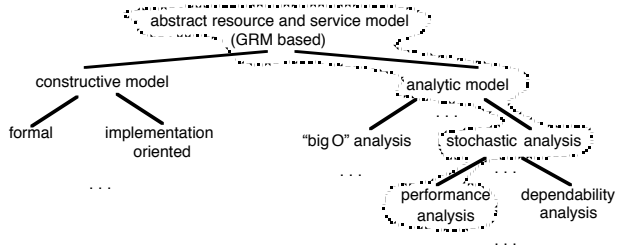


Figure 1. A possible sequence of refinement steps

2 Basic resource and service model

At the root of the model construction process we have an abstract resource and service model, depicted in Fig. 2. By *resource* we mean any run-time entity offering some service, thus encompassing both software components, and physical resources like processors, communication links or other devices (like printers and sensors). As a consequence, a *service* can correspond to some "high level" complex task, or some "low level" task such as the processing service offered by a processor. We think that at this stage it is unnecessary to introduce different models for these entities, since this would obscure the numerous characteristics they share. Distinct models can be possibly introduced at some next refinement step. This abstract model is derived, with some adaptation, from the General Resource Modeling (GRM) framework defined in the standard "UML Profile for Schedulability, Performance and Time Specification", so we refer to [1] for more details about some of the elements shown in Fig. 2. However, we would like to remark that, even if the model depicted in Fig. 2 adopts a UML-like notation, this is only made for notational convenience, but it does not mean that we are adopting UML as modeling language.

As depicted in Fig. 2, besides the basic concepts of resource and service (and some notion of attribute and parameter that can characterize them, not shown in the figure, except for QoS attributes), other concepts are introduced at this root stage. They include the distinction between *simple* services,[1] that do not require any external service to carry out their task, and *composite* services, that instead do require them. In the latter case, different resource usages can be specified, corresponding for example to different quality levels of the provided service.

Moreover, for the composite services offered by some component we also introduce the concept of a *component time* service model, where the required services are specified through a set of constraints that characterize them, and of an *assembly time* service model where the service is actually linked to service instances that satisfy those constraints.

Finally, this root model includes the basic notion of a dynamic service usage model, whose role is to specify some pattern of use of the required services. A usage pattern includes the specification of action executions, where an action could be a specific instance of an invocation of some required service, characterized, possibly, by its own actual parameters.

[1] An example of simple service is the processing service offered by a processor resource, but it could also be a more sophisticated service offered by some "black box" resource.

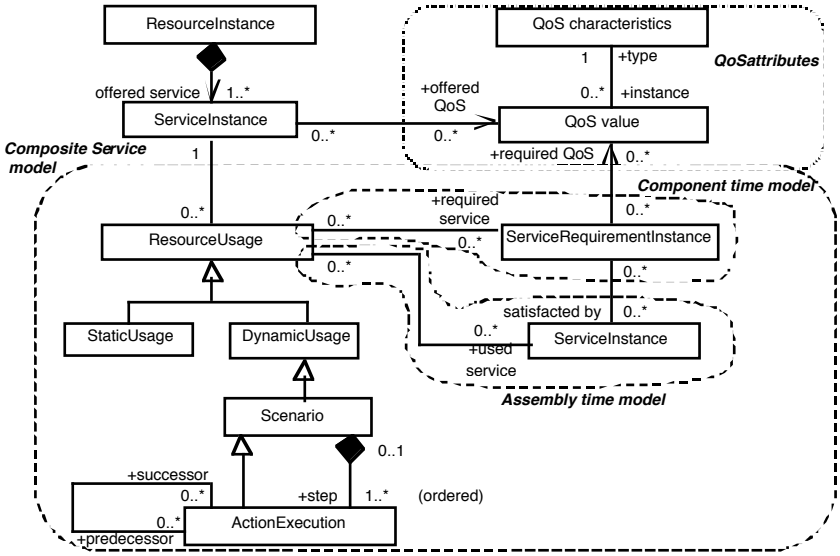


Figure 2. The root model

3 Refinement steps

3.1 Constructive vs. analytic refinement

The model just described is an abstract conceptual framework that outlines the fundamental elements needed to model a system built as a composition of services and resources offering them. Following [7], we distinguish at least two possible ways (as depicted in Fig. 1) of defining a first refinement of this conceptual model, each suitable for different modeling domain and goals: a *constructive* refinement, that spans aspects related to the actual construction of an assembly of services and resources, and an *analytic* refinement, that spans aspects related to the use of some analysis methodology to support statements and predictions about quantitative and qualitative properties of the modeled system.

Elements of the conceptual framework that need to be specialized according to these two different refinements include the *service*, *scenario* and *action execution* concepts. For these elements, we give examples of some information that should be included in their constructive and analytic refinements:

- constructive refinement: note that a further choice along this path concerns the selection of a suitable model to express it; we can adopt a formal model (like CSP), or an implementation oriented model (e.g. C-like); following for example this latter path we have:
  - *service*: specification of a "constructive" interface (e.g. the service signature: name and data type of the formal parameters);
  - *scenario*: specification of pattern of "activities", expressed using C-like control constructs (conditional statements, loops);
  - *action execution*: specification of values of the actual parameters for external required services invocation;
- analytic refinement: also in this case a further choice must be made, concerning the selection of a suitable analysis setting (e.g.: stochastic, "big O", ...); adopting for example a stochastic setting we have:
  - *service*: specification of an "analytic" interface (e.g. name and set of values of the

formal parameters);
- *scenario*: specification of a pattern of "activities" expressed using some stochastic model (e.g. probabilistic execution graph, stochastic Petri net);
- *action execution*: specification of random variables modeling the values of the actual parameters of a service invocation (these random variables must take values in the set of values for the corresponding formal parameter).

As outlined in Sect. 1, some relationship generally exists between alternative refinements we can follow at each stage. In the case of the constructive and analytic refinements of the conceptual model, it consists of an "abstraction mapping" from the constructive to the analytic model. Indeed, since the goal of the analytic refinement is to support some kind of predictive analysis, it must reasonably give rise to a more "abstract" model with respect to its constructive counterpart. In the following, we suggest a possible mapping for the elements described above, that can be used to check the existence of possible inconsistencies between the different system views corresponding to these two refinements, if both of them are available:
- *service*: the mapping from the domain of values for the data type of the "constructive" formal parameter to the set of values of the corresponding "analytic" formal parameter can be obtained by partitioning the original domain into a (possibly finite) set of disjoint subdomains, and then collapsing all the elements in each subdomain into a single representative element [3];
- *scenario*: the mapping from a constructive (e.g. C-like) to a stochastic specification of a pattern of activities is obtained by mapping, for example, conditional statements to probabilistic selections of alternative paths, or conditional loops to iterations whose number is controlled by a random variable;
- *action execution*: the mapping from the constructive to the analytic actual parameters (with the latter modeled by random variables) is obtained by guaranteeing that the probability distribution of the adopted random variables be representative of the actual distribution of values in the constructive parameters.

Note that in the case of the analytic stochastic refinement, a further refinement concerns how to specify the random variables introduced in the model: they could be specified by actually providing their probability distribution, or, at the other extreme, just their mean value, with an obvious trade off between degree of precision and analysis complexity.

3.2 Stochastic model refinement for the performance domain

In Sect. 3.1 we have presented some elements for the construction of a stochastic model of a component-based application. In our opinion, these basic elements play a relevant role in any stochastic model of a component-based system, independently of the particular quality category we are interested in (e.g. performance, dependability). However, once we select a particular category, further elements must be added to specialize the stochastic model.

Let us focus on performance analysis. In this case we are interested in the timeliness aspects of a system. Hence, a basic information that must be included in the "provided QoS" attributes of a resource concerns the time taken to carry out a single request for some service it offers. Let $T_{exec}(i)$ denote this time for an offered service Si. In a stochastic setting, $T_{exec}(i)$ is specified by a random variable (e.g. by its distribution, or mean value) that, in general, is parametric with respect to the service input parameters, as it will be shown in Sect. 4. However, besides the service

parameters, two other factors must be taken into account to completely specify $T_{exec}(i)$:
- whether the service is a *simple* or *composite* service;
- whether the service is a *no contention* or *contention-based* service.

The former distinction has been already discussed within the basic conceptual model of Sect. 2. The latter distinction concerns services that are always able to serve a request with no interference with other concurrent requests, with respect to services where multiple concurrent requests can interfere with each other, thus requiring the specification of some scheduling and/or access control policy. As a consequence, $T_{exec}(i)$ as observed by someone requiring Si consists, in general, of a part $T_{int}(i)$ corresponding to the time spent in internal actions that do not require any external service, a part $T_{cont}(i)$ that takes into account the time spent waiting before actually accessing the service, and a part $T_{ext}(i)$ corresponding to the time to carry out externally required services; that is:

$$T_{exec}(i) = T_{int}(i) + T_{cont}(i) + T_{ext}(i) \quad (1)$$

$$\text{with: } T_{ext}(i) = \bigoplus_{Sj \in Required(Si)} T_{exec}(j) \quad (2)$$

where $\oplus$ denotes some "composition" operation. Hence, $T_{exec}(i)$ can be completely specified at component time only if Si is a simple and no contention service (i.e. $T_{cont}(i) = T_{ext}(i) = 0$). In all the other cases, we can only completely specify the $T_{int}(i)$ part of $T_{exec}(i)$; for what concerns $T_{ext}(i)$ we can instead only specify the amount of service demand addressed to external services; finally, for what concerns $T_{cont}(i)$ we can at most specify it as some function of the demand addressed to Si (depending on the adopted scheduling and access policy), which is unknown at component time.

To give a more complete characterization of $T_{exec}(i)$ in the case of composite or contention-based services we must wait the construction of an assembly time model. In the following we briefly outline two possible approaches to the evaluation of $T_{exec}(i)$, once an assembly time model has been built:[2]
- *contention unaware*: we assume $T_{cont}(i) = 0$ for all services, that corresponds to assuming that all services are no contention services;
- *contention aware*: we assume $T_{cont}(i) \geq 0$, thus taking into account the impact of contention.

With the former approach, we can build a model for the calculation of $T_{exec}(i)$ using only information associated to the dynamic resource usage of each assembled service Si, neglecting any contention or access control issue. For example, if the dynamic usage is specified through a probabilistic execution graph, we can suitably "connect" the execution graphs of the assembled services (according to the mapping between required and offered services) and then we can use graph analysis techniques [6] to calculate the overall completion time.

The value of $T_{exec}(i)$ calculated according to the contention unaware approach can be considered as a lower bound for the (more realistic) value calculated with the latter, contention aware approach. However, it can provide valuable insights (e.g. to perform an first service selection), at a lower modeling and computational effort. If the impact of contention must be taken into account, then we must build a more complex model.

---

[2] These approaches basically correspond to constructive characterizations of the $\oplus$ operation.

Queueing network models appear natural candidates for this purpose, as they are specifically addressed to model resource contention, but other kind of models could be used as well (and the selection of a specific stochastic model is a further refinement step). In the case of queueing networks, the execution graphs associated to each service can be used to build the workload for the queueing network servers. However, we would like to remark that our general model of resource implies that several "layers" of workloads and servers should be taken into consideration, with each layer generating workload for the lower layers. As a consequence, the traditional "flat" queueing network models (like the EQN models of [6]) could result insufficient, and more complex modeling and solution techniques could be necessary [4, 5]. For space limits we do not discuss in more detail this point.

4 Example

We use a very simple example to illustrate the above ideas. For this purpose, we consider a resource that offers a search service for an item in a list; to carry out this service, it requires a sort service (to possibly sort the list before performing the search) and a processing service (for its internal operations). In turn, the sort service requires a processing service. Let us describe how we can apply the proposed approach to this example.

Basic GRM-based model. As already stated, at this level it is necessary to identify the resources involved in the application and the kind of offered and required services with their basic characterization. Table 1 shows an example of such an abstract model.

Table 1. The root model

Resource	*offered service*	*service type*	*required services*
Search_res	search(list, item)	composite	process, sort
Sort_res	sort(list)	composite	process
CPU_res	process(op_list)	simple	*none*

Constructive refinement. The constructive characterization of the composite sort and search services could include the definition of a formal parameter `l:list of T`, whose domain is the set of all the lists with elements of type `T`, and the definition, using some pseudo-code, of a pattern of requests addressed to other services, with the actual parameters of service invocations (in this example, `sort_algorithm` and `search_algorithm` denote a list of operations implementing some sort and search algorithm, respectively, passed as "actual parameter" to a processing service):

```
Sort_res.sort(l:list of T) =
      {call(process(sort_algorithm(l)))};[3]
Search_res.search (l:list of T, i:T) =
      {if (not_ordered(l)) call(sort(l));
          call(process(search_algorithm(l)));
       }
```

On the other hand, the processing service characterization does not contain any request addressed to other services, since it is a simple service:

[3] In this pseudo-code, `call(s(p))` denotes the request for some service `s` with actual parameter `p`.

```
CPU_res.process(oplist:list of MachineOperation) =
      {do(oplist)}
```

Analytic refinement: stochastic approach. In the analytic characterization of the sort and search services, the list formal parameter could be defined as `l:integer`, with domain given by the set of non negative integers, each representing the size of some list (under the assumption that the list size is the only information we need within some analysis methodology). Analogously, the analytic characterization of the process service can be given in terms of a service offered by an entity executing a single kind of "average" operation (at some constant speed), with a formal parameter defined as `oplist:integer` that specifies the number of such operations, rather than their actual sequence.

For what concerns the pattern of activities of the sort and search composite services, it can be given in terms of a stochastic model, expressed through the probabilistic execution graphs of Figs. 3 and 4. In these graphs, the actual parameters of the service requests associated to each node are random variables. To take into account the dependency between the demand addressed to other services and the demand addressed to the service itself, these random variables are parametric with respect to each service formal parameters (the list parameter in this example). We think that introducing this kind of parametric actual parameters is a fundamental requirement to support compositional analysis. For example, in Fig. 3, assuming a quicksort algorithm, and recalling that the formal parameter `l` actually corresponds to the list size, the actual parameter for the process request can be modeled as an integer valued random variable in the range $[k1 \cdot l \cdot \log(l),\ k1 \cdot l^2]$, where `k1` is some suitable proportionality constant. In Figs. 3 and 4 we assume that the involved random variables are specified through their mean value only (where `k2` and `k3` are other proportionality constants).

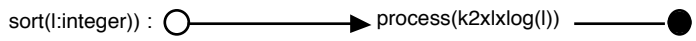


Figure 3. Sort service execution graph

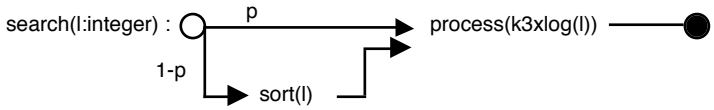


Figure 4. Search service execution graph

For what concerns the consistency between this analytic model and the constructive model note that, for example, according to what discussed in Sect. 3.1, the definition of the analytic list formal parameter can also be seen as the result of a partition of the corresponding constructive list domain into subdomains each containing all the lists with the same size, with each subdomain collapsed to an integer value corresponding to this size. As another example, the probabilistic execution graph in Fig. 4 is consistent with the search service constructive pattern, provided that the `p` probability corresponds to the average number of times the `not_ordered(l)` condition is false.

Contention unaware analysis. For this kind of analysis we assume the $T_{cont} = 0$ for each service execution time. In our example, this analysis corresponds to assuming that two different CPU resources are exploited by the search and sort services,

respectively, and that no other service request arrives at these CPUs and at the sort service, besides those generated by the entities considered in the example.

Since the service demand in the execution graphs of the analytic model is specified through the average value of random variables, we can calculate the average execution time for each service by performing simple summations. Using (1) and (2) we get:

$$T_{exec}(\texttt{process(oplist)}) = T_{int}(\texttt{process(oplist)}) = \texttt{oplist/cpu\_speed}$$
$$T_{exec}(\texttt{sort(l)}) = T_{ext}(\texttt{sort(l)}) = T_{exec}(\texttt{process(k2}\cdot\texttt{l}\cdot\texttt{log(l))})$$
$$T_{exec}(\texttt{search(l)}) = T_{ext}(\texttt{search(l)})$$
$$= T_{exec}(\texttt{process(k3}\cdot\texttt{log(l))) + (1-p)}T_{exec}(\texttt{sort(l)})$$

From which we can derive:

$$T_{exec}(\texttt{search(l)}) = \texttt{(1-p)k2}\cdot\texttt{l}\cdot\texttt{log(l)/cpu\_speed} + \texttt{k3}\cdot\texttt{log(l)/cpu\_speed}$$

Contention aware analysis (queueing network model). In this example we assume that the services for which contention may occur are the processing service offered by a CPU and, possibly, the sort service. In both cases we must define a suitable scheduling policy (e.g. First In First Out (FIFO)) and a workload model corresponding to some overall usage pattern for the considered service. Assuming that contention occurs only for the processing service, we can build a queueing network (QN) model, modeling the CPU as a service center with a FIFO scheduling policy, serving a workload that can be modeled by a suitable set of different job classes, originating from a terminal node (for a closed QN model with a fixed number of job in the network) or from a job source (for an open QN model with unlimited number of jobs). Figure 5 shows examples of these models. For these models, we have:

$$T_{exec}(\texttt{process(oplist)}) = T_{int}(\texttt{process(oplist)}) + T_{cont}(\texttt{process(oplist)})$$

where the T_{cont} component can be calculated using classical QN analysis techniques.

If we assume that contention can occur also to access the sort service, then we can use different modeling approaches that take into account also the competition for "software" resources, such as the "layered" QN (LQN) proposed in [5] or the "multilevel" QN proposed in [4]. Figure 6 illustrates an example of multilevel QN for our example. In the "software" QN model the competition for the sort service is represented by modeling the Sort_res resource as a queueing resource (e.g. with FIFO service discipline), while the impact of other resources (in our example the internal processing in the search service) is modeled by a delay center. On the other hand, the "hardware" QN models the shared CPU resource. The number of jobs in the two QNs is strictly related: a job in the software QN is also using or waiting to use the CPU in the hardware QN, and the number of jobs contending for the CPU is equal to the number of concurrent jobs that are not blocked waiting for the sort resource. Iterative techniques can be used to obtain the performance indices of interest [4].

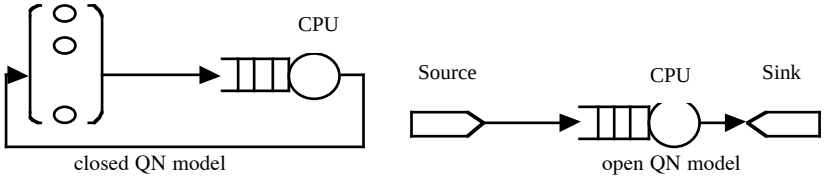


Figure 5. Contention aware models for the CPU resource

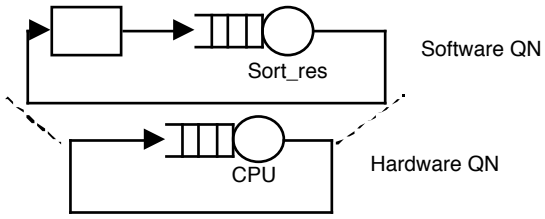


Figure 6. A multilevel contention aware model for the CPU and sort resources

5 Conclusions

We have described a path that leads to the construction of a stochastic model for the compositional performance analysis of component-based systems. For each step of this path, we have outlined basic information that should be provided, and we have given some suggestion about how to structure this information. Moreover, we have also discussed some relationships with different refinement paths. We are currently working toward the actual "implementation" of this path, where an important role is played by the definition of a suitable language to express the needed information, with a precisely defined syntax and semantics that support the development of automatic tools for QoS predictive analysis of component-based systems.

References

1. "UML Profile for Schedulability, Performance, and Time Specification", on line at: http://cgi.omg.org/docs/ptc/02-03-02.pdf.
2. J. Bettin "Model driven software development" *MDA Journal*, April 2004, pp. 1-13.
3. D. Hamlet, D. Mason, D. Woit "Properties of Software Systems Synthesized from Components", June 2003, on line at: http://www.cs.pdx.edu/~hamlet/lau.pdf (to appear as a book chapter).
4. D.A. Menascè "Simple analytic modeling of software contention" *Performance Evaluation Review*, vol. 29, no. 4, March 2002, pp. 24-30.
5. J. Rolia, K. Sevcik "The method of layers" *IEEE Trans. on Software Engineering*, vol. 21, no. 8, August 1995.
6. C. U. Smith, L. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, 2002.
7. K.C. Wallnau "Volume III: a technology for predictable assembly from certifiable components" *Tech. Rep. CMU/SEI-2003-TR-009*, Apr. 2003.

Tailor-Made Containers: Modeling Non-functional Middleware Service

Ronald Aigner, Christoph Pohl, Martin Pohlack, and Steffen Zschaler

Technische Universität Dresden
Dresden, Germany
{Ronald.Aigner, Christoph.Pohl, Martin.Pohlack,
Steffen.Zschaler}@inf.tu-dresden.de

Abstract. We propose to create tailor-made application servers by composing components providing support for individual non-functional properties. In this position paper we start from a static, asymmetric approach splitting the application server in two parts to support realtime properties and generalize this to a symmetric, componentized architecture. We then analyse how modelling of non-functional properties of systems is influenced by this application server architecture.

1 Introduction

In the COMQUAD[1] project we are developing a container that enables guarantees of non-functional properties of component-based software. In a recent paper [1] we reported on the container's architecture, which has been split into two parts—one which is generic, but cannot give realtime guarantees, and one which is specific to realtime—to enable us to efficiently support realtime properties, as well as confidentiality properties, while reusing large parts of code for component management (namely the open-source JBoss application server [2]). In this position paper we want to explore how this idea of splitting the middleware into two parts can be generalized to other properties, and how this affects the modeling of these properties. In particular, we want to examine how the container and the properties it provides for can be modeled in such a way as to allow a property-specific container to be assembled from various parts.

We propose to modularize containers as aspects that are woven together with the actual component *depending on the set of non-functional properties to be supported*. A container then consists of a core part, which is responsible for generic component management and provides the hooks to which the component itself, as well as the aspects implementing various non-functional properties, can be connected. In order for this to work automatically, either at runtime or at deployment time, both the aspects and their relation to the non-functional property(ies) they realize need to be modeled. In particular it is important to model the interdependencies between different properties or their implementing aspects, respectively.

[1] COMponents with QUantitative properties and Adaptation, a project funded by the German Research Council

The rest of this position paper is structured as follows: Section 2 gives a short overview of our split container architecture as presented in [1]. This architecture is then generalized in Sect. 3 into a componentized, symmetric architecture. The main section of the paper is Sect. 4, which gives a more detailed analysis of different approaches to modeling the relation between non-functional properties and the aspects implementing them. Finally, we briefly review some related work and summarize our position paper.

2 A Split Container Architecture

In the course of the COMQUAD project we built a component container with support for non-functional properties. This includes the ability to fulfill realtime requirements of components. Support for realtime includes the ability of the middleware—the container—and the underlying operating system to provide guarantees for timeliness of execution. It also includes that resources required by a component can be provided to this component when required within a predefined time span. Such guarantees can be given if the component—or some representative of the component—make a reservation with a manager of the resources. The resource manager will then guarantee the timely access to the resource.

Thus, it is essential to extend the container by a resource management providing services for admission and reservation of system resources, such as CPU time, network bandwidth, or memory. The key part of the container is the contract manager, which instantiates and connects the components required to service a client request. In order to select the correct components and component implementations, the contract manager uses the functional and non-functional (including realtime) component specifications.

Typically, applications that give guarantees on realtime properties are split into two parts: *(i)* a small part of code that performs the actions for which guarantees of realtime properties are essential and *(ii)* a usually much larger part for which realtime guarantees are not important. This distinction can be applied to the architecture of a component runtime environment. Most of the complex work is done in the non-realtime part of the component environment. It manages the available component implementations and application specifications, handles requests for creation of component networks, negotiates contracts between components, and maintains a model of the instantiated components and the networks formed by them. The realtime part is essentially restricted to actually instantiating components, reserving resources, and executing the instantiated components in response to user requests.

This concept is closer to the reality of applications with realtime properties than the approach of monolithic realtime applications. Additionally, it also allows us to make use of advanced component technology (namely JBoss [2]) on the non-realtime side while still being able to make full use of the realtime and resource reservation features of the underlying platform on the realtime side. Thus, we can leverage the best of both worlds and move towards a running prototype very efficiently.

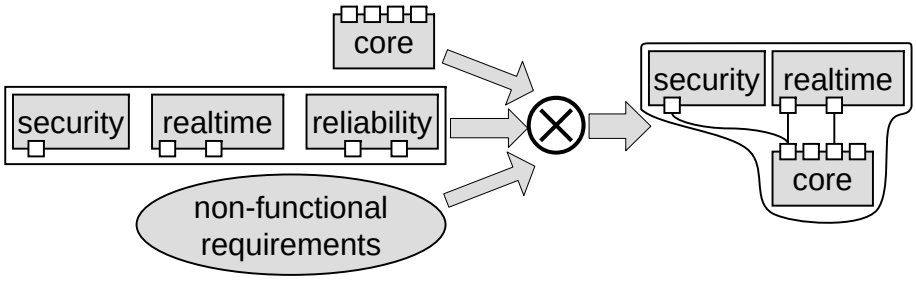


Fig. 1. Composition of a property-specific container from core and property-specific functionality

3 A Componentized Container Architecture

The split container architecture we introduced in Sect. 2 proved to be a sensible decision for the design of dependable component-oriented systems with support for realtime guarantees. Our current container architecture is now split into two parts: The non-realtime container is based on JBoss, running in a standard Java Virtual Machine (JVM) on L^4Linux and providing complex services, such as a component repository. The realtime container is running directly on the L4 microkernel. It contains all the realtime-capable components and the necessary infrastructure.

This motivated us to investigate if other non-functional properties can be handled in a similar way. Consequently, we tried to find similar "splitting lines" in the corresponding middleware support code for those properties and examined possible correlations to our previously introduced architecture split for realtime–non-realtime system parts.

The idea was that, provided we could show these splits to be "congruent"[2], we would then be able to generalize our split architecture for use with other properties. However, these splits are not congruent in general; they rather relate to each other in an arbitrary fashion. We therefore propose a more symmetric architecture, which has been depicted in Fig. 1. This architecture provides a set of *core services* such as instantiation, component connection, and so on, and a set of *property dependent services*. We propose a container generation operator $\otimes$ that creates custom-made containers by selecting the appropriate property dependent services and weaving them together. This container generation is performed either dynamically at runtime or statically at deployment time.

Container generation can only be automated if the non-functional properties under consideration as well as the components providing support for them have been appropriately modeled. This will be the subject of the following section.

[2] congruent in terms of mutual non-overlapping parts: a lean, mission-critical part and a non-critical management part, which are identical for different non-functional properties

4 Modeling Aspects

In this section we are going to evaluate various approaches to modeling property dependent services so that they can be automatically selected and merged. We start out by looking at the JMX [3] architecture as a conceptual model and explain why this is not sufficient for our purposes. We then show how aspect-oriented approaches can be applied in our case, but also what the related drawbacks are.

The idea arose to model specific non-functional aspects using a plug-in–extension model such as JMX in JBoss [2]. To illustrate this approach and its limitations we will outline some steps required to provide fuzzy time. One important aspect of security is confidentiality—that is, information must not become available to unauthorized subjects. Covert channels are one way through which information could be transfered secretly and must therefore be considered if confidentiality is one required non-functional property. Covert channels can be classified into storage and timing channels and have been a research subject for many years [4].

In practice covert timing channels cannot be prevented completely. Instead it is common to limit their bandwidth by impeding unprivileged subjects' access to accurate time sources. For instance, a system could delay the execution of each system call for a random amount of time. In the literature this is called imprecise clocks or fuzzy time.

There is an obvious conflict between realtime requirements and the imprecision of time, as timeliness can not be assured for specific services at the same time as the access to accurate timers is restricted. Investigations about the impact of such solutions on realtime systems have already been undertaken years ago. Son et al. [5] describe a very domain specific solution for a realtime database with security requirements. The authors propose to temporarily weaken the security requirements to fulfill realtime requirements, thereby providing a tradeoff between realtime performance and security.

Enforcement of fuzzy time requires pervasive modifications of several underlying services, possibly including program code inspection. All access to time sources must be encapsulated. Possibly the simplest thing to do would be to add a random delay to all services providing timing information, for example a system call to operating system. However, one also needs to prevent access to other time sources, such as NTP servers, which are reachable via network access. As one cannot filter all network traffic for all time sources (network traffic may be encrypted or steganography might be used to hide the data transported), network access must also be delayed for a random amount of time. Unfortunately, many services might be used as time source. Consequently, to provide fuzzy time many components must be considered. This is more than can be modeled with the concepts provided by JMX, which allow only simple non-invasive extensions to the container's functionality.

Invasive modifications are the domain of aspect-orientation [6, 7]. We will therefore analyze how we can use aspects to model property dependent services. Looking more closely at the examples from our experience, we find that we need to extend the standard notion of aspects in at least three ways:

1. Aspects providing for non-functional properties require join points on two levels:

(a) A runtime level, where they need to be able to intercept method calls, events, or data packets on a stream connection. These join points can be and have been conveniently implemented using interceptors.
(b) A meta-level, where they need to be able to intercept state changes in a component's life cycle. These join points cannot be implemented by standard interceptor technologies, but require special reflective support at components' meta-level [8].

These special issues of dynamic insertion and configuration of non-functional properties have also been realized by the developers of OpenORB/OpenCOM [9], Lasagne [10], and OIF [11], among others.

2. Aspects come with pre-conditions. For example, an aspect may require certain other aspects to be present or absent, or certain operations to be available, in order to provide the non-functional property it supports. These pre-conditions are semantic in nature; that is, they do not only require certain names to be available, but certain structures resp. behaviors. This can potentially be combined with techniques from the Model Driven Architecture (MDA, [12]) using UML extension mechanisms for semantic markup and code generation.
3. Aspects need to be parameterized. For instance, because an aspect providing response time properties will be used for different specific response times, there must be a way to pass this in as a parameter, which may already have to be considered at the time of weaving.

An open research issue remains the interference of apparently orthogonal aspects. There are no general solutions to model such influences. The approach of simply using constraint UML dependencies and OCL expressions is clearly limited to effects that are known in advance. However, unanticipated interactions and side-effects are much harder to capture during the software development life cycle. This effect is actually a subset of a problem area called "feature interaction" [13], which has been discussed in the telecommunications community since the early 1990s. An example for this is the treatment of security by means of cryptography, which we introduced in [14]. The container infrastructure has to handle the tradeoff between performance (in terms of timeliness) and security (in terms of confidentiality and integrity). We believe that there are many of these conflicting non-functional requirements and presented these as a prominent and apparent example. We also believe that there are many domain-specific models to cope with the conflicting goals, which should be integrated into a common modeling framework.

5 Related Work

The concepts of Model Driven Architecture (MDA) and Model Driven Middleware [15] include the application of models to middleware configuration and implementation. These include the integration with QoS aware middleware and application. The described tool chain and models also mention the problem of interfering requirements, but do not provide a solution beyond their detection. Nonetheless does MDA provide means to describe QoS requirements on different levels of abstraction.

The PURE operating system family [16] exploits the concept of aspect oriented programming to build tailored operating systems. Similar to our concept of generated containers for specific non-functional properties PURE uses AspectC++ [17] to generate operating systems adapted to the demands of the application running on top of it and the available hardware. Aspects are used to describe and implement the features of the operating system, such as scheduling strategies. As described in this paper, aspects cannot be applied to interfering non-functional properties, such as security and realtime.

Requirements for realtime extensions for Java were defined in the NIST report [18]. The NIST group proposes partitioning the execution environment into a realtime core providing the basic realtime functionality and a traditional JVM, which services normal Java applications. Based on these requirements, the J Consortium defined the Real-Time Core Extensions for Java (RTCE) [19], which follow the idea of a separate core for realtime services. In contrast, in the Real-Time Specification for Java (RTSJ) [20] all services are provided in one JVM, as such containing the realtime and the non-realtime applications. The architectural RTCE approach is similar to the design of our system, in that both run large and complex parts in a classic non-realtime environment and only small, predictable parts in a realtime environment.

Dassault Systèmes introduced a general architecture for software components with support for arbitrary non-functional properties [21]. Their concept for weaving non-functional aspects is very similar to our proposition in that they suggest to generate specialized containers comprising the specific services that are actually needed according to the non-functional requirements of the application. New services can be defined using an aspect definition language; they can be put at use by means of an aspect user language. However, there is no concept so far for supporting composition of aspects. Feature interaction and interference of orthogonal aspects also remain open issues. Also, the non-functional properties shown in [21] are those which can already be handled by modern application servers. It is not clear from the paper whether the authors would be able to model a realtime aspect such as frame rate of a video player.

6 Conclusions

In this position paper we presented the shift from our split container architecture with support for realtime properties towards a more general, componentized approach to dynamically weaving customized runtime environments with support for arbitrary non-functional aspects for component-based software applications. We conclude that the generalized version of the container architecture is a promising step forward, that needs further work and which we would like to suggest as a field for future research. The most interesting research issues remain in the field of modeling the different services required for supporting non-functional aspects. We have shown that even the current state of Aspect-Oriented Modeling still has a number of drawbacks, especially in terms of capturing interferences of aspects, which require more work in that direction.

In [22] and [23] we presented a formal approach to the specification of non-functional properties of component-based systems. The work started in this position paper should enable us to extend the work reported on in those papers to support independent specification of multiple non-functional properties of the same system.

References

1. Göbel, S., Pohl, C., Aigner, R., Pohlack, M., Röttger, S., Zschaler, S.: The COMQUAD component container architecture. In Magee, J., Szyperski, C., Bosch, J., eds.: 4th Working IEEE/IFIP Conf. on Software Architecture (WICSA), Oslo, Norway, IEEE (2004) 315–318
2. Fleury, M., Reverbel, F.: The JBoss extensible server. In Endler, M., Schmidt, D., eds.: International Middleware Conference. Volume 2672 of LNCS., Rio de Janeiro, Brazil, ACM / IFIP / USENIX, Springer (2003)
3. Sun Microsystems: Java Management Extensions (JMX) Instrumentation and Agent Specification. v1.2 edn. (2002) http://java.sun.com/products/JavaManagement/.
4. Millen, J.: 20 Years of Covert Channel Modeling and Analysis. Technical report, Computer Science Labroratory, SRI International, Menlo Park, CA (1999)
5. Son, S.H., Mukkamala, R., David, R.: Integrating security and real-time requirements using covert channel capacity. IEEE Transactions on Knowledge and Data Engineering **12** (2000) 865–879
6. Elrad, T., Aldawud, O., Bader, A.: Aspect oriented modeling—bridging the gap between design and implementation. In Batory, D., Consel, C., Taha, W., eds.: Generative Programming and Component Engineering (GPCE 2002). Volume 2487 of LNCS., Pittsburgh, PA, USA, ACM SIGPLAN/SIGSOFT, Springer (2002) 189–201
7. Aldawud, O., Elrad, T., Bader, A.: UML profile for aspect-oriented software development. In Aldawud, O., ed.: 3rd International Workshop on Aspect-Oriented Modeling with UML, Boston, USA, ACM SIGSOFT / SIGPLAN (2003) In conjunction with the International Conference on Aspect-Oriented Software Development (AOSD'03).
8. Kiczales, G., Rivieres, J.D., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)
9. Blair, G.S., Coulson, G., Robin, P., Papathomas, M.: An architecture for next generation middleware. In Davies, N., Raymond, K., Seitz, J., eds.: Middleware 1998, The Lake District, England, IFIP, Springer (1998)
10. Jørgensen, N., Truyen, E., Matthijs, F., Joosen, W.: Customization of object request brokers by application specific policies. In: Distributed Systems Platforms. Volume 1795 of LNCS., New York, IFIP/ACM, Springer (2000) 144–163
11. Filman, R.E., Barrett, S., Lee, D.D., Linden, T.: Inserting ilities by controlling communications. Communications of the ACM **45** (2002) 116–122
12. Miller, J., Mukerji, J.: MDA Guide. The Object Management Group. Version 1.0.1 edn. (2003) OMG document number omg/2003-06-01.
13. Pulvermüller, E., Speck, A., D'Hondt, M., DeMeuter, W., Coplien, J.O.: Feature interaction in composed systems, ECOOP 2001 Workshop Proceedings. Technical Report 2001-14, Universität Karlsruhe (2001)
14. Franz, E., Pohl, C.: Towards unified treatment of security and other non-functional properties. In: Workshop on AOSD Technology for Application-Level Security (AOSDSEC'04), Lancaster, UK (2004)
15. Gokhale, A., Schmidt, D., Natarajan, B., Gray, J., Wang, N. In: Model Driven Middleware. John Wiley & Sons, Ltd. (2004)
16. Beuche, D., Guerrouat, A., Papajewski, H., Schröder-Preikschat, W., Spinczyk, O., Spinczyk, U.: On the development of object-oriented operating systems for deeply embedded systems - the PURE project. In: Proc. 2nd ECOOP Workshop on Object-Orientation and Operating Systems (Lisbon, Portugal). (1999) 27–31
17. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: Aspectc++: An aspect-oriented extension to c++. In: Proc. 40th Int'l Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia (2002)

18. National Institute of Standards and Technology: Requirements for Real-time Extensions for the Java Platform. (1999) Available at http://www.nist.gov/rt-java/.
19. J Consortium: Real-Time Core Extensions (RTCE). (2000) Available at http://www.j-consortium.org/.
20. The Real-Time for Java Expert Group: The Real-Time Specification for Java. v1.0 edn. (2001) http://www.rtj.org/.
21. Duclos, F., Estublier, J., Morat, P.: Describing and using non functional aspects in component based applications. In: Proc. 1st Int'l Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, ACM, ACM Press New York, NY, USA (2002) 65–75
22. Zschaler, S.: Towards a semantic framework for non-functional specifications of component-based systems. In: Proc. EUROMICRO conference 2004, track on Component-based Software Engineering, Rennes, France (2004) To appear.
23. Zschaler, S.: Formal specification of non-functional properties of component-based software. In: Proc. Workshop on Models for Non-functional Aspects of Component-Based Systems. (2004) Submitted for Publication.