FESCA 2011

8TH INTERNATIONAL WORKSHOP ON
FORMAL ENGINEERING APPROACHES TO SOFTWARE
COMPONENTS AND ARCHITECTURES

BARBORA BÜHNOVÁ AND JENS HAPPE (EDS.)

http://www.etaps.org
http://etaps2011.cs.uni-saarland.de
Electronic Notes in Theoretical Computer Science

Pre-Proceedings

FORMAL ENGINEERING APPROACHES TO SOFTWARE COMPONENTS AND ARCHITECTURES

FESCA’11

Saarbrücken, Germany

2nd April 2011

Guest Editors:

BARBARA BUHNOVA AND JENS HAPPE
# Contents

**Preface**

<table>
<thead>
<tr>
<th>Abstracts</th>
<th>1</th>
</tr>
</thead>
</table>

**Rolf Hennicker**

| Interface Coherence of Reactive Software Components: Solutions and Challenges | 3 |

**Michael Hauck**

| Model-driven Performance Engineering with the Palladio Component Model | 5 |

**Regular Papers**

| 7 |

**Premek Brada**

| Enhanced Type-based Component Compatibility Using Deployment Context Information | 9 |

**Arnaud Lanoix, Julien Dormoy and Olga Kouchnarenko**

| Combining Proof and Model-checking to Validate Reconfigurable Architectures | 25 |

**Galina Besova, Heike Wehrheim and Annika Wagner**

| Reputation-based reliability prediction of service compositions | 41 |

**Jaroslav Šnajberk and Premek Brada**

| ENT: A Generic Meta-Model for the Description of Component-Based Applications | 57 |

**Position Papers**

| 73 |

**Erik Burger and Ralf Reussner**

| Performance Certification of Software Components | 75 |
Preface

This volume contains the proceedings of the 8th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA). The workshop was held on 2nd April 2011 in Saarbrücken (Germany) as a satellite event to the European Joint Conference on Theory and Practice of Software (ETAPS’11).

The aim of the FESCA workshop is to bring together both young and senior researchers from formal methods, software engineering, and industry interested in the development and application of formal modelling approaches as well as associated analysis and reasoning techniques with practical benefits for component-based software engineering.

Component-based software design has received considerable attention in industry and academia in the past decade. In recent years, the growing need for trustworthy software systems and the increased relevance of systems reliability, performance, and scalability have stimulated the emergence of formal techniques and architecture modelling approaches for the specification and implementation of component-based software architectures. Both have to deal with an increasing complexity in software systems challenging analytical methods as well as modelling techniques.

FESCA aims to address the open question of how formal methods can be applied effectively to these new contexts and challenges. FESCA is interested in both the development and application of formal methods in component-based development and tries to cross-fertilize their research and application.

The previous FESCA workshops at ETAPS 2004 to 2010 enjoyed high-quality submissions and attracted a number of recognized guest speakers, including Constance L. Heitmeyer (Naval Research Laboratory, USA), Manfred Broy, (Technische Universität Muenchen, Germany), Jose Luiz Fiadeiro, (University of Leicester, UK), Frantisek Plasil (Charles University, Czech Republic), Martin Wirsing (LMU, Germany), and Ivana Cerna (Masaryk University, Czech Republic).

The program committee of FESCA’11 consisted of

- Jeremy Bradley (Imperial College London, UK)
- Ivana Cerna (Masaryk University, Czech Republic)
- Martin Fraenzle (University of Oldenburg, Germany)
- Lars Grunske (Swinburne University of Technology, Australia)
- Ludovic Henrio (INRIA Sophia Antipolis, France)
- Holger Hermanns (Universität des Saarlandes, Germany)
- Jan Kofron (Charles University in Prague, Czech Republic)
- Samuel Kounev (University of Karlsruhe, Germany)
The papers were refereed by the program committee and by several outside referees, whose help is gratefully acknowledged.

ENTCS is published electronically through the facilities of Elsevier Science B.V. and under its auspices. The volumes in the ENTCS series are available online at http://www.elsevier.nl/locate/entcs. We are grateful to ENTCS for their continuing support.

For the eighth time, FESCA has been organized as a satellite event to ETAPS. We are very grateful to the ETAPS organizers, especially to Bernd Finkbeiner, for taking care of all the local organization and for accommodating all our special requests.

Karlsruhe, February 07, 2011 Barbora Buhnova and Jens Happe
Abstracts

Invited Talk & Tutorial
Interface Coherence of Reactive Software Components: Solutions and Challenges

Rolf Hennicker

Ludwig-Maximilians-Universität München
hennicke@psl.iif.lmu.de

Summary

Interface coherence is a key issue in component-based software development. It concerns two dimensions of the development process: component implementations should adhere to their interface specifications - and - components should be composed only if their interfaces are compatible. To achieve independent implementability it is essential that both dimensions work properly together, i.e. that compatibility is preserved by refinement and refinement is preserved by composition, which are crucial requirements for any kind of interface theory. This talk analyses solutions and challenges derived from various interface theories based on state transition specifications with input/output-actions (e.g. interface automata, modal I/O-transition systems and open Petri nets). Criteria for the analysis shall be the communication style (synchronous, asynchronous), expressiveness (observational abstraction, treatment of data), decidability of refinement and/or compatibility, as well as semantic soundness and completeness.
Model-driven Performance Engineering with the Palladio Component Model

Michael Hauck

FZI Research Center for Information Technology, Karlsruhe, Germany
hauck@fzi.de

Summary

The Palladio Component Model (PCM) has been developed over the last 6 years. Today it is a mature modelling language for modelling component-based or service-oriented software systems with a special focus on predicting extra-functional properties of the system based on its constituting components. The PCM highly relies on model-driven software development techniques for this and uses automated transformations into well-known prediction models or simulation systems. It is supported by a mature, industry proven tool set based on the Eclipse platform. The tutorial consists of two parts. The first part presents the PCM’s foundational ideas from the area of component-based or service-oriented software development, its analysis capabilities, and possible extension points. The tutorial presents components in different phases of their life-cycle, and discusses the PCM’s understanding of a typical component-based software development process and the developer roles involved into it. Then it focuses on performance prediction and the annotations necessary for this. The second part of the tutorial introduces the PCM’s tool set and shows how to use it to create and analyse PCM models.
Regular Papers
Enhanced Type-based Component Compatibility Using Deployment Context Information

Premek Brada
Department of Computer Science and Engineering
University of West Bohemia, Pilsen, Czech Republic
brada@kiv.zcu.cz

Abstract
Consistency and compatibility in component-based applications have been the subject of many methods and approaches, from formally sound ones with difficult practical implementation to pragmatic rules for comparing version meta-data which offer only weak guarantees. This is especially true of many industrial component frameworks in routine use. In this paper we contribute a formal description of a method which ensures application run-time type consistency, by performing type-based substitutability checks as part of the component binding and update processes. The method takes into account the environment of the currently deployed component version and uses its so-called contextual complement in the checks. This novel approach overcomes the limitations of the standard notion of compatibility by allowing non-contravariant differences on the required side of the component’s surface. The method was successfully implemented for the OSGi component framework, and in later parts of the paper we share the experiences gained through the implementation.

Keywords: component, application consistency, subtyping, compatibility, deployment context

1 Introduction
Component frameworks have become commonplace in software engineering and are increasingly often used to develop software systems with complex architectures. More and more projects see the benefit of frameworks like Spring or OSGi [18] despite their relative simplicity in terms of the underlying component models.

1 This work was partially supported by the Grant Agency of the Czech Republic under grant number 201/08/0266 “Methods and models for consistency verification of advanced component-based applications”.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Even though it is easier to manage complex architectures with components, safe evolution of these architectures — and preserving the internal consistency of the application throughout the evolution in particular — is still a challenging task. This is especially true for dynamic architectures (those which can evolve during application run-time) where changes cannot be fully anticipated by the architects and are not necessarily governed by any rules or patterns.

In our work we address the common scenario in which consistency verification is needed — the dynamic replacement of a component currently deployed in an architecture by another component, be it a completely different one (leading to the verification of substitutability) or a new version of the current one (resulting in the verification of backward compatibility as a special case). The verification is governed by the general principle of substitutability, summarised by Wegner and Zdonik [25]: a replacement component should be usable whenever the current one was expected, without the client noticing it.

1.1 Goals and Structure of this Paper

Research approaches to safe substitution aim to ensure reliable substitutability by employing formal methods. However, the models used in these approaches tend to be too complicated to be usable by average software developers and the methods often suffer from prohibitive algorithmic or space complexity (cf. for example [15,16]).

Industrial systems on the other hand almost exclusively use rather simple meta-data, most often version identifiers (e.g. [18]), to manually tag components as being compatible with their previous versions. The key disadvantage of this approach is fragility caused by the reliance on human effort to provide correct meta-data.

In this paper we describe a practically-oriented method of verifying substitutability of black-box components. It aims at being readily usable in today’s component frameworks with their relatively simple means of component specification while at the same time providing a high level of formally-backed assurance about substitutability.

The method therefore uses the (sub)type relation as its foundation since its formal strength is sufficient to prevent serious run-time errors while its evaluation can be done on current state of the practice component models, with relative simplicity and low algorithmic complexity. The method is only as strong in terms of formal verification of component’s substitutability as the component specification at hand allows. Mostly, it will therefore offer type-safety guarantees for current industrial component frameworks but it is able to incorporate the assessment of extra-functional properties compatibility [14] or advanced formal checking (e.g. to assert behavioural protocol compliance [19]) where appropriate data are available as part of the component’s specification.
A novel and significant aspect of the method is the possibility to overcome the limits of standard subtyping — strict covariance of provided and contravariance of required features — by considering the environment in which the components are deployed (e.g. the application architecture or component framework’s container), hereafter termed the deployment context.

The paper is structured as follows. The following section provides an overview of the state of the art in component substitutability. Section 3 contains foundational formal definitions of component type and deployment context, and section 4 defines the notion of contextual substitutability.

The second part of the paper contains validation of the formal part. In section 5 we describe an implementation of the method for the OSGi component framework and discuss several fundamental issues we encountered. Both the formal and practical aspects of the method are evaluated in section 6, and the paper concludes with notes on the subsequent work.

2 Related Work

This work contributes to the challenging area of correctness and robustness in component frameworks “in the absence of a closed-world assumption” [22]. This is still a relevant issue, since for example Taylor [23] notes that in service-oriented architecture research there has been “little attention to orchestrating [architectural] changes across service ( . . . ) boundaries”.

On a very high abstraction level, Georgas et al. [12] use a model of application architecture at run-time to manage its evolution. Constraints can be specified on the policies governing the evolution (adaptation) in order to preserve chosen architectural properties. The work however does not provide concrete details about the model, the constraints and ways to check them.

Many research approaches have addressed this need using holistic approaches with global integrity properties [21,11]. Chaki et al. [9] for example use compositional reasoning and dynamic assume-guarantee checks to provide formally sound evaluation of substitutability with similar practical properties as our contextual one.

Most of these methods are however based on advanced formal systems (e.g. model checking, behavioural subtyping) often supported by specialized specification notations. These methods tend to suffer from prohibitive algorithmic or state complexity [16,9] and the notations tend to be too complicated to be usable by average software developers [15].

Few research works have been concerned with use in industrial component frameworks. Polakovic et al. [20] implement architectural consistency checks for a resource-constrained component model, using a combination of compile-time type conformance verification and error handling code. Our approach would be hardly feasible in such cases due to the resource demands.
The work closest to ours in its spirit is Belguidoum and Dagnat’s [3] which extends an earlier version of our contextual substitutability [5] with the notion of forbidden dependencies. This prevents multiple conflicting implementations of the same component feature (e.g. a messaging service) to invalidate context invariants. While this is an important observation, the substitutability itself is formalized on a rather abstract level and further refinement of service comparison methods, effects of component (de)installation, and forbidden dependencies representation is needed for a full applicability of the approach.

Last but not least, several methods that use type systems have been proposed [11,17]. They moreover enable multi-component substitution which goes beyond our method proposed in this paper. However, their approaches have not been validated on industrial component frameworks.

3 Component Type and Deployment Context

To provide a sound basis for type-based substitutability and compatibility verification, a formal model at the type level of the compared components has to be available. In this section we describe such a model together with the representation of component’s view of its deployment context.

3.1 Type Representation of Component Interface

A component implementation, available in some form of distribution package, may comprise many different elements — executable code, meta-data, resource objects, etc. All elements that are part of the component’s interface (i.e. are accessible on the surface of the component’s black box) may participate in inter-component interactions — both the provided ones, used by component’s clients, and required ones, which express component’s dependencies that need to be satisfied in order to guarantee the provided functionality or properties.

We capture the structure of the component interface in the form of component type, a structured data type [8] which contains all such elements and their role with respect to inter-component interactions. The information about the component and its interface elements is assumed to be available from some kind of specification; the internals of the implementation are abstracted from since the component is a black box.

Definition 3.1 A component interface element is a tuple $e = (n, T, r, o, a)$ where $n \in \text{String} \cup \{\epsilon\}$ is the element’s name (possibly empty), $T$ is element’s type (in the respective specification language type system), $r \in \{\text{provided, required}\}$ is the element’s role on the component interface, $o \in \text{Boolean}$ is an indication whether the presence of the element at run time is optional, and $a \in N \cup \{\ast\}$ denotes arity (how many counterpart elements can be bound to it; ‘*’ stands for “any”).
Examples of component interface elements are: a named OSGi service typed to an interface or class, a Java interface implemented by a EJB component (in this case \( e.n = \epsilon \)) \(^2\), or property with a primitive type of a SOFA component. Optionality of the element can be indicated directly in the component specification (as in OSGi “optional” directive on imported packages or similar Fractal attribute of a component’s interface) or via element cardinality.

**Definition 3.2** Let \( E_C = \{e_i\}_{i \in I} \) (for a finite index set \( I \)) be the set of all component interface elements of a component implementation \( C \) which can be observed from outside of its black box.

The **component type** \( C \) of \( C \) is a pair of provided and required element sets: \( C = (E^P, E^R) \mid (E^P \cup E^R = E_C) \land (E^P \cap E^R = \emptyset) \) where \((\forall e \in E^P : e.r = \text{provided}) \land (\forall e \in E^R : e.r = \text{required})\).

The **component** as a run-time instance of \( C \) is a tuple \( c = (id, C, P, R) \) where \( id \) is a unique identification of the instance, \( C \) its component type, \( P \) and \( R \) are the sets of interface elements actually present at the instance’s interface. (We will abbreviate the expression “component \( C \) with type \( C \)” to just “component \( C \)” where unambiguous.)

The element role (provided, required) is distinguished in the component type for a fundamental reason: it affects handling of component during type operations, namely matching and subtype comparison. In this respect this definition of component type and its associated (sub)typing rules, while similar to standard structured data types, reflect the core notion of component-based programming.

The latter two parts of the component’s tuple represent its effective type at the given time. It holds that \( c.P \subseteq c.C.E^P \) and \( c.R \subseteq c.C.E^R \) because (some of) the optional elements may be omitted by the instance. We note further that the set of \( c \)’s interface elements may change in time, i.e. \( \exists t_1 \neq t_2 \cdot c.P/t_1 \neq c.P/t_2 \) and likewise with \( c.R \).

The subtype relation for interface elements is the standard reflexive transitive one, so it is a preorder on types:

**Definition 3.3** We say that component interface element \( e_i \) is a **subtype of element** \( e_j \) (denoted \( e_i <: e_j \)) if all of the following conditions hold:

- \( e_i.T :: e_j.T \) (the element types are in a subtyping relation);
- \( e_i.r = e_j.r \);
- \( \begin{cases} 
\neg e_j.o \Rightarrow e_i.o = \text{false} \quad \text{if } e_i.r = \text{provided,} \\
 e_j.o \Rightarrow e_i.o = \text{true} \quad \text{if } e_i.r = \text{required};
\end{cases} \)

\(^2\) Notational remark: Throughout the text, we use the dot notation to denote the individual parts of a tuple, so for \( A = (a, b) \) the expression \( A.a \) denotes the first part of \( A \)’s structure.
\[ \begin{align*}
  e_i.a &\geq e_j.a \text{ if } e_i.r = \text{provided}, \\
  e_i.a &\leq e_j.a \text{ if } e_i.r = \text{required}.
\end{align*} \]

The next subsection formalizes the representation of the environment in which a component is deployed.

### 3.2 Component Context

We noted in the introduction that it is useful to capture the deployment environment of a particular component for evaluating substitutability. This component deployment context contains the other components and architectural connections within the environment in which the component is employed. The environment can be a component cluster (a closely coupled part of a component application), the component-based application or the whole run-time environment surrounding a deployed component in the run-time framework.

**Definition 3.4** The deployment context of a component instance \( c \) is a tuple \( D = (K, B) \) where \( K = \{ c_i \}_{i \in I} \), \( c \in K \) is the set of components existing in the deployment environment of \( c \), and \( B = \{ (e^p, e^r) \mid \exists C_p, C_r \in K \cdot (C_p \neq C_r) \land (e^p \in C_p.P) \land (e^r \in C_r.R) \land e^r \text{ is bound to } e^p \} \) are the bindings between component elements within the context.

A deployment context \( D \) is **architecturally consistent** if all the inter-component bindings are correctly resolved (including the matching of types of bound elements) and the components are correctly functioning.

A subset of this environment which is particularly interesting from the substitutability point of view is a component’s complement within the deployment context. The model of the complement uses the same abstractions as that of the component type described in the previous subsection.

**Definition 3.5** Assume an architecturally consistent context \( D \) and component \( c \in D.K \) with component type \( C \). The contextual complement of \( c \) in \( D \) is a “virtual” component type \( \bar{C}_D = (\bar{P}, \bar{R}) \) such that

- \( \bar{P} = \{ e \mid (\exists c_r \in D.K \cdot c_r \neq c, e \in c_r.R) \land (\exists e_p \in c.P \cdot (e_p, e) \in D.B) \} \) – this set consists of the actual client elements of \( c \)’s provided ones.
- \( \bar{R} = \bigcup_{s \in D.K} e_{c_s}.P \mid c_s \in D.K, c_s \neq c \) – this set consists of all elements available in \( D \) which can satisfy a component’s requirements;

The complement can be seen as an inverted effective type of \( c \) at the given time (cf. the \( P, R \) element sets of the component instance). It captures the following two aspects of the context from the point of view of the component:

(i) The real usage of the component’s provided elements, as given by the bindings to particular required elements of other components.
(ii) The elements available in the environment that can possibly satisfy any $c$’s substitute component’s requirements, most commonly via other component’s provided elements.

Fig. 1. Component in an architecture and the elements forming its contextual complement

Several interesting observations can be made about the contextual complement from typing perspective. They are summed up in the following lemma.

**Lemma 3.6 (Properties of contextual complement)** Assume a component $c$ with type $C = (P,R)$ and its contextual complement $\bar{C}_D = (\bar{P},\bar{R})$ according to the definitions above. Let $P' = \{ e' \mid (e' \in P) \land (\exists \bar{e} \in \bar{P} : (e',\bar{e}) \in D.B) \}$ be the set of actually bound provisions of $C$. Then it holds that

(i) $P' \subseteq P$ (not all provisions need to be bound).

(ii) $\forall \bar{e} \in \bar{P}, e \in P : (e,\bar{e}) \in D.B : e.T <:\bar{e}.T$ (context’s bound required elements can have generalized types).

(iii) $\forall e \in R : e.o = \text{false}$ $\exists \bar{R}_e \subseteq \bar{R}$ such that $\bar{R}_e \neq \emptyset \land \forall \bar{e} \in \bar{R}_e : \bar{e}.T <: e.T$ (all component’s mandatory requirements are satisfiable by the context, via one or more elements with possibly specialized types).

**Proof.** The proof of these properties is straightforward:

(i) If all $e \in c.P$ are bound to client elements in the context, then $P' = P$; otherwise, $|P| - |P'|$ is the number of unbound provided elements.

(ii) Assume there is a required element $\bar{e}_x$ of some component $c_x \in D.K$ bound to a provided element $e$ of $c$ (consequently $\bar{e}_x \in \bar{R}$) such that $\bar{e}_x.T <: e.T$. Then the binding of the provided $e$ to the other component’s required $\bar{e}_x$ would be type-unsafe, since $e$ cannot cover all the type features of $\bar{e}_x$. This violates the assumption of architectural consistency of the deployment context. (In practice, the run-time framework would decline to establish such binding a-priori.)

(iii) (a) Assume $\bar{P}_e = \emptyset$, meaning the non-optional required element $e$ isn’t bound to any corresponding provider. This again violates the architectural consistency assumption. (b) Assume $e.T <: \bar{e}.T$ which is a situation analogical to point 2. The type unsafety of the binding between $e$ and
would eventually lead to a malfunction of \( c \), which contradicts the assumptions (with the same practical interpretation).

\[ \square \]

Note finally that the \( \mathcal{P} \) captures the real types of elements bound to the actually used \( c \cdot P \) elements, not simply the types of these, and that there is no requirement for element name uniqueness in the \( \mathcal{P}, \mathcal{R} \) sets.

The idea of component’s complement and its properties is illustrated on Figure 1. Component \( \text{Gate} \) is deployed in a simple architecture, bound to the \( \text{Control} \) and \( \text{ParkingPlace} \) components. An additional \( \text{TrafficLane} \) component is installed in the run-time framework. The deployment context of \( \text{Gate} \) comprises the two counterparts of its provided interfaces in the \( \mathcal{R} \) set, and the three other provided interfaces available in the framework as its \( \mathcal{P} \) set.

4 Component Substitutability

The principle of substitutability introduced in Section 1 provides a very general definition of the notion. In the context of component-based software engineering, it can be elaborated upon by taking into account the features available in the current state-of-the-art component models.

The replacement component is substitutable for a component currently deployed in a consistent application architecture if it satisfies the following general requirements [4]:

(i) Presents the same operational interface (at the syntactic and typing level) to its environment.

(ii) Exchanges the same data (with respect to their location and format) as the current one.

(iii) Conforms to the semantics and behavioural specifications of the current component in all interactions in which it is engaged.

(iv) Exhibits compatible extra-functional (quality of service) characteristics.

In the approach presented here we concentrate on the first area, which is a deliberate simplification of the issue. The rationale for this decision is based on the challenges faced when working with industrial component frameworks. There, specifications of advanced aspects are not available or cannot be reconstructed from implementation in most cases; therefore, especially semantic compatibility is hard to verify.

We therefore need to base the formal notion of substitutability only on such artifacts and abstractions that are products of standard component development process. This lead us to (a) working with information directly available in the component distribution package — semi-formal component interface specifications, possible meta-data created during development, and data ex-
tracted by run-time component introspection; (b) using the least common denominator of the formal foundations — the type system and its subtyping rules — which are always available for any programming or specification language and provide a reasonable degree of trust in the conclusions as to the run-time safety of substitute component.

Let us now define the basic kind of type-based substitutability (presented in earlier versions in [6]). The following section then presents the novel kind of substitutability which considers the deployment context.

**Definition 4.1** We say that a (replacement) component type $R = (P', R')$ is **strictly substitutable** for the (current) component type $C = (P, R)$ if $(\forall e \in P \exists e' \in P' : e'.n = e.n \land e' <: e) \land (\forall e' \in R' \exists e \in R : e'.n = e.n \land e <: e')$. This fact is denoted as $R \prec C$.

The definition corresponds to the natural understanding of “vertical” compatibility [3]: the replacement component provides at least the same, and requires at most the same, component interface elements with respect to their names and types (irrespective of element’s optionality). It uses the common notion of co- and contra-variance at the component type level (cf. [24] or [7]). This definition ensures a-priori substitutability of any pair of a component instance and its replacement which have the types $C$ and $R$.

### 4.1 Contextual Substitutability

The principle of substitutability tells us that this property does not concern just the two components (current, replacement) in question: we also need to take into account their use by clients. From this point of view, changes in the provided and required parts of component interface do not affect substitutability equally.

In many component-based architectures not all of the component’s provided features are utilised, i.e. bound to clients. On a case-by-case basis, these unused features can therefore be omitted when evaluating substitutability in the given deployment context. Similarly, it is common that new features are added during component evolution which results in the need to add corresponding dependencies to make them work. In the programming language research this led to the notion of covariance. In the case of deployed software components, we can take the advantage of the knowledge of deployment context and match the replacement component’s extended requirements with any of those provided by the context’s components.

From the architectural point of view this is a clean solution since the component should be agnostic of who is providing the required functionality, as long as it conforms to its stated specification. This leads to the following definition:
**Definition 4.2** Given a currently deployed component $c$ with type $C$ and its contextual complement $\bar{C}_D$, we say that the (replacement) component type $R = (P', R')$ is **contextually substitutable** for $C$ if it holds that $R \prec \bar{C}_D$ that is if $(\forall \bar{e} \in \bar{P} \exists e' \in P' : e' <: \bar{e}) \land (\forall e' \in R' \exists \bar{e} \in \bar{R} : \bar{e} <: e')$. This is denoted $R \prec_D C$.

In plain words, the contextually substitutable replacement component provides at least the same features as are those used by the clients of the current one, and requires at most what is available in the context. It can be said it is horizontally compatible [3] with the context.

![Diagram](image)

**Fig. 2.** New component version vs. the contextual complement of the currently deployed one

Continuing with the example introduced in the previous section, Figure 2 shows a second version of the *Gate* component that should replace the original one. Its requirements are clearly greater than those of the original version, moreover one of the provided interfaces has changed its type. However, comparing *Gate v2* to the contextual complement of its currently deployed version shows that it can actually be used in the given architecture (provided the *LaneGateStats* interface is a subtype of *CountingStats*).

Intuitively, one would expect that strict substitutability implies contextual.

**Proposition 4.3 (Strict substitutability implies contextual)** Assume components $c$ and $r$ with types $C$ and $R$ respectively. It holds that $\forall D : c \in D.K : R \prec C \Rightarrow R \prec_D C$, i.e. if $R$ is strictly substitutable for $C$ then it is also contextually substitutable for $C$ in any architecturally consistent deployment context $D$.

**Proof.**

We first need to prove that $\forall D : C \prec_D \bar{C}_D$ meaning that $C$ “fits in” its (any) context. Using the standard notation $C = (P, R)$ and $\bar{C}_D = (\bar{P}, \bar{R})$ this can be done in two parts:

- $\forall \bar{e} \in \bar{P} \exists e \in P : e <: \bar{e} \ (P$ is a substitute for $\bar{P}) \ \text{– follows from Lemma 3.6 items 1 and 2.}$
- $\forall e \in R \exists \bar{e} \in \bar{R} : \bar{e} <: e \ (R$ is a “supertype of” $\bar{R}) \ \text{– follows from Definition 3.4 ($|R| \leq |\bar{R}|$) and Lemma 3.6 item 3.}$
From the assumptions of the proposition we have $R \prec C$, the above says that effectively $C \prec \bar{C}_D$ and since the substitutability relation is transitive, it follows that $R \prec \bar{C}_D$.

□

This fact can be useful in certain common scenarios, e.g. in the special case of component backward compatibility: for a subsequent revision of a component we can easily prove strict substitutability with its immediately preceding revision at component release, store appropriate indication in its meta-data, and use it when upgrading the component.

Only if no such indication is available the assessment of substitutability must be carried out at the component binding or upgrade time. At this time it also makes sense to perform the contextual substitutability checks.

Due to its time-dependent nature, both the component’s effective type (the actually used provided and required elements) and the deployment context may change in component instance’s lifetime. Once compatibility is verified during upgrade, architectural consistency needs therefore to be continuously verified and ensured by other means.

5 Realization for the OSGi framework

Building on our previous work [6] we have implemented a contextual substitutability verifier for the OSGi framework [18]. The overall technical design of the verifier was driven by several goals motivated by the need for practical utility. One of them was a simplified scope (evaluate compatibility of subsequent bundle revisions, not any-to-any substitutability checks), another one a non-intrusive integration in the host framework.

The verifier application has the form of a set of OSGi bundles. The overall architecture of the implementation comprises three layers — a simple user interface, bundle and context representation loaders plus substitutability verifier (comparator), and an underlying Java type system model and subtyping rules implementation.

The first two layers are shown in Figure 3. Once the type representations for both the contextual complement and the replacement bundle are created by the Loader bundles (forming a tree data structure with provisions for primitive types and circular references), they are submitted to the Bundle Comparator which implements the substitutability verification. The result of its work is essentially an annotated type tree. It is aggregated into a single assertion about the type relation between the bundle and the complement. The Substitution Verifier bundle wraps the whole process, taking care of activating the loaders and comparator, and interpreting its result for the user.

Bundle is the OSGi term for a component.
There are several fundamental issues that an implementation of contextual substitutability needs to address in general. Firstly it has to implement the element subtype relation at run-time, and choose appropriate type representation on which to perform it. Secondly there needs to be a means of extracting this component types and contextual complement representation from various sources. In the following paragraphs we will discuss our approach to addressing these issues for the OSGi case.

5.1 Type Representation

The foundational issue is the means of obtaining and representing the types of elements contained in the component specification. Normally this issue is delegated to the relevant language compiler; however, in our case a run-time component type representation is needed together with mechanisms to obtain it from both the installed and replacement components. Additional complication is that in OSGi, the specification data is scattered in several places (the manifest file as the pivotal point, XML and other additional meta-data e.g. for the declarative services, and the bytecode of bundle implementation).

Since no suitable run-time type representation of OSGi bundles is available, we created a custom-build model [1] called BundleTypes. It consists of domain classes capturing selected characteristics of the whole bundle, both at the module layer (its exported and imported packages) and the service layer (provided and depended-on services). This representation then references a lower layer model, called JavaTypes, which captures the type information of the individual Java classes.

The reconstruction of the type representations uses different means depending on the bundle in question. For the replacement bundle we use bytecode analysis with the ASM library\(^4\) wrapped by a custom classloader. Also, stubs are created for shared (JRE, OSGi core, . . .) or unreachable classes. These techniques have to be used because the bundle package of the replace-

\(^4\) http://asm.ow2.org/
ment version is accessible only as standalone .jar file in the filesystem (the component has not been installed in the framework yet) so we can’t use reflection and framework APIs. For the imported packages in particular we assemble their type representation [2] from class and operation references extracted from the bundle bytecode.

To obtain the type representation of the current bundle and its contextual complement, we use standard OSGi framework services (package admin, bundle metadata and classloader methods) and Java reflection API. This information is easily reachable since the bundle’s metadata and bindings to client and provider components are available in the respective framework registries.

5.2 Subtype and Compatibility Verification

Concerning subtype relation implementation, the design of the algorithms had to reconcile the differences between the theoretical notion of the type relation — as used in the previous section — and the rules employed by Java as the actual specification language, its linking mechanism and the run-time system of the OSGi framework.

Most prominently, Java uses subclassing rather than subtyping [10] in its type matching rules and differentiates subtyping from binary compatibility [13]. This relation is actually the source of the underlying element “subtype” relation since OSGi bundles are bound and updated as binary .jar files.

In the bundle substitutability verifier, subtyping is evaluated by the JavaTypes layer and the results are aggregated by the BundleTypes layer to represent compatibility at both the module and service layer of OSGi bundles. Although OSGi is rich in features and modifiers at the module layer (optional imports, “uses” constraint, version ranges, etc.) most of them do not affect substitutability on the exported side and the effective type on the imported side (which reflects the effects of these modifiers) is easily obtained. At the service layer, element substitutability is verified implicitly — all service interfaces must be declared in the exported/imported packages so their type comparison is handled at the module layer.

6 Evaluation and Lessons Learned

The presented method of component substitutability verification can be considered a rather simple one. It uses only typing rules as its basis, leaving out the much more powerful levels of semantic and behavioural compatibility.

Apart from the reasons given in preceding sections, this design can be defended for the following fundamental reason: the method does not place any limitations on the kinds of component interface elements it is applied to. Therefore it can incorporate any semantic or behavioural specification
compatible with our model of component type. An example of an advanced kind of component interface element for which our method could be applied is the behaviour protocol [19] originated in the SOFA component model. In its case, the protocol compliance relation plays the role of element subtyping.

With respect to the presented OSGi substitutability verifier, we would like to share several observations. For obtaining the type representation of the imported packages, the bytecode analysis techniques used are working solutions but neither method is completely reliable — e.g. the bytecode need not contain return types of class operations. The only good solution to this problem would probably be to somehow include the type information of the referenced classes with the component distribution package.

Also, it is essentially difficult to reliably obtain the complete list of bundle’s services for both standalone (not installed) and running bundles. The reason in the first case is that the bundle meta-data need not statically declare the services which use the simple core OSGi model so even if the bundle uses the alternative declarative services model with explicit specification, some services may not be found.

The situation is only partially better in the latter case: the sets of services which a bundle exports and uses are available via the framework API but they may change in time. Thus the bundle context representation is only on a current snapshot (plus possibly the history) of the bundle’s bindings and may not cover all its elements. Evaluation of component substitutability for the OSGi framework will therefore always suffer from potential incorrectness, due to the nature of the component model.

The current implementation of the substitutability verifier has several shortcomings. At the bundle representation and comparison level, it does not handle fragment bundles and optional imports. It also intentionally omits dynamic imports and bundle dependencies (known bad practices in the OSGi world). Also, the chosen architecture — implementing the tool as user-space bundles — enables its portability but prevents integration into the bundle installation/resolving/updating process (which would be a desirable goal since it would provide user-transparent compatibility verification). These issues are the subject of further improvements of the implementation.

7 Conclusion

In this paper we presented the formal definition and practical implementation of a component substitutability verification method. Its key contributions are the novel use of the component’s deployment context to enable safe substitution for non-subtype replacement components, and the ability to provide

\[5\] The Export-Service and Import-Service manifest headers are deprecated.
sufficiently strong formal guarantees of type consistency even when applied on current industrial component frameworks.

This type-based substitutability verification can be wrapped into easy to use tools and data that promote its practical use. One such practical extension implemented by our team is the automated creation of correct semantic version identifiers for the OSGi framework.

Concerning further research, the formal definitions of the method should be extended to clusters of components (e.g. to support safe substitution of larger sub-sets of applications) and applied more specifically to inter-component relations in dynamic architectures. The practical implementation for OSGi will need to supply the missing aspects of the component model, and overcome the issues of tighter integration in the frameworks.

References


Brada


Combining Proof and Model-checking to Validate Reconfigurable Architectures

Arnaud Lanoix
LINA, Nantes University, Nantes, France

Julien Dormoy Olga Kouchnarenko
LIFC, University of Franche-Comté, Besançon, France

Abstract
This paper deals with the formal specification and verification of dynamic reconfigurations of component-based systems. To validate such complex systems, there is a need to check model consistency and also to ensure that dynamic reconfigurations satisfy architectural and integrity constraints, invariants, and also temporal constraints over (re)configuration sequences. As architectural constraints involve first-order formulae, and a behavioural semantics of reconfigurations gives rise to infinite state systems, we propose to associate proof and model-checking within the well-established B method, to support the modelling of such systems and the (partial-)validation of their dynamic reconfigurations. The objective of the paper is twofold. First, given a hierarchical B model of component-based architectures, we validate it by proving its consistency. Second, given linear temporal logic formulae expressing the desirable dynamic behaviour of the system, we validate reconfigurable system architectures by using bounded model-checking tools supporting the B method. The main contributions are illustrated on the example of a HTTP server architecture.

Keywords: Component, Architecture, Reconfiguration, Proof, Model-checking, B method

1 Introduction
This article is dedicated to an automatic checking of dynamic reconfigurations of component-based systems whose development provides significant advantages like portability, adaptability, re-usability, etc. Dynamic reconfiguration of distributed applications is an active research topic [2,3,17] motivated by practical distributed applications like, e.g., those modelled in Fractal [9]. In
many recent works, the idea of using temporal logics to specify dynamic reconfigurations and to manage applications at runtime has been explored [7,15,12].

In [12], a formal semantics of component-based systems architectures with reconfigurations together with a linear time temporal logic over (re)configuration sequences have been proposed. This logic, called FTPL, is based on architectural constraints and on event properties. To validate such complex systems, there is a need to check model consistency and also to ensure that dynamic reconfigurations satisfy architectural and integrity constraints, invariants, and also FTPL constraints. As these constraints involve first-order formulae, and a behavioural semantics of reconfigurations gives rise to infinite state systems, we propose to combine proof and model-checking techniques within the well-established B method [1].

Let us explain our current B-based validation approach and contributions on Fig. 1. First of all, we propose to model components and basic dynamic reconfigurations using the B formal framework. It allows us to define and to validate a generic B model for component architectures. We use the AtelierB tool to interactively prove the architectural constraints consistency (1). Then this generic architecture is instantiated to represent an architecture under consideration (2); The dynamic architectural reconfigurations are specified by using the previously defined primitive B operations (3). Note that we do not take into consideration specification of the controller which would use these dynamic reconfigurations (through adaptation policies as instance). Consequently, the validation of the instantiated architectural model cannot be done using interactive proof. Nevertheless, we perform a partial validation of these dynamic reconfigurations through animations of samples of the running model, thanks to the ProB model-checker features (4). In addition, temporal properties over configuration sequences expressed in FTPL are translated into LTL (5). These properties can be checked with the ProB model-checker (6).

The remainder of the paper is organised as follows. After giving a motivating example in Sect. 2, the B method and its tools supports are introduced in Sect. 3. We formally define a generic B model of component architectures in Sect. 4. This model is then instantiated to validate a particular architecture in Sect. 5. To automatically verify temporal properties, Section 6 introduces
Lanoix, Dormoy, Kouchnarenko

FTPL and gives its translation into LTL. Finally, Section 7 concludes before discussing related work.

2 Motivating Example

To motivate and to illustrate our approach, let us consider an example of an HTTP server from [10]. The architecture of this server is displayed in Fig. 2.

The RequestReceiver component reads HTTP requests from the network and transmits them to the RequestHandler component. In order to keep the response time as short as possible, RequestHandler can either use a cache (with the component CacheHandler) or directly transmit the request to the RequestDispatcher component. The number of requests (load) and the percentage of similar requests (deviation) are two parameters defined for the RequestHandler component:

- The CacheHandler component is used only if the number of similar HTTP requests is high.
- The memorySize for the CacheHandler component must depend on the overall load of the server.
- The validityDuration of data in the cache must also depend on the overall load of the server.
- The number of used file servers (like the FileServer1 and FileServer2 components) used by RequestDispatcher depends on the overall load of the server.

![Fig. 2. HTTP Server architecture](image)

We consider that the HTTP server can be reconfigured during the execution by the following reconfiguration operations:

(i) AddCacheHandler and RemoveCacheHandler which are respectively used to add and remove the CacheHandler component when the deviation value increased/decreased around 50;

(ii) AddFileServer and removeFileServer which are respectively used to add and remove the FileServer2 component;

(iii) MemorySizeUp and MemorySizeDown which are respectively used to increase and to decrease the MemorySize value;
(iv) DurationValidityUp and DurationValidityDown to respectively increase and decrease the ValidityDuration value.

As an illustration, we specify the AddCacheHandler reconfiguration expressed in the FScript language [11]. When the deviation value exceeds 50, the reconfiguration consists in instantiating a CacheHandler component. Then, the component is integrated into the architecture, and the binding with the required interface of RequestHandler is established. Finally, the component CacheHandler is started.

```plaintext
1 action AddCacheHandler(root)
2 if(value($context/child: ∈ RequestHandler/attribute: ∈ deviation) > 50){
3   newCache = new("CacheHandler");
4   add($root, $newCache);
5   bind($root/child: ∈ RequestHandler/interface: ∈ getcache, $newCache/interface: ∈ cache);
6   start($newCache);
7 }
```

3 Proof-based Approach: the B Method

B is a formal software development method used to model systems and to reason about their development [1]. When building a B model, the principle is to express system properties which are always true after each evolution step of the model, the evolution being specified by the B operations. The verification of a model correctness is thus akin to verifying the preservation of these properties, no matter which step of evolution the system takes.

The B method is based on set theory, relations and first-order logic. Constraints are specified in the INARIANT clause of the model, and its evolution is specified by the operations in the OPERATIONS clause. Let us assume here that the initialisation is a special kind of operation. In this setting, the verification of a B model consists in verifying that each operation—assuming its precondition and the invariant hold—satisfies the INARIANT, i.e. the model is consistent. A strength of the B method is its stepwise refinement feature: each refinement makes a model more deterministic and also more precise by introducing programming language-like features.

Tool supports, such as B4free or AtelierB³, automatically generate proof obligations (POs) to ensure the consistency in sense of B [1]. Some of them are obvious POs whereas the other POs have to be proved interactively if it was not done fully automatically by the different provers embedded into AtelierB. Another tool, called ProB⁴, allows the user to animate B specifications for their debugging and testing. On the verification side, ProB contains a constraint-based checker and a LTL bounded model-checker with particular

⁴ http://www.stups.uni-duesseldorf.de/ProB
features; Both can be used to detect various errors in B specifications [18,19].

4 Specifying a General Architectural Model with B

In [12], we have defined a configuration to be a set of architectural elements (components, interfaces and parameters) together with a relation to structure and to link them, through a graph-based representation. The model we have proposed was inspired by the model in [16,17] given for Fractal. Unlike [16,17], in our model only the basic and generic concepts are considered to allow their application to various hierarchical component models: components as runtime entities, required and provided interfaces as interaction points between components, bindings to link component interfaces. Components are either primitive or composite components. Only primitive components can have some attributes used as configuration parameters.

Component-based models must provide mechanisms for systems to be dynamically adapted—through their reconfigurations—to their environments during their lifetime. These dynamic reconfigurations may happen because of architectural modifications specified in primitive operations. Notice that reconfigurations are not the only manner to make an architecture evolve. The normal running of different components also changes the architecture by modifying parameter values or stopping components, for instance.

In this paper, we give a B specification of the generic model for component architectures in [12]. Firstly, we express all the architectural constraints between the architectural elements as B properties and invariants; secondly, we model the primitive reconfiguration operations as B operations; thirdly, we prove the consistency of this architectural B model.

4.1 Specifying the Architectural Configurations with B

The architectural elements we consider are the core entities of a component-based system: COMPONENTS, INTERFACES, INTERFACE_TYPE, and the component PARAMETERS; and relations over them to express various links between these architectural elements. Some of these relations do not evolve during the system reconfigurations. They are then defined as B CONSTANTS, and architectural constraints over them are expressed in the PROPERTIES clause.

<table>
<thead>
<tr>
<th>SETS</th>
<th>COMPONENTS ; INTERFACES ; INTERFACE_TYPE ; PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANTS</td>
<td>ProvidedInterfaces , RequiredInterfaces , InterfaceType , Provider , Requirer , Contingency , Definer</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>ProvidedInterfaces ⊆ INTERFACES ∧ RequiredInterfaces ⊆ INTERFACES ∧ ProvidedInterfaces ∪ RequiredInterfaces = INTERFACES ∧ ProvidedInterfaces ∩ RequiredInterfaces = Ø ∧ InterfaceType ∈ INTERFACES → INTERFACE_TYPE</td>
</tr>
</tbody>
</table>
∧ Provider ∈ ProvidedInterfaces → COMPONENTS
∧ Requirer ∈ Required Interfaces → COMPONENTS
∧ Contingency ∈ RequiredInterfaces → CONTINGENCY
∧ Definer ∈ PARAMETERS → COMPONENTS

The ProvidedInterfaces and RequiredInterfaces are defined to be subsets of INTERFACES. Their union is disjunctive. InterfaceType is a total function that associates a type with each required and provided interface. Provider is a total surjective function which gives the component having at least a provided interface, whereas Requirer is only a total function. Contingency is a total function which indicates for each required interface if it is mandatory or optional. Definer is a total function which gives the component of a considered parameter.

The other architectural relations we consider may evolve. They are defined as B VARIABLES, and architectural constraints over them are expressed in the INVARIANT clause.

VARIABLES
Instantiated Components, Parent, Binding, ProvDelegate, ReqDelegate,
State, Value

INVARIANT
InstantiatedComponents ⊆ COMPONENTS
∧ Parent ∈ InstantiatedComponents → InstantiatedComponents
∧ ran(Parent) ∩ ran(Definer) = Ø
∧ closure1(Parent) ∩ id(InstantiatedComponents) = Ø

InstantiatedComponents is a subset of COMPONENTS. Parent is partial function linking sub-components to the corresponding composite component. Composite components have no parameter, and a sub-component must not be a composite including its parent component, and so on.

∧ Binding ∈ ProvidedInterfaces → RequiredInterfaces
∧ ∀ (iprov, ireq) . (iprov → ireq ∈ Binding ⇒ ( Provider(iprov) ∈ InstantiatedComponents
∧ Requirer(ireq) ∈ InstantiatedComponents
∧ Provider(iprov) = Requirer(ireq) )
∧ InterfaceType(iprov) = InterfaceType(ireq) ) )
∧ dom(Binding) ∩ dom(ProvDelegate) = Ø
∧ ran(Binding) ∩ dom(ReqDelegate) = Ø

Binding is a partial function which connects together a provided interface and a required one: a provided interface can be linked to only one required interface, whereas a required interface can be the target of more than one provided interface. Moreover, two linked interfaces do not belong to the same component, but their corresponding instantiated components are sub-components of the same composite component. The considered interfaces must have the same interface type, and they have not yet been involved in a delegation.
\[ \forall \text{ProvDelegate} \in \text{ProvidedInterfaces} \mapsto \text{ProvidedInterfaces} \]
\[ \forall (\text{isub}, \text{isuper}) . (\text{isub} \mapsto \text{isuper} \in \text{ProvDelegate} \Rightarrow (\text{isub} \neq \text{isuper}) \]
\[ \land \text{Provider(\text{isub})} \in \text{InstantiatedComponents} \]
\[ \land \text{Provider(\text{isuper})} \in \text{InstantiatedComponents} \]
\[ \land \text{Parent(\text{Provider(\text{isub})})} = \text{Provider(\text{isuper})} \]
\[ \land \text{InterfaceType(\text{isub})} = \text{InterfaceType(\text{isuper})} \}
\]
\[ \forall \text{ReqDelegate} \in \text{RequiredInterfaces} \mapsto \text{RequiredInterfaces} \]
\[ \forall (\text{isub}, \text{isuper}) . (\text{isub} \mapsto \text{isuper} \in \text{ReqDelegate} \Rightarrow (\text{isub} \neq \text{isuper}) \]
\[ \land \text{Requirer(\text{isub})} \in \text{InstantiatedComponents} \]
\[ \land \text{Requirer(\text{isuper})} \in \text{InstantiatedComponents} \]
\[ \land \text{Parent(\text{Requirer(\text{isub})})} = \text{Requirer(\text{isuper})} \]
\[ \land \text{InterfaceType(\text{isub})} = \text{InterfaceType(\text{isuper})} \}
\]

Provision and Requirement delegation links express delegation links and are similarly defined. They are both partial bijections that associate a provided (resp. required) interface of a sub-component with a provided (resp. required) interface of its composite: the parent of the component which provides (resp. requires) isub must be the provider (resp. requirer) of isuper. Finally, both interfaces must have the same type, and they have not yet been involved in a binding.

\[ \land \text{State} \in \text{InstantiatedComponents} \mapsto \text{STATE} \]
\[ \forall (\text{ireq}) . (\text{ireq} \in \text{RequiredInterfaces} \]
\[ \land \text{Contingency(\text{ireq})} = \text{mandat} \lor \text{\forall} \]
\[ \land \text{Requirer(\text{ireq})} \in \text{InstantiatedComponents} \]
\[ \land \text{State(\text{Requirer(\text{ireq})})} = \text{started} \]
\[ \Rightarrow (\text{ireq} \in \text{ran(Binding)} \lor \text{ireq} \in \text{dom(ReqDelegate)}) \}
\]
\[ \land \text{Value} \in \text{PARAMETERS} \mapsto \text{INT} \]

State is a total function which associates a value from \{started, stopped\} with each instantiated component: a component can be started only if all its mandatory required interfaces are bound or delegated. Last, Value is a total function which gives the current value of a considered parameter.

### 4.2 Modelling the Dynamic Reconfigurations with B

Once a configuration-based model is given, the primitive reconfiguration operations can be specified as B operations of this B model. Namely, we specify:

- instantiate (newComponent) and delete (component) to instantiate/destroy a component;
- add(subComponent,composite) and remove(subComponent) to add/remove sub-components to/from a composite;
- bind(iprov, ireq) and unbind(iprov) to bind/unbind component interfaces;
- delegate(isub, isuper) and undelegate(isub) to delegate/undelegate component interfaces;
- start (component) and stop(component) to start/stop components;
- set(parameter, newValue) to set a new parameter value.
Let us detail some of these B operations. The example below defines in B the primitive reconfiguration operation adding a subcomponent to a composite component:

```
add(subcomponent, composite) =
PRE subcomponent ∈ COMPONENTS ∧ composite ∈ COMPONENTS THEN
SELECT
 subcomponent ∈ InstantiatedComponents
 ∧ composite ∈ InstantiatedComponents
 ∧ subcomponent ≠ composite
 ∧ composite ∉ ran(Definer)
 ∧ subcomponent ∉ dom(Parent)
 ∧ subcomponent→composite ∉ Parent
 ∧ composite→subcomponent ∉ closure1(Parent)
 ∧ ∀ (iprov) . ( ( iprov ∈ ProvidedInterfaces ∧ Provider(iprov) = subcomponent )
 ⇒ ( iprov /∈ dom(Binding) ∧ iprov /∈ dom(ProvDelegate) ) )
 ∧ ∀ (ireq) . ( ( ireq ∈ RequiredInterfaces ∧ Requirer(ireq) = subcomponent )
 ⇒ ( ireq /∈ ran(Binding) ∧ ireq /∈ dom(ReqDelegate) ) )
 THEN
 Parent(subcomponent) := composite
 END
END ;
```

The `add(subComponent,composite)` operation must establish that the both components are instantiated components, `composite` is a composite component (i.e. a component without parameters). Moreover, `subComponent` is not a subcomponent of another composite nor is already used: none of its interfaces is bound or delegated. Finally, the modification does not introduce a cycle into `Parent`.

The binding of component interfaces is expressed by `bind(iprov, ireq)`: this operation must establish that the considered interfaces are correct, i.e. they are provided (resp. required) interfaces of instantiated components, they are not bound nor delegated, and their types are compatible. Furthermore, the binding has not been done yet.

```
bind(iprov, ireq) =
PRE iprov ∈ INTERFACES ∧ ireq ∈ INTERFACES THEN
SELECT
 iprov ∈ ProvidedInterfaces
 ∧ ireq ∈ RequiredInterfaces
 ∧ Provider(iprov) ∈ InstantiatedComponents
 ∧ Requirer(ireq) ∈ InstantiatedComponents
 ∧ iprov /∈ dom(Binding)
 ∧ ireq /∈ dom(Binding)
 ∧ InterfaceType(iprov) = InterfaceType(ireq)
 ∧ iprov→ireq /∈ Binding
 THEN
 Binding(iprov) := ireq
 END
END ;
```
The unbinding primitive operation is specified as follows: this operation expresses as precondition that the considered interface is provided by an instantiated component. This provider must be stopped. Moreover, a required interface bound with the considered interface, must exist. Then, the considered interface is removed from Binding.

When considering a started component, applying stop(component) changes to stopped the state of the considered component and the states of all its sub-components, if they exist; and so on.

Once these primitive reconfiguration operations are specified, more complex reconfiguration operations can be written, as explained in Sect. 2.

4.3 Validating the Architectural B Model

We use AtelierB to validate the consistency of our generic B model for component architectures. The tool generates proof obligations (POs) to check the consistency of all the architectural constraints expressed in the INVARIANT and the fact that each B reconfiguration operation respects these architectural constraints.

<table>
<thead>
<tr>
<th>PO</th>
<th>Proved</th>
<th>Unproved</th>
<th>% proved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialisation</td>
<td>30</td>
<td>30</td>
<td>25 (30)</td>
</tr>
<tr>
<td>instantiate</td>
<td>20</td>
<td>18</td>
<td>2 (1)</td>
</tr>
<tr>
<td>delete</td>
<td>27</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>add</td>
<td>18</td>
<td>14</td>
<td>4 (3)</td>
</tr>
<tr>
<td>remove</td>
<td>18</td>
<td>17</td>
<td>1 (1)</td>
</tr>
<tr>
<td>bind</td>
<td>9</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>unbind</td>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>start</td>
<td>3</td>
<td>0</td>
<td>3 (3)</td>
</tr>
<tr>
<td>stop</td>
<td>3</td>
<td>0</td>
<td>3 (3)</td>
</tr>
<tr>
<td>set</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TOTAL</td>
<td>139</td>
<td>114</td>
<td>25 (11)</td>
</tr>
</tbody>
</table>

The AtelierB generates 139 POs. Notice that the work on POs is in progress: at this time, 114 of them are automatically or interactively discharged. The proof of 25 POs remains to be done.

That does not mean that they are false, but that the AtelierB provers simply don’t succeed in demonstrating the rule: it may be due to the fact that some B operators are not handled in an efficient manner by the provers (like closure1(), for 11 unproved POs), or due to the fact that the heuristics
used for the proof are not efficient enough in this case. An effort remains to be done to manually demonstrate the unproved POs.

5 Validating a Specific Architecture

As summarized in Fig. 1, our approach consists in instantiating the generic architectural B model to specify and to verify a particular architecture. Primitive reconfiguration operations are used to give the complex reconfiguration operations corresponding to the running example.

As explained in Section 1, the instanciation of the general model cannot be checked with the proof approach. Indeed, we don’t consider any specification of the controller to manage the dynamic reconfigurations. Nevertheless, to partially validate the model, we apply bounded model-checking to validate the instantiated model.

5.1 Instantiating a Running Architecture

To instantiate a specific architecture using the generic B model, we just give values to all the previously defined sets, constants and variables in Subsect. 4.1, to make them represent the running architecture.

Let us now specify the different manipulated basic architectural elements corresponding to the HTTP server example:

<table>
<thead>
<tr>
<th>COMPONENTS</th>
<th>INTERFACES</th>
<th>INTERFACE_TYPE</th>
<th>PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>{HttpServer, RequestReceiver, RequestHandler, CacheHandler, RequestDispatcher, FileServer1, FileServer2}</td>
<td>{httpRequest, request, getHandler, handler, getDispatcher, getCache, cache, dispatcher, getServer, server1, server2}</td>
<td>{Trequest, Thandler, Tdispatcher, Tcache, Tserver}</td>
<td>{deviation, load, validityDuration, mem, ySize}</td>
</tr>
</tbody>
</table>

The PROPERTIES clause is extended to state values of the architectural relations:

\[
\begin{align*}
\land \text{ProvidedInterfaces} &= \{ \text{httpRequest }, \text{request }, \text{handler }, \text{cache }, \text{dispatcher }, \text{server1 }, \text{server2 } \} \\
\land \text{RequiredInterfaces} &= \{ \text{getHandler }, \text{getDispatcher }, \text{getCache }, \text{getServer } \} \\
\land \text{Contingency} &= \{ \text{getHandler } \rightarrow \text{mandat } \lor \text{getDispatcher } \rightarrow \text{mandat } \lor \text{getCache } \rightarrow \text{optional }, \text{getServer } \rightarrow \text{mandat } \lor \text{y } \} \\
\land \text{InterfaceType} &= \{ \text{httpRequest } \rightarrow \text{Trequest }, \text{request } \rightarrow \text{Trequest }, \text{handler } \rightarrow \text{Thandler }, \text{getHandler } \rightarrow \text{Thandler }, \text{getDispatcher } \rightarrow \text{Tdispatcher }, \text{getCache } \rightarrow \text{Tcache }, \text{cache } \rightarrow \text{Tcache }, \text{dispatcher } \rightarrow \text{Tdispatcher }, \text{getServer } \rightarrow \text{Tserver }, \text{server1 } \rightarrow \text{Tserver }, \text{server2 } \rightarrow \text{Tserver } \} \\
\land \text{Provider} &= \{ \text{httpRequest } \rightarrow \text{HttpServer }, \text{request } \rightarrow \text{RequestReceiver }, \text{handler } \rightarrow \text{RequestHandler }, \text{cache } \rightarrow \text{CacheHandler }, \text{server1 } \rightarrow \text{FileServer1 }, \text{dispatcher } \rightarrow \text{RequestDispatcher }, \text{server2 } \rightarrow \text{FileServer2 } \} \\
\land \text{Requirer} &= \{ \text{getHandler } \rightarrow \text{RequestReceiver }, \text{getDispatcher } \rightarrow \text{RequestHandler }, \text{getCache } \rightarrow \text{RequestHandler }, \text{getServer } \rightarrow \text{RequestDispatcher } \} \\
\land \text{Definer} &= \{ \text{deviation } \rightarrow \text{RequestHandler }, \text{load } \rightarrow \text{RequestHandler }, \text{validityDuration } \rightarrow \text{CacheHandler }, \text{mem } \lor \text{ySize } \rightarrow \text{CacheHandler } \}
\end{align*}
\]
Finally, we initialise the remaining architectural relations as follows:

\[\begin{align*}
\land \text{InitComponents} &= \{ \text{HttpServer}, \text{RequestReceiver}, \text{RequestHandler}, \text{FileServer1}, \\
& \quad \text{RequestDispatcher} \} \\
\land \text{InitParent} &= \{ \text{RequestReceiver} \mapsto \text{HttpServer}, \text{RequestHandler} \mapsto \text{HttpServer}, \\
& \quad \text{RequestDispatcher} \mapsto \text{HttpServer}, \text{FileServer1} \mapsto \text{HttpServer} \} \\
\land \text{InitBinding} &= \{ \text{handler} \mapsto \text{getHandler}, \text{dispatcher} \mapsto \text{getDispatcher}, \\
& \quad \text{server1} \mapsto \text{getServer} \} \\
\land \text{InitProvDelegate} &= \{ \text{request} \mapsto \text{httpRequest} \} \\
\land \text{InitReqDelegate} &= \emptyset \\
\land \text{InitState} &= \{ \text{HttpServer} \mapsto \text{started}, \text{RequestReceiver} \mapsto \text{started}, \\
& \quad \text{RequestHandler} \mapsto \text{started}, \text{RequestDispatcher} \mapsto \text{stopped}, \\
& \quad \text{FileServer1} \mapsto \text{stopped} \} \\
\land \text{InitValue} &= \{ \text{deviation} \mapsto 49, \text{load} \mapsto 75, \text{validityDuration} \mapsto 2, \text{mem} \lor \text{ySize} \mapsto 100 \}
\end{align*}\]

5.2 Modelling the Running Reconfigurations with B

After having specified the primitive reconfigurations in Subsect. 4.2, we can write more complex reconfiguration operations calling the primitive ones, by composing them sequentially and by using SELECT, IF and/or WHILE statements.

\begin{verbatim}
AddCacheHandler = SELECT (Value(deviation) > 50) THEN 
  instantiate (CacheHandler) ; 
  add(CacheHandler, HttpServer) ; 
  bind(cache, getCache) ; 
  start (CacheHandler) 
END ;
\end{verbatim}

The AddCacheHandler reconfiguration from Sec. 2 can be expressed using the B reconfiguration primitives as depicted here. All the reconfigurations of the running example can be expressed by B operations in a similar manner.

In addition, the normal running of different components could also change the architecture by modifying, for example, parameter values. To handle this behaviour, a solution would be to define an abstraction of the running of the system as a (set of) B operation(s). For our example, a B operation, called RUN, is added: basically, it changes the values of the (load) and (deviation) parameters of the RequestHandler component.

5.3 Validating the Running Architectural B Model

Addressing the validation of the instantiated B model using a proof process is not possible at this step of the work. Indeed, we have not taken into consideration a specification of the necessary associated controller to manage dynamic reconfigurations (by the mean of adaptation policies, as instance). Then, there is no information about the context where the reconfiguration operations are called. The AtelierB cannot be used to verify the instantiated model because its provers have no hypothesis to help them to prove that the reconfigurations preserve the architectural invariant.

Nevertheless, in order to (partially) validate the instantiated B model, ProB can be used to check the consistency of samples of the instantiated B model: at each reconfiguration step, ProB checks the architectural constraints.
to find an example violating the invariant. It is therefore possible to produce traces containing sequences of running and reconfiguration operations.

For example, it is easy to reproduce this short scenario (the right-side figure depicts the corresponding trace generated by ProB):

(i) The initial configuration of the HTTP server is without the CacheHandler and FileServer2 components;

(ii) Next configuration is obtained by running the architecture and changing the load and deviation values;

(iii) At this stage, we add CacheHandler to the global architecture, following the AddCacheHandler reconfiguration operation;

(iv) After that the FileServer2 component is added to the architecture, through the AddFileServer reconfiguration;

(v) The architecture is running;

(vi) By applying the RemoveCacheHandler reconfiguration, the component CacheHandler is deleted from the global architecture.

Automatic random explorations allow us to check the instantiated B model in a more general manner. As example, ProB model-checks 1000 nodes into 5301 milliseconds: neither invariant violation nor deadlock was found, and all the operations were covered.

6 Checking Temporal Formulae over Reconfigurations

In this section, we exploit the linear temporal logic for dynamic reconfigurations introduced in [12] and called FTPL. It allows us to characterise the correct behaviour of reconfiguration-based systems by using architectural invariants and linear temporal logic patterns. FTPL has been inspired by proposals in [13], and their temporal extensions for JML [21,8,14]. In this work, we propose to translate the FTPL patterns into LTL formulae in order to check FTPL properties with the ProB model-checker.

6.1 FTPL Syntax and Semantics

Let us consider the subset of FTPL in the figure below. It is based on trace properties, each of them being a temporal constraint on (a part of) the execution of the dynamic reconfiguration model. The configuration properties, called $conf$, are first order logic formulae over sets and relational operations
on the primitive sets and over relations defined in Sect. 4.1. Further, for a reconfiguration operation $ope$, its ending is considered as an event.

The trace properties specify the constraints to ensure on a sequence of reconfigurations. We mainly specify the always and eventually constraints which respectively describe that a property has to be satisfied by every configuration of the sequence, or by at least one configuration of the sequence.

Every temporal formula concerns a part of the execution trace on which the property should hold: it is specified with special keywords, like e.g., after/before a particular event has happened, or between two particular events.

Let us now illustrate the FTPL language by expressing some properties on the example of the HTTP server from Sect. 2.

**Example 1** The following property expresses an architectural constraint saying that always there is at least one file server. In other words, always there is at least one provided interface connected to the required interface $getServer$ of RequestDispatcher:

$$\text{always } \exists \text{iprov} \in \text{ProvidedInterfaces. Binding(iprov)} = \text{getServer}$$

**Example 2** The reconfiguration $AddCacheHandler$ (resp. $RemoveCacheHandler$) adds (resp. removes) $CacheHandler$ when the deviation value is greater (resp. less) than 50. The following property specifies that the deviation value eventually becomes less than 50 between the considered reconfigurations:

$$\text{between } AddCacheHandler \text{ terminates } RemoveCacheHandler \text{ terminates eventually } \text{deviation} < 50$$

These examples show that FTPL is more expressive than the proposals in [11], which only handle architectural invariants. Indeed, FTPL allows expressing event properties and temporal properties involving different kinds of temporal patterns which have been shown useful for practical applications [13].

### 6.2 From FTPL to LTL

We adapt the results in [21] and [8] and propose a translation of FTPL patterns into the LTL dialect considered by ProB, called $LTL^e$ [19]. The translation procedure, denoted $LTL(x)$, is inductively defined on the structure of the FTPL formula.
Let \( \text{conf} \) be a configuration property, \( \text{B}(\text{conf}) \) a rewriting procedure giving the \( \text{B} \) predicate corresponding to \( \text{conf} \), and \( \text{ope} \) a reconfiguration.

Let \( \text{trace} \), \( \text{trace}_1 \) and \( \text{trace}_2 \) be trace properties, \( \text{event} \) an event property, and \( \text{temp} \) a temporal property. Remark that a trace property is translated into LTL according to the temporal context in which the property is used, that is why we define an auxiliary functions \( \text{LTL}_B \).

| \( \text{LTL}(\text{after event temp}) \) | \( \text{G}(\text{LTL}(\text{event}) \Rightarrow \text{LTL}(\text{temp})) \) |
| \( \text{LTL}(\text{after event trace}) \) | \( \text{G}(\text{LTL}(\text{event}) \Rightarrow \text{LTL}(\text{trace})) \) |
| \( \text{LTL}(\text{before event trace}) \) | \( \text{F}(\text{LTL}(\text{event}) \Rightarrow \text{LTL}_B(\text{event, trace})) \) |
| \( \text{LTL}(\text{trace until event}) \) | \( \text{F}(\text{LTL}(\text{event}) \land \text{LTL}_B(\text{event, trace})) \) |
| \( \text{LTL}_B(\text{event, always conf}) \) | \( \text{LTL}(\text{conf}) \lor \text{LTL}(\text{event}) \) |
| \( \text{LTL}_B(\text{event, eventually conf}) \) | \( \neg(\neg(\text{LTL}(\text{conf}) \lor \text{LTL}(\text{event}))) \) |
| \( \text{LTL}(\text{between event}_1 \text{ event}_2 \text{ trace}) \) | \( \text{LTL}(\text{after event}_1 \text{ (trace until event}_2)) \) |

The FTPL property presented in Example 1 has been translated into the LTL formula below. This formula has been partially checked with ProB in 126 milliseconds. The model checker generates 2002 atoms and 16064 transitions when the maximum number of new states is 1000.

\[
\text{G( } \{ \exists(\text{iprov} \ldots (\text{iprov} \in \text{ProvidedInterfaces} \land \text{Binding}(\text{iprov}) = \text{getServer} ) \} \text{ )}
\]

Applying the above translation to the property in Example 2 results in the LTL property below, checked in 1802 milliseconds. The model checker generates 16048 atoms and 129704 transitions when the maximum number of new states is 1000.

\[
\text{G( } [\text{AddCacheHandler}] \Rightarrow \neg(\neg(\{\text{Value}(\text{deviation}) < 50\}) \lor \text{RemoveCacheHandler}) \text{ )}
\]

More sophisticated temporal properties involving architectural constraints can be written thanks to FTPL [12]. Then, thanks to our translation procedure, their verification can be investigated with ProB. Note that this verification is size-bounded and partial because of ProB features.

7 Conclusion

The different proposals presented in this paper concern the verification of dynamic reconfigurations of concurrent component-based systems. Dynamic architectural constraints could be expressed with a linear time temporal logic over (re)configuration sequences, as FTPL [12]. As architectural constraints involve first-order formulae, and a behavioural semantics of reconfigurations gives rise to infinite state systems, we have proposed an approach combining proof and model-checking to support the modelling of such systems and the validation of their dynamic reconfigurations, as depicted Fig. 1. We first have
proposed a generic B model for component architectures and we have proved the consistency of the model architectural constraints. Then, we have instantiated the general model to address a particular architecture validation: its consistency and some temporal properties over (re-)configuration sequences—expressed in FTPL and translated into LTL—have been model-checked.

Our contributions for temporal properties specification and verification—including static and dynamic analysis—allow monitoring instrumentation and managing applications at runtime, thanks to available tools to animate specifications.

**Related work.** In the context of dynamic reconfigurations, ArchJava [4] gives means to reconfigure Java architectures, and the ArchJava language guarantees communication integrity at run-time. Barringer et al. give a temporal logic based framework to reason about the evolution of systems [6]. In [5], a temporal logic is proposed to specify and verify properties on graph transformation systems.

In the Fractal-based framework, the work in [17] has defined integrity constraints on a graph-based representation of Fractal, to specify the reliability of component-based systems. Unlike [17], our model lays down only general architectural constraints, thus providing an operational semantics to other component-based systems. On the integrity constraints side, the FTPL logic allows specifying architectural constraints more complex than architectural invariants in [11].

To enforce software robustness while adding adaptive behaviour, the work in [20] proposes a formal framework for the Fractal component model, named FracL. Like our B-based proposal, the FracL static approach allows verifying the consistency of the application architecture. However, our proposal allows checking the model consistency and monitoring temporal properties, both fully automatically.

Among other applications, our proposals aim at a monitoring of component-based systems. In [7], Basin et.al have shown the feasibility of monitoring temporal safety properties (and, more recently, security properties) using a runtime monitoring approach for metric First-order temporal logic (MFOTL). Like the model in [7], our model is a first-order structure, but instead of considering a sequence of time stamps, we focus on reconfiguration operations. In [15], knowledge-based controllability is studied for constructing distributed controllers. The problem there is somewhat different than ours: the goal is to make the system behave exactly according to a given knowledge-based priority property, while here the reconfigurations must satisfy some given architectural and temporal constraints.
References


Reputation-based reliability prediction of service compositions

Galina Besova\textsuperscript{1}, Heike Wehrheim\textsuperscript{2}

\textit{Department of Computer Science}
\textit{University of Paderborn}
\textit{Germany}

Annika Wagner\textsuperscript{3}

\textit{Department of Computer Science}
\textit{Fulda University of Applied Sciences}
\textit{Germany}

\textbf{Abstract}

Today, the concept of service oriented architectures provides a way of building integrated solutions out of existing services. To this end, services from different providers are composed using advanced orchestration and choreography techniques. However, while this principle allows for greater flexibility at a smaller cost, the use of third party services also includes a risk: Deployed services might not work as claimed by their providers. In this paper, we propose a technique for analyzing the expected reliability of service compositions based on ratings given by (previous) service users. Every service thereby comes with a reputation, and the analysis computes an overall reliability of a service composition from the reputations of its constituent services. The proposed model-driven approach proceeds by translating statechart models of service compositions into input for a probabilistic model checker (PRISM) using state-of-the-art model transformations techniques. The approach has been implemented as an Eclipse plug-in and is fully compliant with UML.

\textit{Keywords:} Reliability prediction, service oriented architectures, probabilistic model checking, model transformations.

\section{Introduction}

Today, complex systems can be built by composing services to deliver integrated solutions, allowing greater flexibility, reuse of existing functionality,
scalability, etc. Alike component-based systems, services can be obtained from different software providers by searching existing repositories for the required functionality. However, the use of third-party services entails a risk: Most often the time scale for building a new system does not allow for an extensive testing of an external service. A software designer thus has to rely on the provider’s specification of the service functionality. A deviation of the actual behavior of the service from its specification thus only becomes apparent during execution of the constructed software.

In this paper, we propose an alternative to time-consuming testing or expensive formal analysis which is based on the use of a software provider’s or service’s reputation. The reputation can (for instance) be obtained by ratings of users of a service, like ratings for hotels and restaurants, or alternatively, by monitoring the service every time it is executed and recording whether it behaves as specified. The reputation is thus based on previous experience of users with the correctness of the service. Depending on the reputation of single services, the overall expected reliability of a service composition can be computed. Reliability herein is the probability of failure-free, correct operation of a service composition (“continuity of correct service” [9]). We take the reputation of a software provider as being an indication for the reliability of its services. The overall expected reliability of a service composition is then not just the average or sum of the reputations of its constituent services. Depending on the single service’s effect on the overall behavior, a service with a bad reputation might or might not have a large influence on the reliability of the complete composition. In this paper we propose a technique for systematically computing the reliability of a service composition based on its model annotated with reputations of single services.

In our approach we follow a principle employed by a large number of techniques for the analysis of non-functional properties of component-based systems. In particular for performance analysis, a large body of work employing model-driven approaches has been developed in recent years [22,10,14]. Models of component-based systems are enhanced with information about performance attributes of single entities, and these enhanced models are afterwards translated into various sorts of analysis models (e.g. stochastic Petri nets, Markov chains, queuing networks). The actual performance analysis is then carried out using standard tools operating on such analysis models. In our setting, single services will be modeled using UML state machines [6]. The reputation of a single service is given as a numeric value in the range [0..1], describing the ratio of correct executions of the service. A service composition is then a choreography made up of reputation-annotated state machines. Such service composition models are translated into Markov decision processes, in which the probabilities of executing transitions are set according to the reputations and the type of events (send, receive or internal event). We generate
Markov decision processes (MDPs) in the form of an input to the probabilistic model checker PRISM [18]. PRISM is then used to query the MDP for the probability of not reaching error states, and the answer to this query is the overall expected reliability of the service composition. For the generation of MDPs as input to PRISM we follow state-of-the-art model transformation techniques: using the language ATL [1] we define metamodel based rules for transforming UML state machine models to PRISM models. The required state machine metamodel was taken directly from the UML, and the PRISM metamodel had to be created. The model transformation is part of a larger Eclipse-based tool which provides automation of our approach.

The rest of the paper is organized as follows: In section 2 we describe the modeling and analysis formalisms used throughout the paper. Section 3 describes the transformation between the design and analysis models described in these formalisms. The tool support of the proposed approach is discussed in section 4. Section 5 gives an overview of related work and section 6 concludes the paper and gives some directions for future work.

2 Concepts

2.1 Modeling of choreographies

A service choreography consists of communicating services that perform activities and coordinate with each other by means of message exchange. Three basic activities that can be performed by a service in a composition include sending and receiving of messages, and internal activities. Communication can have synchronous or asynchronous character depending on the particular composition. In our setting, communication between the services is synchronous - it is possible only if both communicating processes are able to execute the transitions with the sending and receiving events pair at the same time. This might require one of the communication parties to wait.


A model consists of one or more UML state machines that contain an arbitrary number of parallel composed statecharts representing services (separated by regions). Each statechart is described by its states and transitions, that can be triggered by an internal, receiving, or sending event which is indicated through labels $a$, $a?$, and $a!$ respectively. For modeling these events we use the

---

4 Asynchronous type can be modeled through an additional statechart that represents the communication channel.
specific kinds of events provided by the UML for inter-process communication: *ExecutionEvent*, *ReceiveSignalEvent*, and *SendSignalEvent*.

Figure 1 illustrates the above discussed with an example of two services - *supplier* and *buyer*. Each service is presented by a statechart in its own state machine and region. They communicate on two occasions: first, to pass an order from the *buyer* to the *supplier*, and later, to inform the *buyer* about the order status. Note that the diagram additionally contains the reputations associated with the individual services. This is realized by a UML profile keeping our choreography models UML compliant.

### 2.2 Reliability analysis model

The modeling notation chosen in this work and described above is fairly straightforward and familiar to most software architects. However, it is mostly not supported by existing formal analysis techniques including model checking. Therefore, in order to analyze a choreography using model checking, its model has to be first transformed into the corresponding analysis model in accepted format. The model checker used in this work is the Probabilistic Symbolic Model Checker (PRISM) [3], therefore, its modeling language has to be introduced first.

The PRISM modeling language is based on the Reactive Modules formalism [7]. It allows description of a system as a Discrete Time Markov Chain (DTMC), Continuous Time Markov Chain (CTMC), or Markov Decision Process (MDP) model.

Main elements of an analysis model are modules and variables. Modules contain finite range local variables and commands. A PRISM command has the following form:

\[
[a] \ g \rightarrow \ p_1:(upd_{11}) \& \ldots \& (upd_{1m_1}) + \ldots + p_n:(upd_{n1}) \& \ldots \& (upd_{nm_n});
\]

The command consists of two parts divided by the transition sign $\rightarrow$. The left hand side contains an action label $a$ within the square brackets, which is used for processes synchronization, followed by a transition guard $g$. When the...
guard $g$ is true, an action $a$, if not empty, forces simultaneous execution of all commands labeled with $a$ within the model. The command’s right hand side contains $n$ possible mutually excluding variable updates sets each of which is equipped with a probability $p_i$. Each set contains $m_i$ variable updates $upd_{ij}$ which take place simultaneously.

Before providing an example of a PRISM module, the setting when more than one command can be executed at the same time needs to be discussed. In this case, the choice of one of the alternatives depends on the PRISM model type. DTMC and CTMC models assign equal probabilities to all alternatives, whereas MDP models simulate non-deterministic choice. In this work we focus on MDP models for service compositions since, compared to DTMC and CTMC, they additionally provide a mechanism for describing the cases when the choice between alternative execution paths of a service is not determined by a fixed probability distribution. This allows expressing influence of external environment on the system, changes of parameters within such environment, etc..

```plaintext
module M
  x : [0..1] init 1;
y : [0..1] init 0; z : [0..1] init 0;

[] x=1 -> 0.2: (y'=1) & (x'=0) + 0.8: (z'=1) & (x'=0);
[a1] y=1 -> (x'=1) & (y'=0);
[a2] z=1 -> (x'=1) & (z'=0);
endmodule
```

Figure 2 provides an example of a PRISM module $M$ with three finite range variables $x$, $y$ and $z$, and commands containing updates. The internal command with the guard $x=1$ and without any synchronization takes place first, since the local variable $x$ is initially set to 1. This command has two alternative update sets: the first assigns variables $y$ to 1 and $x$ to 0, and the second assigns variable $z$ to 1 and $x$ to 0. The first set has the probability of 0.2 and the second - 0.8. As variable $x$ has been set to 0, the first command cannot be executed again. Instead, one of the commands with the guards $y=1$ or $z=1$, depending on the previously chosen update set, can now be executed. These commands synchronize with some module on action labels $a1$ or $a2$ and set variable $x$ to 1, so the cycle can be repeated.

Modules that represent different interacting processes within the model, can be composed together in a process-algebraic expression. This expression should feature each module exactly once, and contain CSP-based operators including: parallel composition with full or partial synchronization over shared actions, asynchronous parallel composition, and operators for hiding and re-
naming of actions within the module. An analysis model described in PRISM modeling language is later translated by the model checker into a Markov model.

The described language allows definition of analysis models which can be used to model check various system properties including its reliability. The question to be discussed next, is the transformation of a design model of a choreography (section 2.1) into a model in the PRISM language.

3 Transformation concept

3.1 Reputation interpretation

To be able to analyze a service composition described in a design model, it has to be transformed into the analysis model in the selected format. In order to describe this transformation, it is necessary to define element mappings and analyze transition probabilities within the design model.

In this work we determine transition probabilities based on the reputation data of individual services and transition types. Given a reputation of a service as a whole, it is possible to choose between various interpretations of this information with regards to the probabilities of individual transitions. In this work we use the following interpretation:

• **Internal actions** of a service are not observable for other communication parties and, therefore, do not directly influence its observed reliability. Hence, transitions due to internal actions are assumed to always occur, i.e., have probabilities of 1.

• **Message receipts** by a service are observed by other parties. Services are assumed to always accept messages, possibly discarding them later. Following this assumption, such transitions always occur, and also do not directly influence the observed reliability, i.e., have probabilities of 1.

• **Message sendings** by a service are observed by other communication parties, as they initiate communication. Their success probabilities influence the observed reliability of the containing service. Therefore, such transitions occur with the probability equal to the observed reputation of the containing services.

Additionally, it is assumed that a message sending failure causes service interruption with no possibility of repair.

For the example presented earlier in Figure 1 this interpretation would mean that all transitions are executed with the probability of 1 except of the confirmation and rejection sending, which are executed with the probability of 0.88. This interpretation combined with the knowledge of the design and analysis model formats can now be used to define the required transformation.
rules.

3.2 Transformation rules

The idea behind the proposed transformation rules is to create an analysis model, where each service is represented by its own PRISM module with the same name. These modules contain local variables needed to describe different states of their corresponding services, and commands to describe transitions between these states. A set of local variables of such a module always contains exactly one \textit{start} variable and variables for unique final states of the service. The \textit{start} variable initialized with 1 represents the start state of the service.

We use the following rules to transform a transition into one or more PRISM commands. Depending on the type of triggering event, one of the options in Figure 3 is chosen:

a) \[ \begin{array}{c}
\xymatrix{ s \ar[r]^{a} & t } \\
\end{array} \]

\[ [s] v_s = 1 \rightarrow (v'_1 = 1) \land (v'_s = 0); \]

b) \[ \begin{array}{c}
\xymatrix{ s \ar[r]^{a?} & t } \\
\end{array} \]

\[ [s] v_s = 1 \rightarrow (v'_1 = 1) \land (v'_s = 0); \]

c) \[ \begin{array}{c}
\xymatrix{ s \ar[r]^{a!} & t } \\
\end{array} \]

\[ (v'_{\text{reputation}} = 1) \land (v'_s = 0) + (1-\text{reputation}); (v'_{\text{net}} = 1) \land (v'_s = 0); \]

\[ [a] v'_{\text{senda}} = 1 \rightarrow (v'_1 = 1) \land (v'_{\text{senda}} = 0); \]

Fig. 3. Transformation concept for transitions with different triggering event types

Note that the definition of these rules is based on the interpretation of reputation for probabilities of individual transitions discussed in section 3.1.

In all three cases the transition is performed between the states \( s \) and \( t \). Translated into the PRISM command, the first fact is represented by the boolean expression \( v_s = 1 \) as a guard. The fact, that the service has left the state \( s \) and entered the state \( t \), is represented by two updates \( v'_s = 0 \) and \( v'_t = 1 \) respectively, where \( v_s \) and \( v_t \) denote local variables corresponding to the service states \( s \) and \( t \). Further event type specific details of the transformation rules can be combined under the corresponding sub items:

a) Transitions triggered by internal actions \( a \) are transformed into commands that contain mentioned updates \( v'_s = 0 \) and \( v'_t = 1 \) without any synchronization.

b) Transitions triggered by receiving events \( a? \) are transformed into commands similar to the case a) that are, however, labeled with synchronization actions \( a \). This is done to ensure that the receiving and sending (case c)) command pair, that represents communication between two services through a message \( a \), is only executed synchronously.

c) Transitions triggered by sending events \( a! \) are transformed into pairs of two subsequent commands. The first command represents two alternatives: a
message will be sent with the probability equal to the \textit{reliability} reputation of the containing service, or a failure with the complementary probability. If the first alternative is chosen, the variable update $v'_{\text{senda}} = 1$ is performed, representing the fact that the message will be sent. If the second alternative is chosen, the update $v'_{\text{fail}} = 1$ is performed, representing sending failure which makes further commands execution within the module impossible.

The second command is only executed if the first probabilistic choice indicates that the message $a$ will be sent. This command is labeled with the synchronization action $a$ to ensure its simultaneous execution with the receiving command $a?$ (case b)).

Figure 4 illustrates this transformation rule on an MDP fragment for transition $a!$. The fragment consists of three transitions: from state $s$ either to one of the added intermediate states $senda$ or $fail$, and from state $senda$ to state $t$. The first two transitions with the complementary probabilities \textit{reliability} and $1 - \textit{reliability}$ represent the two alternatives within the first command, whereas the last transition represents the second command and the actual sending of the message $a$ with probability of 1.

As the states within a design model are unnamed, we had to define a naming mechanism for the variables $v_s$, $v_t$, $v_{\text{senda}}$ and $v_{\text{fail}}$. The following naming conventions have been used:

- The variable $v_s$ is named depending on the location of the state $s$ within the statechart as follows:
  - \textit{start}, if $s$ is an initial state.
  - variable that represent successful completion of an incoming transition $(s', s)$, if $s$ is an intermediate state. Naming of such variables is discussed next in the context of variable $v_t$.

- The variable $v_t$ is named depending on the type of the state $t$ and of the transition $(s, t)$ with the triggering event $a$ as follows:
  - corresponding final state variable, if $t$ is a final state.
  - $a$, if $t$ is an intermediate state and $(s, t)$ is an internal or receiving transition.
  - $a_{\text{Sent}}$, if $t$ is an intermediate state and $(s, t)$ is a sending transition.

- The variable $v_{\text{senda}}$ is named $a$, where $a$ is the name of the triggering event of the original sending transition $a!$.

- The variable $v_{\text{fail}}$, unlike $v_{\text{senda}}$, is shared by all commands within a module.
and is, therefore, simply named \textit{fail}.

Note that the set of variables may contain a fail variable, if the reputation of the corresponding service is less than 100\%. To comply with the requirements of the PRISM language all variable names are extended by the name of the containing module.

Figure 5 illustrates the result of this transformation for the supplier and buyer design model example. It contains two modules - \textit{Buyer} and \textit{Supplier},

\begin{verbatim}
module Buyer
  // local variables
  startBuyer: [0..1] init 1; orderBuyer: [0..1] init 0;
  orderBuyerSent: [0..1] init 0; finishBuyer: [0..1] init 0;
  // order?
  [order] startBuyer=1 -> (orderBuyer'=1) & (startBuyer'=0);
  [order] orderBuyer=1 -> (orderBuyerSent'=1) & (orderBuyer'=0);
  // rejection?
  [rejection] orderBuyerSent=1 -> (finishBuyer'=1) & (orderBuyerSent'=0);
  // confirmation?
  [confirmation] orderBuyerSent=1 -> (finishBuyer'=1) & (orderBuyerSent'=0);
endmodule

module Supplier
  // local variables
  startSupplier: [0..1] init 1; failSupplier: [0..1] init 0; orderSupplier: [0..1] init 0;
  processingSupplier: [0..1] init 0; confirmationSupplier: [0..1] init 0;
  rejectionSupplier: [0..1] init 0; finishSupplier: [0..1] init 0;
  // order?
  [order] startSupplier=1 -> (orderSupplier'=1) & (startSupplier'=0);
  // processing
  [] orderSupplier=1 -> (processingSupplier'=1) & (orderSupplier'=0);
  // confirmation?
  [] processingSupplier=1 -> 0.88; (confirmationSupplier'=1) & (processingSupplier'=0) +
  (1 - 0.88); (failSupplier'=1) & (processingSupplier'=0);
  [confirmation] confirmationSupplier=1 -> (finishSupplier'=1) & (confirmationSupplier'=0);
  // rejection?
  [] processingSupplier=1 -> 0.88; (rejectionSupplier'=1) & (processingSupplier'=0) +
  (1 - 0.88); (failSupplier'=1) & (processingSupplier'=0);
  [rejection] rejectionSupplier=1 -> (finishSupplier'=1) & (rejectionSupplier'=0);
endmodule
\end{verbatim}

Fig. 5. Analysis model for the supplier and buyer example

described by their local variables and commands, and synchronized on shared actions: \textit{order}, \textit{confirmation}, and \textit{rejection}. Module \textit{Buyer} has variables \texttt{startBuyer}, \texttt{orderBuyer} and \texttt{orderBuyerSent} for the commands representing order sending transition, and a variable \texttt{finishBuyer} for the final state\textsuperscript{5}. Module \textit{Supplier} apart from start variable \texttt{startSupplier} contains a \texttt{failSupplier} variable, as its reputation is less than 100\%. This module also has an \texttt{orderSupplier}

\textsuperscript{5} When the final state has no name, the corresponding variable is named \textit{finish}. 

49
variable for the order receiving command, and a processingSupplier variable for the processing command. Two variables confirmationSupplier and rejectionSupplier are added for the confirmation and rejection sending commands.

The commands within both modules can be derived by application of the discussed transformation rules to transitions, and usage of appropriate variables and synchronization actions. The resulting model can now be analyzed in PRISM to check various properties of the composition. These properties have to be formalized in the PRISM properties specification language. In this work, in order to obtain the expected reliability, we expressed it as a probability of not reaching a failure state during the lifetime of the system or formally, for an MDP system model containing $k$ failure states:

$$P_{\min} = \square \neg (\mathbf{F} (\text{fail}_1 = 1 || \ldots || \text{fail}_k = 1))$$
$$P_{\max} = \square \neg (\mathbf{F} (\text{fail}_1 = 1 || \ldots || \text{fail}_k = 1))$$

The value of these properties for the above example is equal to 0.88.

The proposed rules enable step-by-step creation of comprehensible analysis models of service compositions for further analysis with the PRISM model checker. However, to facilitate application of the proposed reliability prediction approach, it is necessary to provide required tool support. Therefore, the tool support, which development was carried out as part of this work, will be discussed next.

4 Tool support

As already mentioned, the reliability prediction approach proposed in this work contains several steps, which are summarized in Figure 6.

First, a service composition is modeled as described in section 2.1, and annotated with reputations. Then this design model is transformed into the analysis model via application of the transformation rules informally explained in section 3.2. Finally, the resulting model together with the model-specific reliability property specification is analyzed using the PRISM model checker. The result of such an analysis provides the reliability value of the service composition described in the initial design model.

Tool support for some of these steps like UML modeling and model checking with PRISM already exist. Other steps like model annotation and transformation required development of appropriate supporting mechanisms.

The Eclipse platform with its flexible plug-in based architecture and numerous useful third-party plug-ins has been chosen as a development and application platform for our approach. This choice allows the reuse of already existing UML2 conform modeling tools (e.g. UML2 Tools [5], TOPCASED [4]) realized as Eclipse plug-ins, to support the design model definition. The
following steps of our approach are not directly supported in Eclipse, however, various plug-ins significantly simplified the development process of our transformation tool.

First of all, we rely on the Eclipse Modeling Framework (EMF) for storing and retrieving our models. Moreover, the ATL transformation language [1](supported by a third-party plug-in) was used to define and apply the model transformation rules. Finally, we used the JET-template model-to-text engine [2] which allowed us to generate a textual representation of the transformed model. The PRISM model checker is, unfortunately, not integrated within the Eclipse platform, therefore, the analysis model produced by our tool has to be imported manually. The last two steps of the model-to-model transformation and the model-to-text transformation have been integrated in our tool.

Figure 7 illustrates the transformation principle and the artifacts needed for its implementation. It demonstrates, that the definition of the ATL transformation rules and JET-templates required UML and PRISM language metamodels. The latter was also developed in this work.

The last point to be mentioned is the annotation of design models. Our approach takes advantage of an existing lightweight UML extension mechanism by means of profiles. To enable annotation of UML statecharts with reputations, we defined a UML profile that contains a stereotype for state machine regions allowing these regions to carry a so-called tagged value storing a reliability value. As regions are used to separate services within one state machine, each region requires a reputation value.
5 Related work

Prediction of reliability has always attracted research interest. Numerous approaches have been proposed to address the growing complexity of component-based [28,15,23,16,24,25] and service-oriented systems [17,13,29,31,8,14,26,32,30,27,12].

Most of these approaches rely on some model of a system architecture expressed in a specification language like UML ([14], our method), BPEL ([31,8]), WSCI ([27]), etc. In the case of orchestrated services workflow diagrams (e.g. UML activity diagrams [14]) are generally used. Service choreographies, on the other hand, are modeled through the specification of communication between the parties (e.g. WSCI specification [27], communicating state machines in our case). The further choice of a concrete specification language depends on the desired level of abstraction. Additionally, most approaches require information on reliability of individual services/components. In some methods actual reliabilities are required, whereas other approaches including our rely on reputations.

Depending on the technique provided system models are either directly analyzed using reduction rules to compute QoS [13,20,30], or transformed into some kind of stochastic model [29,31,8,14,26,27,12] for further analysis, like in our case. The most widely used stochastic models for this purpose include Markov models and stochastic Petri nets with corresponding analysis algorithms. For instance, Zhong and Qi [31] consider BPEL specifications and transform them into stochastic Petri nets for analysis. Gallotti et al. [14] consider UML activity diagrams of an orchestration extended with QoS properties, and transform them into Markov models. Our approach is similar to [14] as it also uses Markov model and PRISM model checker for reliability analysis, however, it is focused on service choreographies, and, therefore,
considers different UML diagrams used for communicating services.

Xia et al. [27] propose the only other approach, that we are aware of except of our, that considers choreographies. It is based on WSCI specifications of compositions, which are translated into a General Stochastic Petri net for reliability prediction. Compared to our approach, the authors focus on models at a different abstraction level and apply other analysis techniques. This allows complementary usage of both methods at different system development stages.

Some authors [30,24] also address the problem of obtaining required reliability information for individual services/components. Zheng and Lyu [30] propose collaborative mechanism for predicting reliability of a service for a user based on the data collected from similar users, who have used this service. According to the authors, this mechanism demonstrates better reliability prediction accuracy than other approaches, however, it can only be applied if the failure data of services is available. This requires the service to be implemented and deployed. Roshandel, et al. [24] propose an architecture-based mechanism for reliability prediction of a component using Markov models, which makes it similar to some system-level approaches and does not require a component to be implemented. This is achieved due to the Hidden Markov models used to address the lack of an operational profile. Our approach, like other mentioned techniques except of [30,25], assumes service reputations to be supplied by some reputation provider. Such a provider could be based on these mechanisms.

Later, in [25] the authors extend [24] to estimate system reliability. This approach is based on a system model very similar to ours. It describes communicating components as a set of concurrent state machine containing interaction protocols of components. This model is transformed into a Dynamic Bayesian Network that includes reliabilities of individual components. Compared to [25], our approach does not associate service reliability with its probability of start, instead sending transitions are assumed to carry reliability-relevant probabilities. Unlike [25], where component-based systems are considered, we assume that services have no failure dependencies.

With respect to existing approaches, our method represents a first attempt to develop methodology and tool support for predicting reliability of service choreographies at the early design stage based on formal methods.

6 Conclusion

In this paper we have proposed a technique for computing the expected reliability of service choreographies based on reputations of single services. The technique involved transforming metamodel instances of UML state machines into Markov decision processes in the form of an input to PRISM. The probabilistic model checker PRISM could then be used to determine the expected
reliability. The approach has been implemented on the basis of state-of-the-art model transformation techniques and is UML compliant.

In the future, we intend to investigate how different forms of information about the correctness of services, some obtained by monitoring, some by a formal analysis, can be combined for reliability prediction. Furthermore, we will evaluate our approach on realistic case studies. This will in particular show whether our interpretation of reputations as being probabilities of sending transitions is the right choice for choreographies, or whether other choices, possibly depending on the application domain, are valid as well.

References


ENT: A Generic Meta-Model for the Description of Component-Based Applications

Jaroslav Šnajberk and Přemek Brada

Department of Computer Science and Engineering
Faculty of Applied Sciences
University of West Bohemia
Pilsen, Czech Republic
{snajberk,brada}@kiv.zcu.cz

Abstract

Current best practice in modeling component-based applications is the use of UML extended by a profile. This solution provides a general and common approach of application description and allows to capture some details based on the concrete component model. It has however disadvantages due to the limitations of UML itself, like little scalability or lack of inherent model semantics. In this paper we propose a solution to overcome these limitations, in the form of a meta-model developed directly for the description of components and component-based applications. Its unique aspect is the use of faceted classification to introduce additional semantics and structuring to the derived models. We describe the features and advantages of this meta-model and illustrate its aspects on a model example of a simple OSGi application. At the end of paper we also propose the usage of this meta-model in visualization of component-based applications.

Keywords: meta-model, component, component model, component-based application, UML

1 Introduction

In component-based software development it is important to know for which component framework is the component-based application developed, because the design of single components and the whole application architecture depend on this knowledge. A component framework is an implementation of component model, which means that component model contains specification of how component looks, which types of components exist, how they communicate, how they behave, etc.

When the architect wants to describe a component-based application using a textual or visual representation, there are two options.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
(i) General “boxes-and-lines” description (model) of the application. Such model can’t describe all aspects of the application as it has to be sufficiently abstract to be general, on the other hand it can be used for any component-based application and any component model.

(ii) Component model-specific description of the application, which is bound to one concrete component model. Such description can express all the aspects of the application as it is developed for the concrete component model but it can’t be used for applications in any other component model.

The component model-specific description is useful only for the exchange of information between the domain experts of the given component model. The specifics of the particular component model make it difficult to read and understand the architecture of application for experts from different domains. Moreover component models often use their own graphic notation (e.g. SaveCCM [4]), making it hard to read for experts not familiar with it.

A general description of application is on the other hand useful for exchange of information between domains but it provides no details bound to the specifics of concrete component model(s) and thus can only provide shallow understanding of the component-based application. The best example of these general description languages is UML 2.0 [7] and its underlying component model. There are therefore component constructs that are very difficult to model in UML, for example events, which can be important even on component diagram level to show the indirect connections between components.

The UML 2.0 component model also has only three groups of elements - attributes, operations, imported/exported interfaces, though stereotypes doesn’t have to provide sufficient grouping. Using an extended UML with all stereotypes has negative impact on the complexity and clarity of the final diagrams. UML also suffers from insufficient content awareness which makes it difficult to interpret model data. These deficiencies are also present in UML exported into the XMI format used for the exchange of model information between machines.

The problem therefore is the lack of an approach which would be:

(i) General, so it can describe applications of any component model in a similar fashion.

(ii) Flexible, to provide user implementation details and specifics of concrete component model.

(iii) Rich, with eligible groupings to provide more precise classification of elements than just attributes, operations, and imported/exported interfaces.

(iv) Content aware, so machines can understand the information exchanged.

(v) Customizable, so users can view or analyze only information they are interested in.
Usable by various roles in the software development process (architect, application assembler, etc.) so they can understand and modify the relevant parts of a model.

1.1 Goal of the Paper

We believe that development of a domain-specific alternative is a better solution than efforts to extend UML for modeling component-based applications. The advantages for developing a new meta-model are:

• Freedom from legacy models inadequate for current and future needs.
• Developed directly for the problem domain, thus clearer and easier use.
• Visual representation can use advanced features brought by the meta-model.

Potentially problematic on the other hand are the following aspects:

• Divergence from UML which is a standard language for most software engineers may decrease understanding and acceptance.
• Since component models are very diverse, a common meta-model may not sufficiently capture all their idiosyncrasies.

In this paper we therefore present a new meta-model called ENT designed directly for the description of component-based application. It aims to conform to all the requirements listed above while trying to avoid the problematic aspects. While there is not enough place to provide its full description together with a complete case study, an example application is used to show how its parts can be described in ENT.

The following section surveys the related work in the field of component meta-models. Section 3 provides an overview of our model, sections 4–6 describe in detail its formal structure. The Conclusion contains also a brief status of current implementation efforts related to the ENT meta-model.

2 Related work

The area of software composition described in [12] has several common features with our work. The author of this paper is concerned with the creation of a generic tool capable of handling different component models used for software composition. The author also uses generic approach and takes advantage of description of component models to achieve the goal of her work.

UML [7] was designed to be a universal and general modeling notation backed by a meta-model, but it also supports extensions that makes it more usable when user need to add some details to the model. UML 2.0 supports extensions through UML profiles, which uses stereotypes, tagged values and constraints. Using UML 2.0 and extending its meta-model through profiles is
best practice for description of component-based applications. These efforts are well illustrated on earlier mentioned SaveCCM, which already has its UML profile [10].

Extending UML through profiles is not the only way how it can be extended. It is also possible to extend the core meta-model of UML as described in [9]. The author used this “heavyweight” approach to provide better description of the C3 architectural style described in [15]. However this approach does not meet our requirements on customizability and content-awareness.

There already are research works covering the area of component model description. For instance, Crnkovic et al [3] describe advanced framework able to classify any component model from various angles. On the other hand, Medvidovic [6] uses ADL for description purposes. While these articles capture a lot of experience, their aim is only to describe the features of component models without developing a meta-model based on the results.

In [14] the author describes the need for component meta-model capable of modeling the various existing component models to unify components into modeling paradigm. This work however doesn’t consider modeling of component-based applications.

3 ENT meta-model

The ENT meta-model is a general model defining the structures of component models and component-based applications; see [1] for a previous version of the model. Its distinguishing characteristics is the use of the faceted classification approach [13] to represent components in a way which is flexible enough for users with different interest. A key structure used in the meta-model is the ENT classifier, which is a tuple of identifiers which characterise any component interface element from several orthogonal aspects related to user perception.

The ENT meta-model is structured into two levels: on the component model level the main characteristic features of a given component model are defined, on the application level the concrete components, their interface elements and their bindings in an application are captured.

3.1 Overview of the Meta-Model

Let us start with a brief overview of the meta-model in plain English; the following subsections will then provide the exact definitions. The structural hierarchy of the meta-model starts with a component model as a set of component types. A component type is defined by a complete minimal set of definitions of traits which describe the possible kinds of interface elements which the component type can support. The traits declare the language meta-type and ENT classifier of these elements, capturing their commonalities like the
As an example, there is only one component type in OSGi called “bundle”, with ENT definition described in section 4.1. The ENT meta-model enforces this structuring of component interface (as opposed to a flat collection of items, cf. Figure 7) because it is quite natural for developers to think of e.g. all component’s provided services as a group, regardless of their concrete interface types and location in the specification source. In Enterprise JavaBeans on the other hand several different component types can be identified – SessionBeans, MessageDrivenBeans or Entities. The component types, as well as trait’s characteristic meta-type and classifier, are therefore based on a human analysis of the concrete component model and its component specification language(s).

At the level of a concrete application, a component implementation then conforms to one of the component types defined by its component model. Each component has a set of concrete interface elements manifest on the visible surface of its black box. These elements populate some or all of its actual traits, which again conform to the corresponding trait definitions. The component also holds the connections of its elements to the counterpart elements in client and/or supplier components, and – in case of hierarchical component models – may list the sub-components it is composed from.

In many component models, several run-time instances of a concrete component can be created, each with unique identity. The ENT meta-model does not deal with component instances because its domain is the level of component models and component application design, rather than the run-time instantiation level.

The rest of this paper provides a formal definition of these structures, in a top-down fashion.

3.2 Classification System

The ENT meta-model uses a faceted classification system for characterising various aspects of component interface elements, with eight facets called “dimensions”. These dimensions have predefined values and each dimension represents a different point of view on a component.

Definition 3.1 The ENT classification system is a collection of facets $Dimensions_{ENT} = \{dim_i, i = 1..8\}$ where the $dim_i$ are:

- Nature = \{syntax, semantics, extra-functional\}
- Kind = \{operational, data\}
- Role = \{provided, required, neutral\}
- Granularity = \{item, structure, compound\}
- Construct = \{constant, instance, type\}
• Presence = \{mandatory, permanent, optional\}
• Arity = \{single, multiple\}
• Lifecycle = \{development, assembly, deployment, setup, runtime\}

The **ENT classifier** is a tuple \(K = (k_1, k_2, ..., k_D)\) where \(k_i \subseteq \text{dim}_i, \text{dim}_i \in Dimensions_{\text{ENT}}, D = |Dimensions_{\text{ENT}}|\).

This classification system and the classifier structure are used in the trait and category set definitions, presented in the subsequent paragraphs.

### 4 The Component Model Level

Identification of different component models and the types of components they define forms the top level of the meta-model.

**Definition 4.1** A component model is the pair \(M = (\text{name}, \mathcal{C}_S)\) where \(\text{name} \in \text{Identifiers}\) is the model’s name and \(\mathcal{C}_S = \{\mathcal{C}_{i,\text{def}}\}\) is a set of component type definitions.

Component types consist mainly of trait definitions that declare the kinds of elements (features) the concrete components can have on their surface. Traits thus helps to fully characterize components of such type. For example, OSGi components (cf. Section 4.1.2) have traits Export packages, Provided services, Import packages, etc.

**Definition 4.2** A component type is a tuple \(C_{\text{def}} = (\text{name}, \text{tagset}, T)\) where \(\text{name} \in \text{Identifiers}\) is the name of the component type, \(\text{tagset} = \{\text{tag}_i\}\) is a finite set of extra type information items (“tags”), and the \(T = \{T_{i,\text{def}}\}\) where \(i\) is a finite index is the set of the component type’s trait definitions (also called “trait set”).

The tags in the tagset are triples \(\text{tag}_i = (\text{name}_i, \text{valset}_i, d_i)\) where \(\text{name}_i \in \text{Identifiers}, \text{valset}_i\) is the set of its possible values, and \(d_i \in \text{valset}_i \cup \{\epsilon\}\) is the default value (\(\epsilon\) means “no default”). Tags capture pieces of information that are important for the component model and cannot be described using traits, e.g. component’s persistence and transactionality as used in Enterprise JavaBeans.

The component types of one component model must be distinct: \(\forall C_i, C_j \in M.C_S, i \neq j : C_i \neq C_j \Rightarrow C_i.\text{name} \neq C_j.\text{name}\).

**Definition 4.3** A trait definition is a tuple \(T_{\text{def}} = (\text{name}, \text{metatype}, K, \text{tagset}, \text{extent})\) where \(\text{name} \in \text{Identifiers}\) is the trait’s name, \(\text{metatype} \in \text{Identifiers}\) is the meta-type of the component interface elements grouped by this trait, \(K\) is their ENT classifier, \(\text{tagset} = \{\text{tag}_i\}\) is the finite set of allowed tags of these elements, and \(\text{extent} \in \{\text{one}, \text{many}\}\) defines the maximum
number of elements in the trait\(^1\).

Consistency rule: Traits of one component type must be distinguishable by name, i.e. \(\forall T_i^{\text{def}}, T_j^{\text{def}} \in C^{\text{def}}.T, i \neq j : T_i^{\text{def}}.\text{name} \neq T_j^{\text{def}}.\text{name}\).

The *metatype* of the trait’s elements (such as “interface” or “event”) may be related to or derived from the name of the corresponding non-terminal symbol in the grammar of the component’s interface specification language particular for the trait. The *tagset* has the same definition and meaning as that of the component, described above, except that the concrete tag values are meant to be assigned to individual elements (not to the trait).

The ENT classifier \(K\) describes the classification properties of the trait’s elements – this is a unique aspect and key concept of the ENT meta-model, capturing the human-perceived similarity of the elements grouped by a trait.

Concerning the consistency rule, it is actually preferred that traits are distinguished by their classifiers only, i.e. the following stronger assertion holds: \(\forall T_i^{\text{def}}, T_j^{\text{def}} \in C^{\text{def}}.T, i \neq j : T_i^{\text{def}} \neq T_j^{\text{def}} \Rightarrow T_i^{\text{def}}.\text{name} \neq T_j^{\text{def}}.\text{name}\). There may however be cases when the ENT classification scheme does not provide enough characteristics to reliably distinguish traits. Then, distinguishing by names is the only practical option and this is reflected in the definition.

When the component model level description is designed according to the ENT meta-model, a set of data structures for modeling component-based applications is prepared. These data structures can fully describe all components implemented in the given component model and have to be created manually after analysis of modeled component model. The following section illustrates the ENT component model definition for the OSGi framework.

### 4.1 Example: The OSGi Component Model and Application

To illustrate the ENT structures, this section presents a subset of the representation of the OSGi component model [8] plus examples of behavioural and extra-functional element traits. OSGi was chosen for its industrial relevance, simplicity and ubiquity.

#### 4.1.1 Component Types

OSGi has only one component type called **Bundle**. Bundle can have two additional tags originated in manifest file.

(i) **Bundle**

- **tagset**: symbolic name, version

\(^1\) For simplicity, we do not use concrete numbers, ranges and similar features in extent specification.
• T: { export_packages, import_packages, provided_services, required_services, native_code, require_bundles, required_execution_environment, use_packages }

4.1.2 Trait Definitions

For demonstration purposes we provide the definitions of just four traits here, see [16] for a complete analysis of OSGi ENT representation:

(i) export_packages
- metatype: package
- K: ({syntax}, {operational}, {provided}, {structure}, {type}, {permanent}, {multiple}, Lifecycle)
- tagset: version, parameters
- extent: many

(ii) import_packages
- metatype: package
- K: ({syntax}, {operational}, {required}, {structure}, {type}, {permanent}, {single}, Lifecycle)
- tagset: bundle_symbolic_name, bundle_version, kind, version_range
- extent: many

(iii) provided_services
- metatype: interface
- K: ({syntax}, {operational}, {provided}, {item}, {instance}, {optional}, {single}, Lifecycle)
- tagset: service_filter
- extent: many

(iv) required_services
- metatype: interface
- K: ({syntax}, {operational}, {required}, {item}, {instance}, {optional}, {multiple}, Lifecycle)
- tagset: service_filter, service arity
- extent: many

4.1.3 Behaviour and Extra-Functional Properties

Traits can also represent other than functional elements, for example a quality of service aspect (e.g. [5]) or the expected call sequence protocol [11]. These traits must have value semantics respectively extra-functional in the dimension Nature of the ENT Classification. Sample trait definition for such elements are provided below:

(i) response
- metatype: attribute
• $K$: (\{extra-functional\}, \{data\}, \{provided\}, \{item\}, \{constant\},
\{mandatory\}, \{single\}, \{runtime\})
• tagset: $\emptyset$
• extent: many

(ii) protocol
• metatype: regular-expression
• $K$: (\{extra-functional\}, \{operational\}, \{provided\}, \{structure\}, \{type\},
\{optional\}, \{single\}, \{assembly, runtime\})
• tagset: $\emptyset$
• extent: one

4.1.4 Example OSGi Application

In the subsequent sections we will refer to (parts of) a simple example OSGi application called Parking Lot. It consists of four components as illustrated in Figure 1, the architecture should be self-descriptive.

![Component application example — Parking Lot (OSGi application)](image)

5 Application Level

This level of the ENT meta-model provides modeling constructs for concrete components and applications built from them. The component model level has to be already defined because the application level references its elements. These references assign meaning to the application elements; in particular, the set of traits of a concrete component is gained by assigning it the corresponding component type.

Definition 5.1 A **component application** is a direct acyclic graph $A = (C, B, m)$ where $C = \{c_i, i \in \mathbb{N}\}$ are components, $B = \{b_i, i \in \mathbb{N}\}$ their
Snajberk and Brada

bindings, and $m \in C$ is a main component. We use the term application context for a set of all components $A^* = \{c_i, i \in \mathbb{N}\}, AC \subseteq A^*$ existing in the environment where the component application is deployed.

A consistent (resolved) application is such that has all non-optional required elements bound to provided ones within the given context and all its components’ inheritance parents exist in the context.

We do not model additional pieces of information associated with applications, like configuration properties, access control lists, and similar – these are used at run-time which is out of scope for ENT meta-model.

manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Gate
Bundle-SymbolicName: Gate
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Require-Bundle: Parkinglot;version="1.0.0"
Import-Package: cz.zcu.kiv.parkinglot.parkinglot;version="1.3.0",
org.osgi.service.event;version="1.2.0"
Export-Package: cz.zcu.kiv.parkinglot.gate

Fig. 2. Manifest file for Gate bundle

5.1 Individual Components

In this section an example of the Gate bundle (see Figure 1) will help to illustrate the representation of component information in the ENT meta-model structure. The manifest file of this bundle is present in Figure 2.

Definition 5.2 A concrete component is a tuple $c = (\text{name}, C^{\text{def}}, G, T, P, S)$ where name is the component’s name, $C^{\text{def}}$ is the (reference to) the appropriate component type, $G = \{(\text{name}_i, \text{value}_i)\}$ is the set of its tags, $T = \{t_i\}$ is the concrete trait set of the component with traits as defined below, $P$ is a finite, possibly empty set of (references to) concrete components which are $c$’s inheritance parents, and $S$ is a finite, possibly empty set of $c$’s sub-components and their delegation bindings (see subsection 5.3 below).

The following consistency rules must hold:

- $\forall (n_i, v_i) \in c.G \exists \text{tag}_j \in C^{\text{def}.\text{tagset}} : n_i = \text{tag}_j.\text{name} \land v_i \in \text{tag}_j.\text{valset}$, i.e. tags are taken from component’s type tagset;
- $\forall p \in P : p.C^{\text{def}} = c.C^{\text{def}}$, i.e. the parents are of the same component type.

It is also natural that both $c$ and all its sub-components belong to the same component model.

By component interface element set $E(c)$ we will understand the set of all specification elements (as defined below) contained in the specification of concrete component $c$. In case of component inheritance, it is the union
of element sets of the transitive closure of $c$ and all its inheritance parents. Subsets $E_P(c)$ and $E_R(c)$ of the element set denote the provided and required elements of $c$ where it holds that $E_P(c) \cap E_R(c) = \emptyset \land E_P(c) \cup E_R(c) = E(c)$.

This representation is a complete model of a concrete component, by which we mean that the original specification of the component can be fully reconstructed from the representation.

Concrete component’s trait is a named set of its interface elements with the same meaning, as given by their meta-type and ENT classifier.

**Definition 5.3** A component interface trait (of a concrete component $c$) is a pair $t = (T^{\text{def}}, E)$ where $T^{\text{def}}$ is a (reference to) the trait definition and $E \subseteq E(c)$ is a subset of component’s interface elements.

Consistency rules: It must hold for a given component $c$ that

- $E(c) = \bigcup_i t_i.E, t_i \in c.T \land \forall t_i, t_j \in c.T, t_i \neq t_j : t_i.E \cap t_j.E = \emptyset$, i.e. that the traits together contain all its elements without duplicates

- $\forall t \in c.T : t.T^{\text{def}} \in c.C^{\text{def}}.T$, i.e. traits are defined by its component type.

Traits group the interface elements of a component even if in the source these may be specified in various places – either within one specification file (e.g. a SOFA ADL, disregarding the particular ordering of declarations), or even in several ones (e.g. OSGi manifest plus declarative services’ component.xml).

Traits alone do not say anything about the features of the particular component – they have only grouping purpose and through the reference to their trait definitions give meaning to all interface elements contained in it.

\[
T^{\text{def}} = \text{imported\_packages},
\]
\[
E = \{\text{cz.zcu.kiv.parkinglot.parkinglot, org.osgi.service.event}\}
\]

Fig. 3. The imported\_packages trait of the Gate bundle in ENT representation

**Definition 5.4** An interface element $e$ of a concrete component $c$ with specification written in language $L$ is a tuple $e = (\text{name}, \text{type}, G)$ where $\text{name} \in \text{Identifiers} \cup \{\epsilon\}$ is the (possibly empty) element’s name, $\text{type} \in L$ is a language phrase denoting its type, and $G = \{(n, v)\} \subseteq \text{Identifiers} \times \text{Identifiers}$ is the (possibly empty) set of element’s concrete tags.

Consistency rule: $\forall e \in t.E, \forall g \in e.G \exists d \in t.T^{\text{def}}.\text{tagset} : g.n = d.\text{name} \land g.v \in d.\text{valset}$, i.e. the tag values of elements in trait $t$ must be taken from the value set in the trait definition.

A specification element is a complete representation of one component interface feature identified by language name and/or type. All its parts are directly related to its specification source code (the human classification and
understanding of an element is attached to its containing trait). Operations on them are therefore subject to the syntax and typing rules of the language $L$ used for the component interface specification.

The tags represent additional semantic or other extra-functional information pertaining to the particular element (not to its type), like the `readonly` or `final static` keywords. They are important if one needs to e.g. precisely compare two elements or re-generate a valid source code for the element. Note that the element’s tags are defined in its trait definition, since all elements of one trait necessarily have the same set of tags.

<table>
<thead>
<tr>
<th>name</th>
<th>cz.zcu.kiv.parkinglot.parkinglot,</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>package,</td>
</tr>
<tr>
<td>$G = {(version, 1.3.0)}$</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4. The `parkinglog` element of the `imported_packages` trait in ENT representation

5.2 Component Bindings

To model bindings between components within the application, we use a set of connections which keep information about source element, target element and which direction information flows (provided / required).

**Definition 5.5** Let us have a consistent component application $A$. The application connection set is a finite set $B = \{b_i, b \in \mathbb{N}\}$ where $b = (e^s, e^t)$:

$\exists c_i, c_j \in A.C : e^s \in E^R(c_i), e^t \in E^P(c_j)$ i.e. the connections (arcs in the application graph) lead from required to provided elements.

The connection set of a component $c$ is a set of connections which have incidence with the component: $B(c) \subseteq B, \forall b \in B(c)$ either $b.e^s \in E^R(c)$ or $b.e^t \in E^P(c)$.

The connection set of a component makes it possible for every component to be aware of all connections realized by its elements, both provided and required.

$e^s = \text{Gate::exported\_packages::cz.zcu.kiv.parkinglot.gate}$,
$e^t = \text{Desk::imported\_packages::cz.zcu.kiv.parkinglot.gate}$

Fig. 5. The service `cz.zcu.kiv.parkinglot.gate` bound to bundle `Desk` in ENT representation

5.3 Hierarchical Components

Some component models such as SOFA [2] use hierarchical decomposition which means that composite components can be recursively composed from other components. Components which are not composed from any other components are called primitive components.
For composite components, a special set of connections needs to be modeled: the subsumption and delegation bindings between the composite component interface elements and its sub-components.

**Definition 5.6** For a given component \( c \) in application \( A \), the pair \( S = (S^c, S^d) \) in component’s tuple captures the inner architecture of its composition. \( S^c \subseteq A.C, c \notin S^C \) is the set of sub-components. The \( S^d \) is a set of delegate/subsume binding pairs, \( S^d = \left\{ (e^c, e^s) \mid e^c \in E(c), e^s \in E(s) \cdot s \in S^c \right\} \), i.e. the \( e^c \) and \( e^s \) elements belong to the composite component and one of its sub-components, respectively.

Consistency rule (added to those in Definition 5.2): \( \forall (e^c, e^s) \in S^d : e^c \in c.t_m, e^s \in s.t_n, t_m.T_{def} = t_n.T_{def} \), i.e. elements in subsume/delegate pairs belong to traits with the same trait definition.

For example, suppose that the Parking-lot component from Figure 1 was in fact hierarchical. The handling of client’s requests on the IArriveDeparture element could be delegated to an equally-typed element in a Arrivals sub-component. This would be expressed as an inner architectural binding \( \text{Parking-lot::IArriveDeparture, Arrivals::IArriveDeparture} \). Both elements would belong to the “provided-services” trait of their components.

### 6 Structuring Level: Category sets

Some traits and elements could be at particular times considered as unwanted information when reading a model of component-based application. For example, software architects are interested in other information than programmers. By using all information contained in both layers of an ENT-based model there could also be a danger of confusion when representing big and complex applications.

After representing a component-based application according to the Application level, the ENT classifier allows us to organize the model information using so called category sets. These sets are defined by selector operators on the trait classification which say how to group and display traits.

**Definition 6.1** The category set over an ENT model is a pair \( \text{Catset} = (\text{name}, \{(c, K, f)\}) \) where \( \text{name}, c \in \text{Identifiers} \) are the names of the category set and its categories, and \( f = K \times T_{def} \rightarrow \text{boolean} \) is a function which determines whether the given trait definition fits the (partial) classifier \( K \).

For example, the E-N-T category set defined in Figure 6 has three groups. In the first group are elements that are contained in traits with role = \{provided\} in their classifier (this means those elements which the component exports). Required elements are similarly grouped as needs and elements that are both provided and required are called ties. This category set
Snajberk and Brada
gave the name to the ENT meta-model, as it captures the most fundamental split of any component’s interface element set.

<table>
<thead>
<tr>
<th>E-N-T (Exports-Needs-Ties)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E : K = {(role = {provided}}, f = matches$</td>
</tr>
<tr>
<td>$N : K = {(role = {required}}, f = matches$</td>
</tr>
<tr>
<td>$T : K = {(role = {provided, required}}, f = matches$</td>
</tr>
</tbody>
</table>

Fig. 6. The ENT category set

More category sets are presented in [1], and category sets can be created by any user of the ENT meta-model if another point of view is needed.

6.1 Visualization using the ENT meta-model

The ENT meta-model in general and the category sets in particular have a big use in visualization of components. Component can be visualized very similarly as in UML but the surface of component can be displayed in a tree structure governed by element grouping into traits and category sets. For example, there are three different views on the same OSGi bundle in Figure 7. The ENT and ITC category sets show all traits of bundle component type in different groupings, while the II category set is very selective and shows only imported instances.

The possibilities of grouping and filtering by applying a category set layer over an application model are very rich. Since category sets can be defined by user and switched between easily, this is one of the most useful features of the ENT meta-model itself. This approach ensures that user see only what he wants to consider at the moment.

It is important to note that model data provided by an ENT-based model can be used by a number of different visualization styles; the visualization presented in Figure 7 is only one of the many.

7 Conclusion

In this article we proposed a new meta-model, called ENT, for the description of components and whole component-based applications. This meta-model takes the advantage of the close relation between a component model and its real components. It provides structures for the description of component types supported by a component model and, more importantly, it groups component surface elements to so called traits that use a custom classification system to capture their human-perceived characteristics. The classification provides enough information to machine interpret the resulting component representations in different ways. The ENT meta-model also captures the relations between components and supports hierarchical decomposition of components.
The ENT meta-model addresses the requirements stated in the Introduction through the following properties:

(i) generality: verified support for a wide spectrum of component models [1] and application structures;

(ii) flexibility: traits are able to represent various component elements, tags model implementation details particular to a concrete component model;

(iii) richness: the combination of the trait structure and ENT classifier provides a wealth of information;

(iv) content awareness: thanks to the ENT Classification system, the character of traits can be described and machine interpreted;

(v) customization: category sets enable the users to filter and group component elements and their details based on actual needs;

(vi) role support: the information in ENT-based models can be filtered for concrete roles through the Lifecycle classification facet.

We have successfully created an implementation of the ENT meta-model using model driven development, with a XMI format of model definitions. A loader of OSGi bundles into the ENT-based model data structures is already implemented and loaders for other frameworks (EJB and SOFA) are the subject of implementation at the time of writing of this article.

The ENT meta-model is expected to be used in advanced component application visualizations and a corresponding implementation of a basic tool able to use the advantages of this meta-model is under way. In a longer term we would like to improve this visualization tool by conducting research on different kinds of visual representation and maximizing the possibilities of the ENT meta-model used as data layer.
ACKNOWLEDGEMENTS

The work was partially supported by the UWB grant SGS-2010-028 Advanced Computer and Information Systems.

References


Position Papers
Performance Certification of Software Components

Erik Burger ¹  Ralf Reussner ²

Software Design and Quality
Institute for Program Structures and Data Organization, Faculty of Informatics
Karlsruhe Institute of Technology, Germany

Abstract
Non-functional properties of software should be specified early in the development process. In a distributed process of software development, this means that quality requirements must be made explicit in the specification, and the developing party of a commissioned component needs to deliver not only the implemented component, but also a description of its non-functional properties. Based on these artefacts, a conformance check guarantees that the implemented component fulfills the performance requirements.

We extend the notion of model refinement to non-functional properties of software and propose a refinement calculus for conformance checking between abstract performance descriptions of components. The calculus is based on a refinement notion that covers the performance-relevant aspects of components. The approach is applied to the Palladio Component Model as a description language for performance properties of components.

1 Introduction

During the design of component-based systems, it is useful to model non-functional properties of a system, like performance, already in early stages of the development process. Developers often see quality of service as a property of software that is checked and corrected once the product is completed. This “fix-it-later” practice is, however, a reason for quality problems in software development. Just like testing is an integral part of the implementation process that should be integrated from the beginning, performance modelling enables

¹ This work is granted by the GlobaliSE project, a research contract of the state of Baden-Württemberg which is funded by the Landesstiftung Baden-Württemberg gGmbH.
² E-Mail: reussner@kit.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
the developer of a system to make design decision based on analyses and simulations.

Abstract performance models can, however, also be used to express performance requirements in the specification phase of component-based software development. As the development proceeds, additional performance models are created to describe the properties of the design, and eventually the implemented component. In order to prove that the performance requirements are met in all these stages, a notion of refinement for performance is needed. By using performance refinement in the development process, the developer can check at any time if the requirements are still met and which properties may be violated.

Even if the commissioned component is delivered without a performance specification, it can be reconstructed by reverse engineering methods such as static code analysis and analyses of monitored execution traces [8]. However, as such a reconstructed performance description can differ from a manually specified one, the refinement calculus still is needed to show the compliance.

Since the performance of a component is influenced by many factors, the refinement calculus should take this into account by offering several levels of refinement. In this paper, we propose a refinement calculus that is based on component properties like external call sequences and usage of resources. The aspects of this refinement method make use of formal methods like finite automata and the resource demand calculus presented in [4], which make it possible to prove valid performance refinement on an abstract level.

The contribution of this paper is firstly a model for parameterised component performance specifications and secondly a calculus of refinement. The proposed language for component performance specifications is based on the performance prediction model used in the Palladio Component Model [1], a metamodel for the description of component-based software architectures. The PCM has been used in several industrial case studies and offers methods for the prediction of quality of service attributes, especially performance and reliability, as well as tool support for modelling and prediction. We use the PCM as a description language for performance properties of components since it offers parametric dependencies between various aspects of a component-based system, like deployment, assembly and usage profile.

This paper is structured as follows: In Section 2, we give a brief introduction into the Palladio Component Model. In Section 3, the scenario for software performance certification and the refinement calculus are presented. The assumptions and limitations of the approach are discussed in Section 4. Related work is mentioned in Section 5 before the paper concludes with Section 6.
2 Foundations

The Palladio Component Model (PCM) [1] is a meta-model for the description of component-based software architectures. The model is designed with a special focus on the prediction of Quality-of-Service attributes, especially performance. Service Effect Specifications (SEFF) describe the relationship between provided and required services of a component. In particular, the PCM SEFFs are the first calculus which takes all influencing contextual factors of component performance into account explicitly.

In the PCM metamodel, they are defined in the form of Resource Demanding Service Effect Specifications (RDSEFF), which are used for performance prediction and contain a probabilistic abstraction of the control flow. RDSEFFs use a notation stemming from UML activity diagrams, i.e. activities are denoted by nodes. For each RDSEFF, a resource demand can be specified as well as dependencies of transition probabilities and resource demands on the formal parameters of the service. RDSEFFs can be annotated to each provided service of a component. They describe

- how the service uses hardware/software resources;
- how the service calls the component’s required services.

Resource demands in RDSEFFs abstractly specify the consumption of resources by the service’s algorithms, e.g., in terms of CPU units needed or bytes read or written to a hard disk. Resource demands as well as calls to required services are included in an abstract control flow specification, which captures call probabilities, sequences, branches, loops and forks.

RDSEFFs abstractly model the externally visible behaviour of a service with resource demands and calls to required services. They present a grey box view of the component, which is necessary for performance predictions, because black box specifications (e.g., interfaces with signatures) do not contain sufficient information. RDSEFFs are not white box component specifications, because they abstract from the service’s concrete algorithms and do not expose the component developer’s intellectual property. Component developers specify RDSEFFs during or after component development and thus enable performance predictions by third parties.

3 Certification of Software Component Specifications

3.1 Certification Scenario

The proposed refinement calculus can be applied in a scenario of certification described in [4]. This scenario is depicted in Figure 1. In the proposed component-based software development process, a specifications document for components is created and enriched by non-functional requirements concern-
ing the performance of a component (depicted as “Performance Requirements” on the left hand side). These requirements have to be expressed formally in the specifications document, using an abstract performance description language.

![Diagram](image)

Figure 1. Certification Scenario

The performance requirements serve as a contract which has to be fulfilled by the implementing party. However, performance descriptions can not only be used in the specification of a software system, but also to describe an actual implementation of this system.

Based on the specifications document, the implementation of the component is created, usually by a third party supplier. The resulting component is shipped with a description of its performance properties (depicted as “Performance Description” on the right hand side). This description can be determined by the developer in two ways: In the first case, the developer of the component creates the performance description manually. The conformance of this description to the actual implementation is validated by the methods described in [6]. In the second case, the reverse engineering techniques discussed in [8] are used to create the performance descriptions a posteriori from the implemented component. In this case either the component delivering party or the component commissioning party can perform the reverse engineering. Assuming the correctness of these reverse engineering techniques, the resulting performance description can be used for a comparison with the requirements.

The availability of both the performance requirements and the performance description is a necessary precondition for the approach proposed in this work. If both artefacts are present, it is to be determined if the implementation description is a refinement of the performance requirements. For this purpose, a formal refinement definition is specified that allows both parties to check the conformance of implementation to specification regarding the performance properties, on the level of the abstract descriptions. With the help of a checking tool, which could be provided by a trusted certification authority, it is then checked if a refinement relation between the two artefacts holds, and if
positive, the certificate can be issued. In case this performance description has been created manually, a validation has to be performed, which is indicated by “test-based validation” in Figure 1. If the refinement relation holds and the test-based validation is successful, this means that the implementation complies with the performance requirements.

3.2 Hierarchical Refinement

For the refinement of performance, we propose a refinement calculus, which will be explained in detail in the following. With this calculus, different aspects of refinement are expressed. The conformance of external call protocols is checked first, since this conformance is a necessary precondition for components to be compared for refinement. Then, resource demands of active resources like CPU, memory and hard disk are considered.

For the definition of the refinement calculus, we use the Palladio Component Model [1] as a base for our component-based performance models. In the Palladio Component Model, the performance properties of a component are described using the formalism of Resource Demanding Service Effect Specifications (RDSEFF). These specifications contain different types of actions for the modeling of control flow, acquiring/releasing of resources and resource demands for several types of resources, such as CPU, HDD, Network and so on.

As a running example, we will use the RDSEFFs depicted in Figure 2(a) ($R_1$) as performance specification and the RDSEFF in Figure 2(b) ($R_2$) as implementation performance description. The specification RDSEFF $R_1$ includes calls to required services, which are expressed as ExternalCallAction elements. Computations within the component are abstracted as InternalAction elements. The control flow is only modeled between calls to external
services; control flow within external action is abstracted. In the example RDSEFF $R_1$, there are dependencies on input variables: the BranchAction is parameterised by the input variable number, while the number of loop iterations in the LoopAction depends on the size of the input variable array.

In the following sections, we will use this example to illustrate the different parts of the refinement calculus.

3.3 Refinement of External Calls

External Calls describe how a component interacts with other components. They model the calls of a component to required services of another component to which it is connected. Since an actual component instance can be connected to arbitrary components that offer compatible interfaces, no statements can be made about the performance behaviour of these external calls. This is why the compliance of external call sequences is checked first.

We describe the sequence of external calls as a non-deterministic finite-state automaton according to the approach presented in [12] to express the calls to external services including the parameters as transitions in finite automata. We use the substitutibility notion defined there as the criterion for refinement.

3.4 Refinement of Resource Demands

Apart from passive resources, components can also consume active resources such as CPU, memory, or network. In [4], we presented a rule-based approach for the refinement of performance properties based on resource demands.

Since we have already dealt with external calls in the preceding section, they are not regarded here. For the refinement of resource demands, we do not regard the external calls from the RDSEFFs and only deal with the resource demands of internal actions. Using the refinement calculus from [4], we can match the actions of two different RDSEFFs and check for refinement.

If we take the RDSEFF from Figure 2(a) ($R_1$) and check for refinement from it to the RDSEFF of Figure 2(b) ($R_2$), we can see that the resource demand of $R_2$ is 1200 CPU cycles in innerMethod. In $R_1$, we have a branch action, so we will have to take into account all possible execution sequences to check for refinement. In the (simple) example here, there are two possibilities for CPU resource demands:

(i) $1000 + 400 \cdot \#l$, where $\#l$ is the number of loop iterations

(ii) 1800 in the second branch

Note that we do not take the branch condition or probabilities into account here; the refinement rule states that if the resource demands of $R_2$ are lower or equal than those of $R_1$ for all possible traces, then the refinement relation
holds. In our case here, this is true if the number of loop iterations is greater than zero, meaning that refinement holds if the input variable array is not empty. This illustrates that the refinement relation is dependent on the usage context; in this case, one can easily relate the value of the variable array to the refinement relation. In more complex cases, it may not be possible to solve such a dependency analytically (see next section). Thus, it can only be determined whether the refinement relation holds if the usage profile is known. For example, if we know from the usage profile that arrays always have at least size 1, then the refinement relation from $R_1$ to $R_2$ holds in this case.

3.5 Completeness of the Approach

With the calculus for performance refinement, all constructs that are available for the description of Resource Demanding Service Effect Specifications in the Palladio Component Model are covered. The first aspect, External Calls, covers ExternalCallAction elements, but also the control flow elements BranchAction, LoopAction and ForkAction. The second aspect, Resource Demands, covers InternalAction elements with the annotated ResourceDemand descriptions.

If we look at the contexts that a component possesses, the refinement calculus presented is independent from the assembly context of a component, meaning that the component on which refinement is applied can be composed arbitrarily with other components without losing the refinement property. Also, since the third refinement step is only on abstract resource demands like CPU cycles or memory, the approach is also independent from the deployment context of the component.

An RDSEFF element that has not been regarded in the description of the refinement calculus is SetVariableAction. We neglect variables on purpose, following the paradigm of [12] that parameter values should not be regarded in the description of component interfaces. Furthermore, the elements AcquireAction and ReleaseAction are not included in the current approach; the handling of passive resources is left to future work.

4 Assumptions/Limitations

4.1 Usage Profile

The refinement approach presented in this paper is currently only valid under the assumption of a fixed usage profile. This means that in every ResourceDemand element, the stochastic expressions are computable without dependencies on input parameters. This limits the expressivity of the refinement calculus, since refinement cannot be expressed fully independently from all component contexts. In the refinement scenario presented in Subsection 3.1, this limi-
tion means that the usage profile for which the certificate is to be issued has to be defined before the certification process, and the certificate would then be limited to the specified profiles.

4.2 Formal Semantics of the Palladio Component Model

The formal refinement check is only correct under the assumptions that the refinement rules that are used are also correct. The preservation of resource demands or the fulfillment of performance requirements is not checked directly, but is encoded in the refinement rules: if there is a valid application of rules, then the refinement relation holds. The rules themselves are not formally proven to be correct in this paper. This could be achieved using a formal description of the Palladio Component Model, e.g. using the transformation to Queueing Petri Nets (QPN) in [7, chapter 4.4]. The problem is however that the notion of performance also has to be defined in the formalism that is target of such a transformation. Based on this, a transformation can be used to prove that a “QPN refinement” relation exists, and from this fact, the existence of refinement between the RDSEFFs can be proven.

5 Related Work

For the analysis of performance properties of component-based software, many (academic) component models exist, which are mostly targeted on analysis of existing systems. If a software system is designed from scratch, the process should be supported by a development environment that also offers modeling techniques for creating new systems. As an extension to UML, the UML MARTE profile [11] can be used for the modelling of real-time and embedded systems. From the SPE community, several metamodels of the performance domain are available, most notably CB-SPE [3] and KLAPER [5].

Abstract performance models of software component can be created in early stages of development as well as for existing software. In order to obtain performance models from black-box components, Krogmann et al. [8] have developed a reverse engineering approach that uses genetic algorithms, static and dynamic analysis, and benchmarking. The approach has been validated for Java-based systems. The reverse engineering approach is part of the certification approach shown in Figure 1. If an existing component is to be certified, the performance description of the implemented component must be created first. Since it cannot be assumed that sources of the software are available for the purpose of certification, the black-box approach is used to gain the performance properties.

Performance modelling and analysis is often based on simulations and testing. Formal approaches are rare and can best be found in the field of prob-
abilistic model checking, for example the PRISM tool [9], which combines conventional correctness checks with stochastic processes to reason about reliability and performance [2], [10]. However, the approach is lacking the possibility to model the systems parametrically with respect to usage profiles and execution environment.

6 Conclusion and Future Work

In this paper, we present a refinement calculus which checks whether an implemented component conforms to an abstract performance specification. Together with reverse-engineering methods and test-based performance validation, this calculus can be used in a certification scenario to provide for a complete chain of conformance relations from abstract specifications to source code with respect to performance properties. Expressing refinement on an abstract level protects intellectual property such as internal implementation details and source code, while still providing a certification statement that is based on formal methods rather than just meeting standards in a development process.

The calculus uses the parametric modelling features of the Palladio Component Model, so that the refinement is independent from the execution environment of the component, which comprises deployment on hardware and assembly with other components. Independence from the usage profile is planned in a future version of the refinement calculus, but not included at the moment due to the unsolved problem of comparing stochastical functions with respect to performance properties. Since parametric modelling of user behaviour is one of the key advantages of Palladio, including it into the refinement calculus should be a main objective of future work. Furthermore, the handling of passive resources is not included in the current approach.

In a formal development process, the conformance of implementation to specification is checked using formally proven methods. The refinement calculus presented in this paper enriches the component-based development process in the direction of formal development. However, for a completely formal definition of refinement, the semantics of the performance abstractions used in this paper have to be defined and the rules of the refinement calculus have to be proven for correctness. This is future work since the notion of performance refinement is new and there is little related work in this field.

The proposed development process brings together two techniques that are used to ensure the quality of component-based software: performance engineering and software certification. The novelty of this approach is to certify non-functional properties based on formal description languages. Using sophisticated performance descriptions like the RDSEFF formalism of the Palladio Component Model, developers cannot only make performance predictions
at early stages of the development process, but also check if the performance requirements are met by the final product. In a distributed component development process, performance certification of components helps the system architect to choose from existing implementations and to guarantee the overall quality of the system.

References


