LNCS 4829

Markus Lumpe
Wim Vanderperren (Eds.)

Software Composition

6th International Symposium, SC 2007
Braga, Portugal, March 2007
Revised Selected Papers

Springer

Markus Lumpe Wim Vanderperren (Eds.)

Software Composition

6th International Symposium, SC 2007
Braga, Portugal, March 24-25, 2007
Revised Selected Papers

Springer

Volume Editors

Markus Lumpe
Swinburne University of Technology
Faculty of Information and Communication Technologies
Hawthorn, VIC 3122, Australia
E-mail: mlumpe@ict.swin.edu.au

Wim Vanderperren
Vrije Universiteit Brussel
System and Software Engineering Lab - ETRO
Pleinlaan 2, 1050 Brussel, Belgium
E-mail: wvderre@vub.ac.be

Library of Congress Control Number: 2007941317

CR Subject Classification (1998): D.2, D.1.5, D.3, F.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-77350-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-77350-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12206353 06/3180 5 4 3 2 1 0

Preface

On behalf of the Organizing Committee we are pleased to present the proceedings of the 2007 Symposium on Software Composition (SC 2007). The goal of SC 2007 was to bring together the research and industrial communities in order to address the challenges of the component-based software development approach. SC 2007 was the sixth symposium on software composition in the SC series that seeks to develop a better understanding of how software components may be used to build and maintain large software systems.

This LNCS volume contains the revised versions of the papers presented at SC 2007, which was held as a satellite event of the European Joint Conferences on Theory and Practice of Software (ETAPS) in Braga, Portugal, March 24–25, 2007. The symposium began with a keynote on "Composition by Anonymous Parties" by Farhad Arbab (CWI and Leiden University). The main program consisted of six technical sessions related to specific aspects of component-based software development.

In response to the call for papers, we received 59 submissions from over 20 countries and 6 continents. Each paper was reviewed by at least three Program Committee members. The entire reviewing process was supported by Microsoft's Conference Management Toolkit. In total, 15 submissions were accepted as full papers and 5 submissions were accepted as short papers.

We would like to express our gratitude to the General Chair, Judith Bishop, for her invaluable support and guidance that made the symposium in Braga possible. We would like to thank the European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), the International Federation for Information Processing, Technical Committee on Software: Theory and Practice (IFIP, TC 2), and IBM Zurich for sponsoring this event. We are also thankful to the System and Software Engineering Lab at the Vrije Universiteit Brussel for the administrative support in hosting the symposium's Web page. Last but not least, we would like to thank the organizers of ETAPS 2007 for hosting and providing an organizational framework for SC 2007.

September 2007

Markus Lumpe
Wim Vanderperren

Organization

General Chair

Judith Bishop	University of Pretoria, South Africa

General Co-chairs

Markus Lumpe	Iowa State University, USA
Wim Vanderperren	Vrije Universiteit Brussel, Belgium

Program Committee

Uwe Aßmann	Dresden University of Technology, Germany
Brian Barry	Bedarra Research Labs, Canada
Alexandre Bergel	Trinity College, Dublin, Ireland
Vittorio Cortellessa	University of L'Aquila, Italy
Thierry Coupaye	France Telecom, France
Birgit Demuth	Dresden University of Technology, Germany
Maja DHondt	CWI, The Netherlands
Flavio De Paoli	University of Milan, Italy
Dieter Fensel	DERI Galway/Innsbruck, Ireland/Austria
Dimitra Giannakopoulou	RIACS/NASA Ames Research Center, USA
Volker Gruhn	University of Leipzig, Germany
Thomas Gschwind	IBM Research, Switzerland
Arno Jacobsen	University of Toronto, Canada
Mehdi Jazayeri	Vienna University of Technology, Austria
Wouter Joosen	Katholieke Universiteit Leuven, Belgium
Joe Kiniry	University College Dublin. Ireland
Kung-Kiu Lau	The University of Manchester, UK
Welf Löwe	University of Växjö, Sweden
Karl Lieberherr	Northeastern University, USA
Jeff Magee	Imperial College, London, UK
Klaus Ostermann	Technical University of Darmstadt, Germany
Claus Pahl	Dublin City University, Ireland
Arnd Poetzsch-Heffter	Kaiserslautern University of Technology, Germany
Elke Pulvermüller	University of Luxembourg, Luxembourg
Ralf Reussner	University of Oldenburg, Germany
Lionel Seinturier	University of Lille, France
Jean-Guy Schneider	Swinburne University of Technology, Australia
Mario Südholt	École des Mines de Nantes, France
Ragnhild Van Der Straeten	University of Brussels, Belgium
Éric Tanter	University of Chile, Chile

External Referees

Adina Sirbu
Adrian Mocan
Andrea Maurino
Andrew McVeigh
Charles Zhang
Christoph Bockisch
Cuong Tran
Daniel Sykes
Ellen Van Paesschen
Emilia Cimpian
Faris M. Taweel
Fintan Fairmichael
Florian Heidenreich
Frank-Ulrich Kumichel
Guillaume Pothier
Ilie Savga
Ioannis Ntalamagkas
James Scicluna
Jan Schafer
Jean-Marie Gaillourdet
Karl Klose
Kathrin Geilmann
Katja Lehmann
Ling Ling
Maarten Bynens
Marco Comerio
Mick Kerrigan
Mikolas Janota
Mirko Seifert
Nicole Rauch
Pasqualina Potena
Patrick Michel
Perla Velasco
Radu Grigore
Robin Green
Rodolfo Toledo
Shane Brennan
Simone Roettger
Sören Blom
Steffen Zschaler
Stijn Mostinckx
Vinod Muthusamy
Vladyslav Ukis
Wouter Horre
Zhengdao Xu

Sponsoring Institutions

IFIP, Laxenburg, Austria
IBM Zurich, Switzerland
AOSD-Europe, European Network of Excellence in AOSD, Lancaster, UK
Vrije Universiteit Brussel, Belgium
Microsoft Research, Redmond, WA

Table of Contents

Composition by Anonymous Third Parties

Farhad Arbab

Center for Mathematics and Computer Science (CWI), Amsterdam and
Leiden Institute for Advanced Computer Science, Leiden University
The Netherlands

Composition of algorithms has dominated software composition since the inception of programming. The ubiquitous subroutine call acts as the primary composition operator in virtually all programming models and paradigms, appearing in various guises such as function call, method invocation, remote procedure call, etc. The inadequacies of the tight coupling imposed by such composition mechanisms and the need for more flexible alternatives have become clearer along the evolution through object-oriented to component-based, and now, service oriented computing.

Interaction arises out of how a composition allows the active entities in a composed system to play against one another. Communication primitives used in classical models of concurrency to allow interaction among processes in a composed system share the targeted message passing nature of function calls: in order to interact, they generally require a process to directly address foreign entities, such as other processes or channels, that belong to the environment of the process. Interaction constitutes the most interesting and the most difficult aspect of concurrent systems. We have studied protocols for, and various aspects of, interaction in concurrency theory. Curiously, however, no model of concurrency has hitherto considered interaction as a first-class concept! This makes dealing with interaction protocols more difficult than necessary, by erecting a level of indirection that acts as an obstacle between the concrete structures constructed and manipulated in a model, on the one hand, and interaction as the subject of discourse, on the other.

Recognizing the need to go beyond the success of available tools sometimes seems more difficult than accepting to abandon what does not work. Our concurrency and software composition models have served us well-enough to bring us up to a new plateau of software complexity and composition requirements beyond their own effectiveness. In this sense, they have become the victims of their own success. Dynamic composition of behavior by orchestrating the interactions among independent distributed components or services has recently gained prominence. We now need new models for software composition to tackle this requirement.

In this presentation, I describe our on-going work on a compositional model for construction of complex concurrent systems out of simpler parts, using interaction as the only first-class concept. This leads to a simple, yet surprisingly expressive, connector language, together with effective models and tools for composition of complex systems of distributed components and services.

Defining Component Protocols with Service Composition: Illustration with the Kmelia Model

Pascal André, Gilles Ardourel, and Christian Attiogbé

LINA CNRS FRE 2729 - University of Nantes
F-44322 Nantes Cedex, France
(Pascal.Andre,Gilles.Ardourel,Christian.Attiogbe)@univ-nantes.fr

Abstract. We address in this article the description and usage of component protocols viewed as specific services. In addition to inter-component service composition, our Kmelia component model supports vertical structuring mechanisms that allow service composition inside a component. The structuring mechanisms (namely state annotation and transition annotation) are then used to describe protocols which are considered here as component usage guides. These structuring mechanisms are integrated in the support language of our component model and are implemented in our COSTO toolbox. We show how protocol analysis is performed in order to detect some inconsistencies that may be introduced by the component designers.

Keywords: Component, Service, Composition, Protocols, Property Analysis.

1 Introduction

In this work we address the description and usage of component protocols viewed as specific services and described as such. In [9] Meyer suggests a property classification for a Component Quality Model that may lead to trusted components. We consider the *assertions* and *usage documentation* properties which range in the *Behaviour* category from the classification. The first property requires formal descriptions which are helpful to ensure the correctness of the components and their assemblies. The *usage documentation* property requires specific abstraction means in order to help the component-based system developer to build correct assemblies. Clearly, this component documentation property participates in the development of trusted components: this motivates our work. In this context, component documentation should therefore be more than a list of available services (like IDL descriptions); it should overview the component behaviour and constraints, provide some guidelines to use services, describe precisely the usage conditions of services and the interaction conditions. These requirements are fulfilled by the present work which builds on the Kmelia component model [4] which is an abstract component model based on services. Kmelia services are more than simple operations: they enable complex interactions and are the key element to model components and to connect them to make assemblies. The use of service is central to the verification of compatibility when assembling components according to four compatibility layers: signature, structure, contracts and behaviours layers. In a previous article [4] we presented the Kmelia model and we studied the definition and the verification of component assemblies which are based on a *horizontal service composition*. In the present article we extend the service composition.

In the horizontal composition, services of the same level in various components are composed, with respect to the four compatibility levels, to define new services.

To enforce the idea of component documentation, we consider a methodological layer between services and components. This layer deals with the good usage of the components: which services can be used to fulfil a given need and in what order these services should be called. This layer corresponds to the concept of *component protocol* already used in various component models. Compared with related approaches (see Section 4) which are provider-oriented protocols, our proposal suggests user-oriented protocols. This means that the Kmelia component protocols are not a component life-cycle or a component constraint but merely *macro-services* which play an important role in component composition. To support protocols in Kmelia we now introduce a *vertical service composition*, based on hierarchical structuring operators, to build new provided services from existing ones. Building protocols with service composition is beneficial because: the component model stays simple; protocols can be combined and can play a central role in component composition and last, the verification support of service composition may be reused.

The contribution of this article is twofold: new vertical service composition operators are introduced with their formal descriptions; the definition of powerful component protocols, using service composition, to structure the component interface. From the verification point of view we reuse the existing techniques developed for the service level and we adapt them to the protocol level.

The article is structured as follows. Section 2 is a brief overview of the Kmelia formal component model. In Section 3 we define the vertical service composition. Component protocols are developed in Section 4; first we discuss the concept and compare it with related approaches; then we define protocols in Kmelia and illustrate with an example of a bank Automatic Teller Machine system. The verification aspect is studied in Section 5. Last, we conclude in Section 6 and discuss some perspectives.

2 Overview of the Kmelia Component Model

Kmelia is a component model based on services [4]: an elementary Kmelia component encapsulates several services (Fig. 1). The service behaviours are captured with labelled transition systems. Kmelia makes it possible to specify abstract components, to compose them and to check various properties. A Kmelia abstract component is a mathematical model of an open multi-service system that supports synchronous communication with its environment. A component

```
Component C1
 Interface    <Interface descr>
 Types        <Type Defs>
 Variables    <Var list>
 Invariant
              <Predicate>
 Initialisation
  ... // var. assignments

 Services
  ... // as described at side
end
```

```
Provided aService_1 ()
 Interface  <Interface descr>
 Pre        <Predicate>
 Post       <Predicate>
 Behaviour
 init       aStateI
 final      aStateF
 { state_i --label--> state_j
    ... }
end
Required aService_2 ()
 ... //in the same way
```

Fig. 1. Overview of Kmelia syntax

specification language (also named Kmelia) and a prototype toolbox (COSTO) support the Kmelia model. The toolbox already permits formal analysis via Lotos/CADP[1] and Mec[2]. We recall (from [4]) in the following the main definitions and the related notations to facilitate the reading of the article.

Service Description. A *service* s of a component C is defined with an *interface* I_s and a (dynamic) *behaviour* $\mathcal{B}_s$: $\langle I_s, \mathcal{B}_s \rangle$. The interface I_s of a service s is defined by a 5-tuple $\langle \sigma,\ P,\ Q,\ V_s, S_s \rangle$ where σ is the service signature (name, arguments, result), P is a precondition, Q is a postcondition, V_s is a set of local declarations and the *service dependency* S_s is a 4-tuple $S_s = \langle sub_s,\ cal_s,\ req_s,\ int_s \rangle$ of disjoint sets where sub_s (resp. cal_s, req_s, int_s) contains the provided services names (resp. the services required from the caller, the services required from any component, the internal services) in the s scope.

The behaviour $\mathcal{B}_s$ of a service s is an *extended labelled transition system* (eLTS) defined by a 6-tuple $\langle S, L, \delta, S_0, S_F, \Phi \rangle$ with S the set of the states of s; L is the set of transition labels and δ is the transition relation ($\delta \in S \times L \rightarrow S$). S_0 is the initial state ($S_0 \in S$), S_F is the finite set of final states ($S_F \subseteq S$), Φ is a *state annotation* relation ($\Phi \in S \leftrightarrow sub_s$). The transitions in δ (with the$((ss, lbl),\ ts)$ abstract form) have the `ss--lbl-->ts` concrete form.
The transition labels are (possibly guarded) combinations of actions: `[guard] action*`. The actions may be either *elementary actions* or *communication actions*. An elementary action (an assignment for example) does not involve other services; it does not use a communication channel. A communication action is either a *service call/response* or a message *communication*.

Component Description. A component C is a 8-tuple $\langle \mathcal{W}, Init, \mathcal{A}, \mathcal{N}, I, \mathcal{D}_S, \nu, \mathcal{C}_S \rangle$ with:

[1] www.inrialpes.fr/vasy
[2] altarica.labri.fr

- $\mathcal{W} = \langle T, V, V_T, Inv \rangle$ the state space where T is a set of types, V a set of variables, $V_T \subseteq V \times T$ a set of typed variables, and Inv is the state invariant;
- $Init$ the initialisation of the V_T variables;
- $\mathcal{A}$ a finite set of elementary actions;
- $\mathcal{N}$ a finite set of service names;
- I the component interface which is the union of two disjoint finite sets: I_p the set of names of the provided services and I_r the names of required services.
- $\mathcal{D}_S$ is the set of service descriptions which is partitioned into the provided services ($\mathcal{D}_{S_p}$) and the required services ($\mathcal{D}_{S_r}$).
- $\nu : \mathcal{N} \rightarrow \mathcal{D}_S$ is the function that maps service names to service descriptions. Moreover there is a projection of the I partition on its image by ν:
  $n \in I_p \Rightarrow \nu(n) \in \mathcal{D}_{S_p} \wedge n \in I_r \Rightarrow \nu(n) \in \mathcal{D}_{S_r}$
- $\mathcal{C}_S$ is a constraint related to the services of the interface of C in order to control the usage of the services.

The component behaviour relies on the behaviours of its services. The Kmelia components are composable via the interfaces of the involved services. Interface-compatible and behaviour-compatible services are composed at various levels to build *assemblies*. Assemblies and services can be encapsulated into a larger component called a *composition*.

3 Service Composition

In this section we consider two dimensions for service composition; each dimension is related to service behaviour (eLTS). The first dimension already presented in [4] deals with horizontal structuring mechanisms to compose services and components from existing ones on the basis of a client-supplier relation. The second dimension is introduced in this article; it deals with vertical structuring mechanisms for building new services.

3.1 Horizontal Structuring Mechanisms

Horizontal service composition is tightly coupled with component composition and hierarchical links between components. The horizontal structuring mechanisms are established by linking required services to services which are provided either internally or by the caller service or by a third component. These service calls are handled with communication mechanisms. The services are described in such a way that their interactions are made explicit via communication mechanisms. We use communication channels and the standard communication primitives `!` and `?`; they are complemented with `!!` and `??` to deal respectively with service call and service wait. Indeed as service interactions are not elementary, we distinguish their communication operators from the primitive ones.

The interacting services are viewed (from an observer) as one service. Inter-component interactions are based on service behaviour communications. The communications that support the interaction and hence the composition, are matching pairs: *send message(!)-receive message(?), call service(!!)-wait service start(??), emit service result(!!)-wait service result(??).*

Two services are composable if their signatures are matching (types), the assertions are consistent, the (hierarchical) service dependencies are not conflicting and their behaviours are compatible. When services are composed, they are linked via the information available in their interfaces. Provided services are linked to corresponding required services. In the same way, subservices are linked between the composed services. The transition labels of the service behaviours are used to perform the running of the resulting behaviour: either we have independent behaviours or a synchronising behaviour in the case of matching labels.

3.2 Vertical Structuring Mechanisms

In the following we consider and formalise two *vertical* structuring mechanisms that enable us to structure hierarchically the services: they are the *state annotation* mechanism and the *transition annotation* mechanism. Additionally to the flexibility of service description with *optional behaviours* (syntactically expressed as a state annotation) or *mandatory behaviours* (syntactically expressed as a transition annotation) the structuring mechanisms provide a means to reduce the LTS size, to share common services or subservices and to master the complexity of service specification, while preserving the pre/post condition contract at the begining/termination of services (both client and supplier constraints).

We maintain the principle that formally the unfolding of an eLTS should result in a LTS (in a recursive way). The unfolding of a service consists in the unfolding of all its annotated states ($state\_unfold$ in the sequel) and the unfolding of the annotated transitions ($transition\_unfold$ in the sequel). For the formalisation we use the (standard) operational semantics rules with premises and consequences separated by an horizontal line.

The `<< >>` structuring operator. We use the `<< >>` operator to denote an optional service call at any state of a service running. The principle is that the caller of a service s, of a component C, may call a service ss that belongs to the provided interface sub_s of s, when the running of s reaches a state e_i (of the LTS of s) annotated with ss.

This *optional* service call is syntactically noted with $\mathtt{e}_i$ `<<ss>>` in the eLTS of s. In [4] the state annotation mechanisms (called *branching states*) was informally introduced. According to the established link between a required and a provided service, there is a renaming which results in a uniform link name. Therefore, the service call is performed with `_linkName!!serviceName(...)` where `_linkName` (resp. `serviceName`) stands for the established link name (resp. the service name).