LNCS 6144

Benoît Baudry
Eric Wohlstadter (Eds.)

Software Composition

9th International Conference, SC 2010
Malaga, Spain, July 2010
Proceedings

Lecture Notes in Computer Science 6144

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
TU Dortmund University, Germany

Madhu Sudan
Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Benoît Baudry Eric Wohlstadter (Eds.)

Software Composition

9th International Conference, SC 2010
Malaga, Spain, July 1-2, 2010
Proceedings

Springer

Volume Editors

Benoît Baudry
INRIA Rennes Bretagna Atlantique
Campus de Beaulieu
35042 Rennes Cedex, France
E-mail: benoit.baudry@inria.fr

Eric Wohlstadter
University of British Columbia
Department of Computer Science
351-2366 Main Mall
Vancouver, BC V6T 1Z4, Canada
E-mail: wohlstad@cs.ubc.ca

Library of Congress Control Number: 2010929050

CR Subject Classification (1998): D.2, F.3, C.2, D.3, D.1, D.1.5

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-14045-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-14045-7 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2010
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

The goal of the International Conference on Software Composition is to advance the state of research on modularity and reuse in the context of software development based on components, services, features, or models. Software composition is becoming more and more important as innovation in software engineering shifts from the development of individual components to their reuse and recombination in novel ways.

To this end, for the 2010 edition, researchers were solicited to contribute on topics such as component adaptation techniques, composition languages, modeling, as well as emerging composition techniques such as aspect-oriented programming, service-oriented architectures, and mashups. In line with previous editions of SC, contributions were sought focusing on both theory and practice, with a particular interest in efforts relating them.

This LNCS volume contains the proceedings of the 9th International Conference on Software Composition, which was held during July 1–2, 2010, as a collocated event of the TOOLS 2010 Federated Conferences, in Malaga, Spain.

We received 33 initial submissions from all over the world which were considered for evaluation by a Program Committee consisting of 23 international experts. Among these submissions, we selected 10 papers to be included in the proceedings and presented at the conference. Each paper went through a thorough revision process and was reviewed by three to four reviewers. We would like to thank all the authors of submitted papers for their hard work. We are very grateful to the members of the Program Committee as well as to the external reviewers for providing high-quality recommendations that enabled us to select a set of diverse and excellent papers. Finally, we would like to thank the organizers of TOOLS 2010 Federated Conferences for hosting and providing an excellent organizational framework for SC 2010.

Benoit Baudry
Eric Wohlstadter

Organization

General Chair

Judith Bishop	University of Pretoria, South Africa

Program Chairs

Benoit Baudry	INRIA Rennes, France
Eric Wohlstadter	University of British Columbia, Canada

Program Committee

Sven Apel	University of Passau, Germany
Paul Klint	CWI, The Netherlands
Yvonne Coady	University of Victoria, Canada
Simon Denier	INRIA, France
Theo D'Hondt	Vrije Universiteit Brussels, Belgium
Rémi Douence	EMN, France
Laurence Duchien	University of Lille, France
Robert France	Colorado State University, USA
Jeff Gray	University of Alabama, USA
Volker Gruhn	University of Leipzig, Germany
Valérie Issarny	INRIA, France
Wouter Joosen	KU Leuven, Belgium
Andy Kellens	Vrije Universiteit Brussels, Belgium
Philippe Lahire	University of Nice, France
Welf Löwe	Växjö University, Sweden
Markus Lumpe	Swinburne University of Technology, Australia
Raffaela Mirandola	Politecnico di Milano, Italy
Johann Oberleitner	bwin Interactive Entertainment AG, Austria
Cesare Pautasso	University of Lugano, Switzerland
Mónica Pinto	University of Malaga, Spain
Awais Rashid	University of Lancaster, UK
Wolfram Schulte	Microsoft, USA
Clemens Szyperski	Microsoft, USA

Steering Committee

Uwe Assmann	Dresden University of Technology, Germany
Judith Bishop	University of Pretoria, South Africa
Thomas Gschwind	IBM Zurich Research Lab, Switzerland
Oscar Nierstrasz	University of Berne, Switzerland
Mario Südholt	INRIA - Ecole des Mines de Nantes, France

Table of Contents

Model Composition

Composing Models at Two Modeling Levels to Capture Heterogeneous Concerns in Requirements 1
Erwan Brottier, Yves Le Traon, and Bertrand Nicolas

Managing Variability in Workflow with Feature Model Composition Operators 17
Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France

Context-Orientation and Domain-Specific Composition

Composition and Compositionality in a Component Model for Autonomous Robots 34
Olena Rogovchenko and Jacques Malenfant

Event-Specific Software Composition in Context-Oriented Programming 50
Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi

Predicated Generic Functions: Enabling Context-Dependent Method Dispatch 66
Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens

Composing Services

Dynamically Adaptive Systems through Automated Model Evolution Using Service Compositions 82
Adina Mosincat, Walter Binder, and Mehdi Jazayeri

Visualizing and Assessing a Compositional Approach of Business Process Design 90
Sebastien Mosser, Alexandre Bergel, and Mireille Blay–Fornarino

Languages

Construction of Asynchronous Communicating Systems: Weak Termination Guaranteed! 106
Kees M. van Hee, Natalia Sidorova, and Jan Martijn van der Werf

An Advice for Advice Composition in AspectJ 122
Fuminobu Takeyama and Shigeru Chiba

The .NET Primitives for Open, Dynamic and Reflective Component Frameworks 138
Mircea Trofin, Nicholas Blumhardt, and Clemens Szyperski

Author Index 155

Composing Models at Two Modeling Levels to Capture Heterogeneous Concerns in Requirements

Erwan Brottier[1], Yves Le Traon[2], and Bertrand Nicolas[1]

[1] France Télécom R&D, 2 av. Pierre Marzin, 22 307 Lannion Cedex, France
{erwan.brottier,bertrand.nicolas}@orange-ftgroup.com
[2] University of Luxembourg/CsC/SnT, 6 rue Richard Coudenhove-Kalergi, L-1359 Luxembourg
yves.letraon@uni.lu

Abstract. Requirements specification is initially scattered in numerous partial models (viewpoints), describing heterogeneous concerns (typically functional and non-functional ones). To define these concerns, requirements analysts prefer describing them separately with metamodels so that they can be properly identified, reused and tooled. The production of one unified view of requirements from separate viewpoints is a complex issue which requires a composition process working at two levels of modeling. At the meta-level, separate "of-the-shelf" metamodels allow defining either concerns or variation in the operational semantics. These metamodels have to be composed into a core metamodel, which captures the information and semantics needed for expressing and analyzing the requirements of a dedicated application domain (e.g. real-time critical systems, telecom services). At the instance-level, viewpoints are composed to produce a global requirements model, which has to be conformant with the core metamodel. Although the same composition mechanism is used for both levels, we emphasize in this paper the strong coupling between the two steps and the difficulty to make both compositions consistent with each other. We thus propose a process for dealing with two-level of composition. The process is illustrated in the context of a platform specialized for requirements analysis purposes.

1 Introduction

Model-Driven Engineering [1] (MDE) is a data-centric approach of software engineering where heterogeneous information such as software specifications and domain knowledge is captured homogeneously by models and manipulated with various *Domain Specific Languages* [2] (DSL). This information is likely to be described by a collection of models, whatever the modelling-level it pertains to. At an instance-level, large-scale knowledge is distributed among numerous people and then specified separately. At a meta-level, design of complex software need to be properly modularized as advocated by *Aspect-oriented Software Development* (AOSD) approaches [3, 4] to separate concerns and enhance reuse of design solutions. Information modularization

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 1–16, 2010.
© Springer-Verlag Berlin Heidelberg 2010

thus coexists both at meta- and instance- levels, but with different purposes. Composing metamodels expressing separated concerns is already complex, especially when the operational semantics of these concerns is taken into account. The models composition, each of them mixing several concerns, into a model conformant to one core metamodel and its semantics is also an arduous task. The hard point we deal with in this paper is to synchronize both levels when models and concerns are composed.

This two-level separation of information is particularly true during the early stage of a software development process, where stakeholders define their own *viewpoints* of what the system must be [5]. Each viewpoint encapsulates partial requirements information and may be expressed in different formalisms, chosen according to stakeholders' skills, preferences or enterprise habits. Furthermore, viewpoints may describe information relating to several concerns (e.g. functional, security, real-time) of the system-to-be [3].

In previous works [6, 7], we presented the R2A platform [8] (which stands for Requirements to Analysis). Its analysis goal is requirements validation and it has been designed to perform validation techniques such as system test generation from and simulation of requirements (see [9] and [7] respectively). Its core feature is a model composition mechanism designed to compose viewpoints expressed in various formalisms such as constrained natural language and entity-relationship diagram (instance-level composition). This platform has been designed using MDE technologies to make it adaptable to various input requirements languages. However, one monolithic metamodel was used to capture concerns and operational semantics dedicated to validation techniques. This design hampered significantly the adaptability of the platform to other industrial contexts where concerns to capture and analysis goals are different. We believe that this problem is common to most MDE based platforms.

In this paper, we present how we deal with these two levels of information modularization, and we illustrate the approach on the issue of capturing heterogeneous requirements with an evolvable meta-modelling platform. The hard point is to have a process and transformations which do not need to be modified each time the platform (in our case the R2A platform) is adapted to a new industrial context. Synchronization is thus needed between both levels of composition. We propose a two steps composition process working at two levels of modelling. Each step uses a generic composition mechanism presented in [6]. The first step consists in composing *concerns* and *capacities* to produce the *Core Requirements Metamodel* (CRM) of the platform. *Concerns* define types of information to capture from viewpoints and *capacities* specify operational semantics of concerns for analysis goals. The second step produces a unified view of requirements by extracting and composing heterogeneous information from viewpoints. The *global requirements model* obtained is an instance of the CRM and is ready to perform the targeted analysis goals.

This paper is organized as follows. Section 2 provides examples of viewpoints and an overview of the approach in the industrial contexts where the R2A platform has been initially developed. Some metamodels representing concerns and capacities used in these contexts are depicted in section 3. Section 4 illustrates the two steps of the process with viewpoints and metamodels introduced in section 2 and 3. Related work is discussed in section 5 and section 6 concludes.

2 Context and Overview of the Approach

The R2A platform is the result of collaborations of the INRIA/Triskell team with Thalès Airborne System and France Télécom. The main goal for these industrial partners is the evaluation of MDE approach to minimize the cost of *Verification and Validation* (V&V) activities based on *Software Requirements Specifications* (SRS). Requirements studied from France Télécom and Thalès Airborne System are low-level (also termed *operational*) requirements that are the result of initial goals refinement [10]. They were organized by viewpoints and must be composed to perform any V&V activity.

The project with Thalès Airborne System [11] was to (i) validate requirements by simulation (prototyping) and (ii) generate system test objectives for weapon systems in combat aircrafts. The project with France Télécom aimed at (i) checking requirements consistency and (ii) producing a first high-level design (business model, use case and sequence diagrams, test plan) for telecom services, independently from subcontractors.

This section presents an overview of our two-level composition process. Section 2.1 describes two viewpoints for a Library Management System (LMS), illustrating heterogeneous information to be composed and give some definitions. Section 2.2 recalls the principles of our composition mechanism. Section 2.3 presents the two-level composition process using the composition mechanism and discusses the nature of the relationships between instance- and meta-level compositions.

2.1 Example of Heterogeneous Viewpoints to Compose and Analyze

In the context of this paper, we define viewpoint, concern and capacity and design notions as follows:

Definition - ***Viewpoint***: A set of models which describes a part of a system from a given perspective (i.e. for one or a group of stakeholder(s)). These models may be described in various formalisms [5].

Definition - ***Concern****:* A type of information, such as security, business domain or real-time. A concern affects viewpoints [3] and may crosscut several viewpoints (in that case it can be considered as an aspect in AOSD approaches). A concern is defined by at least one metamodel.

Definition - ***Capacity****:* The operational semantics which may be attached to a (or a set of) concern(s). A capacity is thus related to a concern. A concern may embed a variable number of capacities. A capacity is defined by at least one metamodel (in practice Kermeta[1] metamodels).

Definition - ***Design***: A metamodel used to describe one (or a part of) concern or capacity. This terminology is used only for sake of clarity in the following. Designs are reusable units that have to be composed to produce a core requirements metamodel (CRM) for a specific industrial context.

[1] The Kermeta Project Home Page: http://www.kermeta.org

Figure 1 provides two excerpts of viewpoints similar to those that we studied. They capture some requirements of a library management system (LMS) used as an illustration throughout this paper. Both viewpoints in Figure 1 contain a set of descriptions (numbered in the figure), expressed in terms of three different *input requirements languages* (IRL) which are:

- the entity-relationship diagram notation (1 in Figure 1).
- a controlled natural language called the RDL [6] which stands for Requirements Description Language (2-8 in Figure 1).
- a first-order logic notation used to describe actions that actors of the system can trigger (9 in Figure 1).

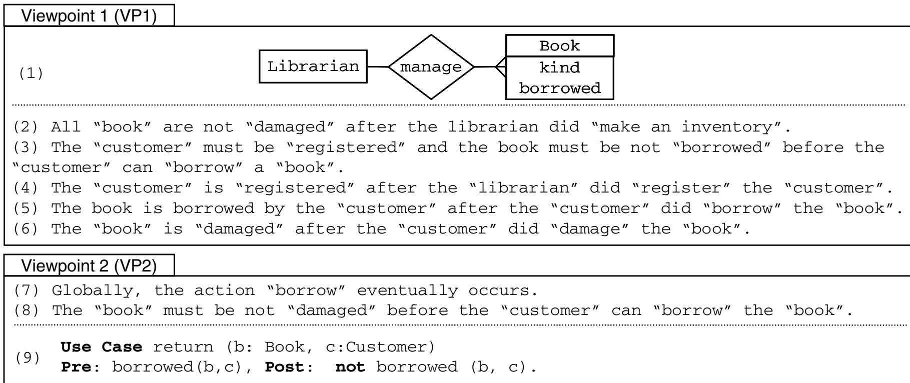


Fig. 1. Examples of viewpoints for a library management system

Viewpoints include partial information on the system and are likely to overlap semantically. For instance, information about the action "borrow" is scattered among viewpoints in descriptions (3, 5, 7, 8). The sentence (3) describes a condition to be satisfied by the customer to borrow a book while sentence (5) states that borrowing a book implies a particular relation (*borrowed*) between the book and the borrower. The notion "borrowed" in (5) is also involved in several descriptions, as a relation in (9) but also as a state of the business concept "Book" in (1, 3).

Different *concerns* of the system crosscut these viewpoints, such as business concepts definition, functional and non-functional descriptions of expected actions. While the diagram (1) defines only business concepts, the two other languages allow specifying information of several concerns. For example, sentence (4) defines two business concepts (*Customer* and *Librarian*) and a boolean state of the first one (*registered*). But it specifies also one causal effect of the business action *register*. Sentence (7) refers the business action *borrow* but describes also a simple non-functional property (temporal property) that the system must fulfill: the action *borrow* must be triggerable with regard to causal chains defined by functional descriptions. These concerns should be defined separately to be properly identified, reused or modified.

Overlaps between information captured by viewpoints must be made explicit to analyze requirements. Viewpoints composition produces a global model of requirements which instantiate concerns defined by designers.

Elements of the global model may be instances of several concerns. Concerns must be composed to explicit their overlaps. Capacities are also inputs of this composition since they describe operational semantics of concerns. In other words, viewpoints must be composed at an instance-level for analysis purposes while concerns and capacities must be composed at a meta-level to permit viewpoint composition.

2.2 Overview of the Composition Mechanism

We proposed in [6] a general MDE-based process for composing viewpoints and producing one global model instance of a *Core Requirements Metamodel* (CRM). In this previous work, we used it at an instance-level. We recall in this section the two steps of this process called *interpretation* and *fusion*, applied at two levels of modeling in the following. The interpretation extracts relevant information in viewpoints and translates it in terms of model fragments, which are instances of the CRM. Each IRL embeds *interpretation semantics*, defined by a set of *interpretation rules* (pattern matching rules). An interpretation rule focuses on a type of information that can be expressed by the IRL and transforms it in terms of one CRM model fragment. Fusion consists in combining all model fragments, each one being an interpretation of requirement description (from a viewpoint), and results in a model of the requirements which is an instance of the CRM. Fusion is also specified with dedicated rules called *fusion rules*.

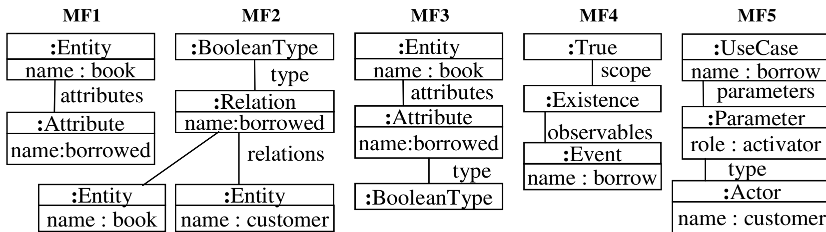


Fig. 2. Five RM model fragments obtained by interpreting viewpoints in Figure 1

Figure 2 presents five model fragments obtained by applying the interpretation step on viewpoints in Figure 1 (a parsing step is required for textual IRLs as described in [6] since interpretation and fusion rules work on models). MF1 is the execution result of an interpretation rule dedicated to extract a business concept state from entity relationship diagrams. It has been extracted from the description (1) of Figure 1. MF2 has been extracted two times from the first part of the sentence (5) and from the logical expression (9). MF3 and MF4 have been extracted from sentences (2) and (7) respectively. MF5 has been extracted three times from sentences (3, 5, 8). It captures the fact that the action *borrow* can be triggered by the actor *customer*.

We can notice that model fragments reflect the partiality and the redundancy of information captured by viewpoints. MF1 is semantically included in MF3, MF2 will be

extracted twice from descriptions (5, 9) and all these model fragments overlap semantically. The fusion step is used to resolve these overlaps. This step is also specified by a set of pattern matching rules called *fusion rules* and used to describe a *fusion semantics* for a given metamodel. Fusion rules can be either equivalence or normalization rules. Equivalence rules specify how to resolve overlaps between model fragments while normalization rules are used to complete the global model after equivalence rules execution or instantiating capacities as presented in section 4.

Figure 3 presents two examples of equivalence rules (ER1 and ER2). Basically, equivalence rule checks if objects in its scope (1) are equivalent. Two objects in the scope are identified as equivalent if they satisfy the *equivalence range constructor* (3) termed constructor. The result of this equivalence identification step is a set of *equivalence ranges*. An equivalence range is a set of equivalent objects. Once equivalence ranges have been identified, each one is resolved i.e. objects in the range are replaced by a new object, instance of the resulting type (2). By default, values of the resulting object properties are the union of values of these properties for all objects in the equivalence range. This default policy can be overridden with *resolution directives* (4).

```
ER1 (c1, c2 : MetaClass) : MetaClass {
  ?: c1.name = c2.name and c1.package = c2.package; }

               (1)                  (2)
ER2 (r1, r2 : Relation) : Relation {
  ?: r1.name = r2.name and fcs(union(r1.source, r2.source)) ≠ null      (3)
                       and fcs(union(r1.target, r2.target)) ≠ null;
  source := fcs(equRanges.collect(r | r.source));     (4)
  target := fcs(equRanges.collect(r | r.target)); }
```

Fig. 3. Two fusion rules defined at a meta-meta-level

For instance, ER1 of Figure 3 states that two METACLASS are merged if they have the same *name*. ER2 is more complex. The function *fcs* in the constructor and resolution directives takes as inputs a set of METACLASS and returns their first common super type if it exists, null otherwise. The keyword “equRanges” represents the equivalence range identified by the rule. Thus, ER2 states that two RELATION are equivalent if (i) they have the same *name*, (ii) their *source* METACLASS are linked by an inheritance relationship and (iii) their *target* METACLASS too. Resolution directives in ER2 determine which metaclasses must be *source* and *target* of the resulting RELATION. Executions of ER1 and ER2 will be illustrated in section 4.

2.3 Overview of the Two-Level Composition Process

Figure 4 gives an overview of the reason why we need a two-level composition process. At meta-level, concerns and capacities are designed with distinct designs. Designs are separated for reusability and extensibility purposes, to allow for the creation of a core requirement metamodel (CRM), dedicated to a specific industrial context. The creation of a CRM is thus seen as a product line where the final metamodel is a

given product adapted to one specific context. Each design embeds one fusion semantics (defined at the meta-level) which describes how a set of its instances must be composed using the fusion step of the composition mechanism.

To compose several designs and their fusion semantics, we apply the composition mechanism at the meta-level. The interpretation step is not used at this level, since designs are instances of the MOF. The fusion step is applied with the fusion semantics embeds in the meta-metamodel MOF (FS_0 in Figure 4). The result of this meta-level composition is the CRM and its associated fusion semantics (FS_f). This FS_f serves to compose the model fragments at the instance-level, as described in the following.

The initial fusion rules embedded in one design refer its elements. Thus, modifications of this design impact these fusion rules. For instance, ER1 specification in Figure 3 will be modified if the concept METACLASS or properties *name* and *package* are modified. When designs are composed, fusion semantics FS_1 to FS_n will be impacted, since some METACLASS will be combined. Furthermore, elements in these designs become elements of the CRM after composition. Therefore, fusion rules associated to designs become fusion rules of FS_f. The composition of fusion rules associated to designs is a side effect of the composition of designs.

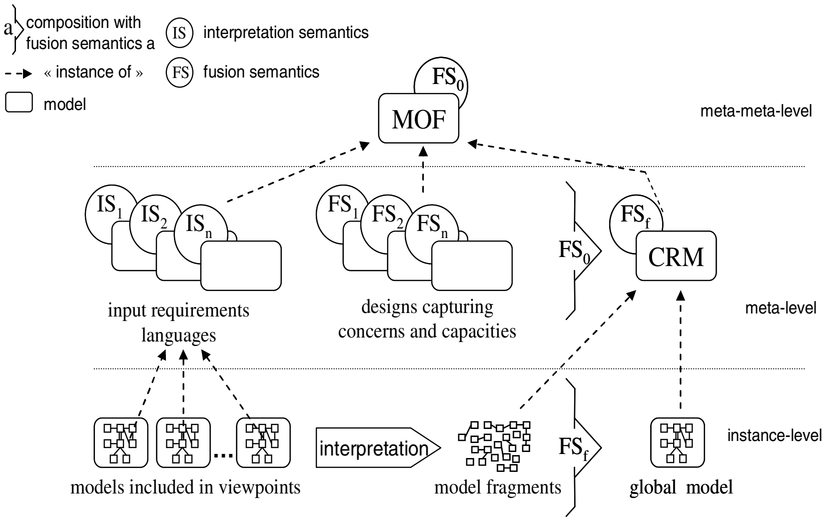


Fig. 4. Overview of the two-level composition process

At instance-level, models included in viewpoints are instances of several IRLs. Each IRL embeds an interpretation semantics (IS_1 to IS_n in Figure 4) that produce, from models instance of it, a set of model fragments of different concerns. At meta-level, the composition of designs will impact also interpretation rules. When designs have been composed, interpretation semantics produce CRM model fragments. The fusion semantics of the CRM (FS_f) is applied on them and the global model is obtained (see Figure 4).

To summarize, the two-level composition process consists in:

- at the meta-level, applying the fusion step of the composition mechanism on designs and embedded fusion semantics. In that case, the fusion semantics executed is FS_0, defined at the meta-meta-level. The CRM and FS_f are produced. FS_f contains modified fusion rules of FS_1 to FSn. Interpretation semantics IS_1 to IS_n of IRLs are also modified.
- at the instance-level, applying the composition mechanism on models included in viewpoints. The interpretation step produces a set of CRM model fragments. The fusion step is executed on them to obtain the global model. In that case, the fusion semantics executed is FS_f, defined at the meta-level.

Model fragments of Figure 2 have been produced in the case where concerns have not been composed (these concerns are presented in the next section). The CRM has not been produced and each model fragment obtained by interpretation is an instance of one concern. It is then not possible to have a unified view of requirements and to perform analysis. The remainder of this paper illustrates how concerns and capacities introduced in section 3 and viewpoints of Figure 1 are composed by using the process described in this section so that one global model of viewpoints is obtained.

3 Examples of Concerns and Capacities

Based on our experiments at Thalès Airborne System and France Télécom, we have defined a formalism which handles a significant subset of typical industrial requirements. This formalism relies on a state-based specification paradigm. It captures currently (i) business domain definition, (ii) functional description of the system and (iii) temporal constraint that the functional description must satisfy. The formalism is the current CRM of the platform. It is represented by a metamodel, called RM (which stands for Requirements Metamodel), and described by a set of designs (concerns and capacities). The rest of this section presents some parts of these designs. The reader could refer to [6-8] for more details.

Designs of Figure 5 describe RM concerns. Design (a) gives an overview of the RM metamodel, structured in main views (we show only those that we present in the paper): the *Functional View*, *Analysis View* and *Temporal Constraint View*. Notice that grey metaclasses are interfaces.

The functional view (b) describes system functions and actor actions, represented by the metaclass USECASE. Conditions and effects of use case activations are described by pre- and post-conditions (relationships *preCondition* and *postCondition*). These conditions are expressed as first order logic expressions (metaclass EXPRESSION). Use case contains a set of formal parameters (metaclass PARAMETER).

Each parameter represents a BUSINESSCONCEPT involved in the use case and its role in the use case (for instance "activator", which states that the parameter is responsible of the use case activation). A business concept is an ACTOR if it can trigger at least one use case, otherwise it is a CONCEPT. Expressions are instances of metaclasses (logical operators, boolean comparisons and atomic propositions) defined in the *expression view*, which is not described in this paper. The analysis view (c) captures roughly a UML class diagram. Entities have Properties with possible states (ATTRIBUTE) and

relationship with others relations (RELATION). Properties have a type (DATATYPE): BOOLEANTYPE or ENUMERATIONTYPE (a finite set of literal values, representing strings, integers or intervals).

Instance of the functional concern can be also instantiated to represent a particular system behavior. In another words, instances of designs (b) and (c) can be also instantiate in order to capture system states and transitions. Design (e) is a small excerpt of the design dedicated to this purpose (called InstanceView). Due to space limitation, we introduce here only two concepts used in the running example of this paper. USECASEINSTANCE represents an instance of the USECASE metaclass and UCSYSTEMSTATE corresponds to a state of the system at a given moment (see [7, 9] for more details).

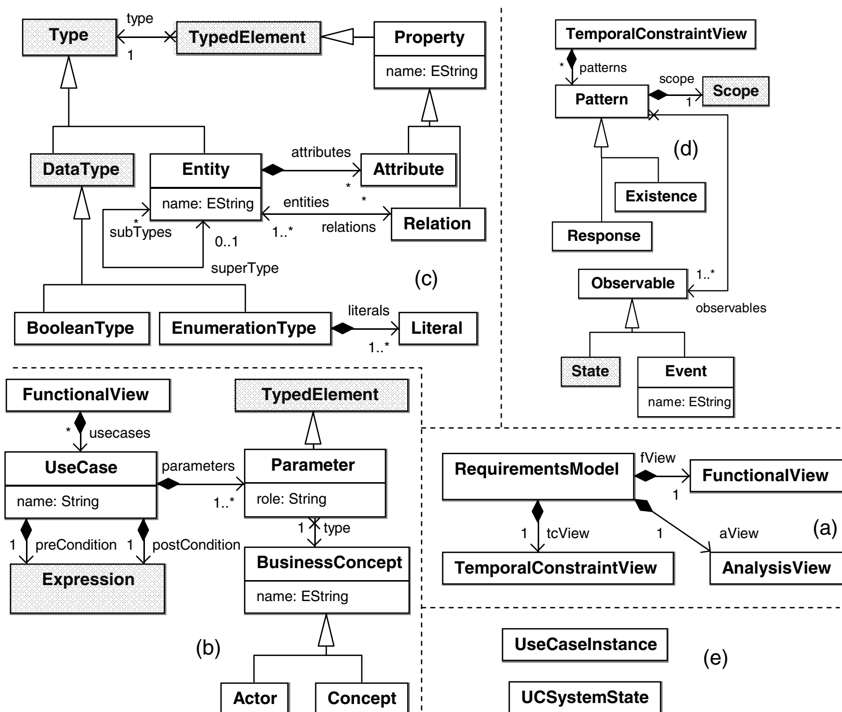


Fig. 5. Some concerns of the current version of the R2A platform

The temporal constraint view (d) captures temporal property patterns corresponding to the Dwyer patterns [12]. A PATTERN captures a temporal constraint that the functional description of the system must satisfy. The pattern RESPONSE may capture for instance that if a condition (first element of *observables*) on the current system state holds, then the other condition (second element of *observables*) must hold. OBSERVABLE can be either a condition on the current system STATE or an EVENT. The SCOPE of a pattern restricts its domain of truth. In the description (7) in Figure 1, the scope is the keyword “Globally”. It means that the property must be always true (in a logical representation, it corresponds to the boolean constant true).

As introduced in section 2.1, concerns are specified by a set of designs. For example, the business concern of the RM is represented by the design (c) while the functional concern is specified by the design (b) and the expression view. We can also notice that these concerns crosscut: the functional concern includes the business domain concern (some concepts in the expression view overlaps with the business concern) and the property concern shares the notion of use case with the functional one (we will see in section 4 that metaclasses EVENT and USECASE overlap). Model fragments of Figure 2 are instances of these concerns: MF1, MF2 and MF3 instantiate the business concern, MF4 instantiate the temporal property concern and MF5 the functional one.

The RM metamodel embeds also capacities. Design (f) of Figure 6 is a capacity designed to weave simulation functionalities in a functional concern. The metaclass SYSTEM contains a method s*imulate* which starts the simulation process. The metaclass SIMULATOR is linked to initial and current states of the system. It is also linked to the set of transitions which can be run according to the current state (relation *runnables*). This set is automatically calculated when one of the first third methods of SIMULATOR is executed. The first two methods allow defining the initial state for a simulation. The third method is used to run a transition, included in the *runnables* set. Furthermore, this design defines the notion of SCENARIO which is a sequence of EXECUTIONSTEP, starting from an initial STATE. Previous work [7] details the simulation semantics in deeper details.

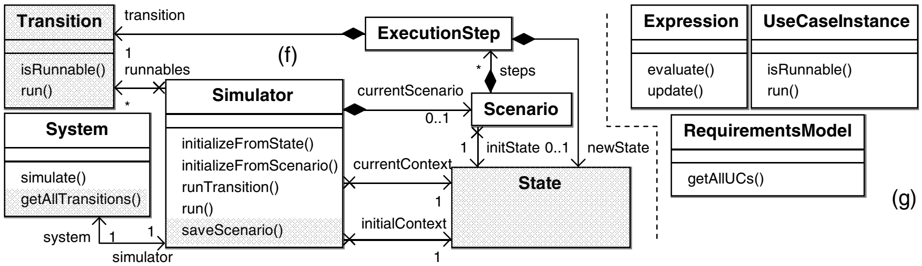


Fig. 6. A generic capacity dedicated to simulation purpose (f) and design additions (g) used to specialize this capacity for concerns of Figure 5

Methods defined in this design are implemented with the Kermeta language which is an open source meta-modeling environment, developed by the Triskell team at IRISA. The reasons of this choice are twofold. Firstly Kermeta has been especially designed to enable the description of operations in metamodels thus providing semantics and behavior for metamodels [13]. This makes instances of capacities executable (see [7] concerning the capacity of Figure 6). Secondly Kermeta code can be loaded as a model. This is an important point since the meta-level composition may impact code defined in capacities as discussed in section 4.

Design (f) is a high-level design of one possible simulation implementation. Some methods in its metaclasses are not implemented (the grey ones) and the metaclass TRANSITION and STATE are abstract concepts. Notice that grey methods are abstract (not implemented). Design (g) describes how to specialize the simulation capacity for the

RM. It specifies methods in concern metaclasses defined in Figure 5 so that abstract methods of design (f) will be implemented after meta-level composition (see section 4). For verifying whether a use case instance can be triggered, the EXPRESSION class has an *evaluate* method which returns true if the current system state implies the precondition. The method *run* in the USECASEINSTANCE metaclass modifies the current system state by calling the method *update* of the EXPRESSION class with the corresponding post-condition.

4 Illustration of the Two-Level Composition Process

This section explains how we use the fusion step to compose at two different levels of modeling designs and viewpoints. Some fusion rules are defined at a meta-meta-level to compose designs while others are defined at a meta-level in designs to compose viewpoints. Fusion rules embedded in designs are impacted by the meta-level composition since they refer designs elements and in particular metaclasses.

Designs introduced in section 3 capture concerns and capacities. They are semantically linked, although they have been designed separately. Making explicit these relations is a prerequisite to any RM instantiations, in particular viewpoints composition. To compose them, we apply the fusion semantics embedded in MOF as described in section 2.3.

ER1 and ER2 in Figure 3 are part of this fusion semantics. When these two rules are executed to produce the RM, ER1 identify and resolve an overlap between metaclasses TYPEELEMENT of designs (b) and (c). This is an obvious case of equivalence between designs. Relations *type* in (b) and (c) overlap too. More precisely, relation *type* between TYPEDELEMENT and TYPE metaclasses in (c) subsumes relation *type* between PARAMETER and ENTITY metaclasses in (b). ER2 will identify and resolve it automatically.

The equivalence identification of our approach is basically based on name matching and equivalence checking in the immediate context of observed objects. However, concerns and capacities to compose are described separately and overlapping metaclasses may be named differently in distinct metamodels, according to the role they play. The pairs of notions (USECASEINSTANCE, TRANSITION) or (ENTITY, BUSINESSCONCEPT) in Figure 5 and Figure 6 are relevant examples of such a conflicting situation, known as terminology clash (see [14] for a classification of conflicting situations). Metaclasses EVENT and USECASE overlap since they designate both actions of the system. But ER1 will not identify these three overlaps since names of these metaclasses are different.

To overcome this issue, our composition mechanism provides a set of directives (called *pre-composition directives*) used to specify manually equivalence at the meta-level. Figure 7 presents such directives to compose designs of Figure 5 and Figure 6. Refinement directive of the form "*a* refines *b*" declares that metaclass *a* is another representation of *b*. It sets *name* and *package* property values of *a* with those of *b* (*a* and *b* have then the same name and package). By this way, ER1 of Figure 3 will match *a* and *b* during composition. Override directive of the form "*a* overrides *b*" declares that the method *b* is replaced by *a*. It sets the name of *b* with the name of *a*.

B.BusinessConcept **refines** C.Entity
D.Scope **refines** E.SystemState.
D.State **refines** E.SystemSate.
D.Event **refines** B.UseCase.
F.System **refines** A.RequirementsModel.
F.Transition **refines** E.UseCaseInstance.
F.State **refines** E.SystemState.
G.RequirementsModel.getAllUcs **overrides** F.System.getAllTransitions.

Fig. 7. Pre-composition directives to compose concerns of Figure 5 and capacity of Figure 6

Figure 8 depicts a piece of the RM obtained by composition where relations *type* and ENTITY notions of designs (b, c) have been automatically merged by ER2 and ER1 respectively. Several metaclasses in Figure 8 have been impacted by directives of Figure 7. Among them, we can notice the metaclass REQUIREMENTSMODEL which is the composition result of metaclasses REQUIREMENTSMODEL and SYSTEM where the method *getAllUcs* returns all system transitions (i.e. USECASE instances by composition).

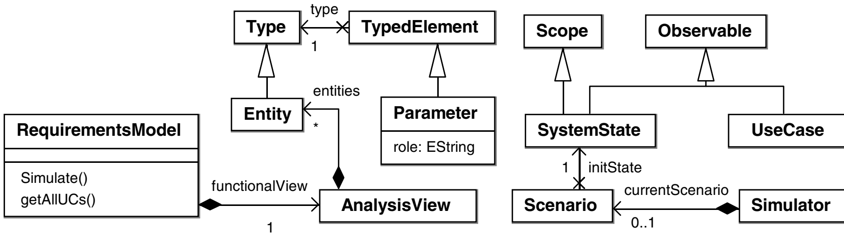


Fig. 8. Piece of the RM after composition

Each RM concerns and capacities embeds a fusion semantics as introduced in section 2.3. Figure 9a provides few fusion rules from them, defined at a meta-level as opposite to the rules of Figure 3. ER3 to ER5 are defined on the business domain concern, ER6 and ER7 on the functional concern (design (b)), ER8 on the temporal property concern and NR1 on the simulation capacity. ER3 specifies that the concept of RELATION subsumes the concept of ATTRIBUTE. ER4 states that two ATTRIBUTES are equivalent if they are contained by an equivalent ENTITY (*attributes*$^{-1}$ means to navigate the link *attributes* in the other side).

NR1 is a particular type of fusion rule called *normalization rule*. It is part of the fusion semantics of aspect (e). Normalization rules are executed at the end of the composition process, after equivalence rules. They are used to complete the global model without adding any new concern information. Examples of normalization rules are described in [6]. Such a rule consists of a guard and an action (respectively terminals "?" and '!' in Figure 9). The action is executed if the guard is evaluated to true. NR1 states that an instance of the SIMULATOR must be linked to an instance of SYSTEM. NR1 is thus an imperative form of a contract that an instance of RM must fulfill if the simulation capacity has to be woven.

ER3 (a :Attribute, r : Relation) : Relation { ?: a.*name* = r.*name;*
linkedEntities := *ER*.collect(r: Relation|r.linkedEntities) ∪ c.collect(a: Attribute | a.owningEntity). }
ER4 (a_1, a_2 : Attribute) : Attribute { ?: a_1.attributes^{-1} = a_2.attributes^{-1} and a_1.name = a_2.name; }
ER5 (e_1, e_2 : Entity) : Entity { ? e1.name = e2.name; }
ER6 (u_1, u_2 :UseCase) : UseCase { ?: u_1.name = u_2.name; }
ER7 (a : Actor, c : Concept) : Concept { ? a.name = c.name; }
ER8 (e_1, e_2 : Event) : Event { ?: e_1.name = e_2.name; }
NR1 { ?: card(model.instancesOf(Simulator)) < 1
!: Simulator s := create(Simulator); s.system := model.instancesOf(System).one(); } (a)

- -

ER8’ (e_1, e_2 : UseCase) : UseCase { ?: e_1.name = e_2.name; } (b)
NR1’ { ?: card(model.instancesOf(Simulator)) < 1
!: Simulator s := create(Simulator); s.system := model.instancesOf(RequirementsModel).one(); }

Fig. 9. Fusion rules at a meta-level

Specifications of these rules may be impacted by the composition of designs as introduced in section 2.3. For instance, ER8 depends on the metaclass EVENT which has been refined as USECASE. These rules are automatically rewritten since they are instance of the fusion language metamodel (not presented here). Kermeta code specified in the simulation capacity of Figure 6 is rewritten in the same way. The fusion rules of Figure 9b are new versions of rules in Figure 9a that have been automatically refactored by the meta-level composition (in this rule, only one element has been rewritten, the underlined one). We can notice that the ER8’ is semantically equivalent to ER6. In fact, conflicts between fusion rules may appear after the meta-level composition. The mechanism to detect them automatically is out the scope of this paper.

It is possible that the resulting CRM is under-specified or inconsistent. So far, we have noticed three well-formed rules that must be satisfied by any CRM and its fusion semantics. The first two rules are OCL constraints defined on the MOF and implemented with the invariant checker feature of Kermeta:

(1) Leaf metaclasses of the CRM inheritance tree must be concrete.
(2) All metaclass methods defined in the CRM must have a concrete implementation.
(3) There are no conflicts in the CRM fusion semantics.

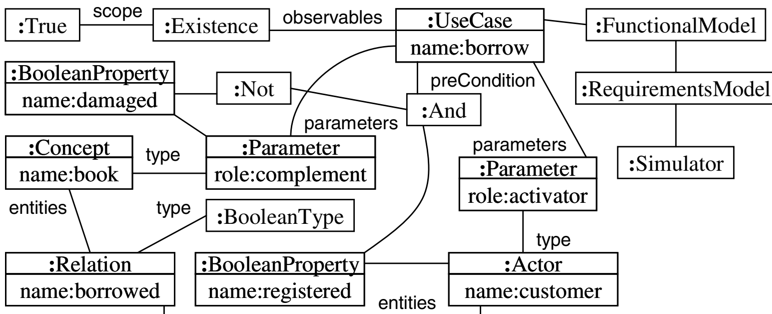


Fig. 10. A piece of the global RM model obtained by composing viewpoints of Figure 1

We have produced the RM and its fusion semantics by meta-level composition. To compose viewpoints in Figure 1, we use the two steps of the composition mechanism at an instance-level. First, the interpretation is used to extract RM model fragments from viewpoints. Second the fusion semantics of the RM is applied on model fragments to produce the global requirements model. Figure 10 is a partial view of the global model produced from viewpoints of Figure 1. It includes information captured by model fragments of Figure 2. The result of ER3 execution on MF1 is illustrated in Figure 10. Only the notion of relation "borrowed" remains and it is properly linked to "customer" and "book" entities.

This model illustrates the interest of our approach. First, the action "borrow" is represented in the global model of Figure 10 by one object (the use case "borrow") which is linked to other objects via instances of relationship defined in different concerns (e.g. *observables* and *preCondition*). If concerns have not been composed, this action will be represented separately by an instance of UseCase and an instance of Event and no semantic relations will appear between them. Second, viewpoints composition has combined all information scattered in viewpoints concerning the use case *borrow*. Third, an instance of Simulator has been properly created and linked the RequirementsModel since the simulation capacity has been composed with concerns. The model is then directly executable to perform simulation by executing methods of the simulator object. In other words, if the composition mechanism is applied only at the instance-level, the composition result will be a set of separated models (each one instance of a concern) without operational semantics, instead of one global model (fusion semantics of each concern will be applied on model fragments, each one an instance of one concern).

5 Related Work

To the best of our knowledge, we are not aware of any work identifying and addressing the issue of composing information at two-level of modelling and discussing the strong coupling between these levels during composition. First we present related work in the requirements engineering domain [15-18] where viewpoints composition is an important issue since it produces a global model which reflects the global complexity of the stakeholder's need [15]. Second, we describe related work in the AOSD domain [3, 4], where capacities and concerns must be properly composed after separation to face of complexity involved by crosscutting concerns in software development processes.

Concrete composition solutions have been proposed in requirements engineering as a way to perform analysis goals [16, 17]. In [17], the authors define a framework to produce an HOL (High Order Logic) model from viewpoints expressed in different notations. Authors of [16] propose a method for composing viewpoints expressed in Z. These two approaches are tedious since much of the work is not automatic and are specialized to one IRL.

Sabetzadeh and Easterbrook [15] provide an interesting composition framework based on a formal definition of models to compose. The composition operator (category-theoretic concept of colimit) is formally defined and traceability links are automatically inferred. However this operator requires a model type definition which

restricts highly the accepted IRLs (the mathematical framework is too much restrictive). Furthermore viewpoints must be expressed in the same input requirements language and overlaps are identified manually.

Zave and Jackson outline in [18] main issues and foundations of multi-formalism composition of heterogeneous information. They describe a formal CRM designed to formalize a wide range of conceptual modeling language where composition is the conjunction of elements extracted from viewpoints. They describe a notion of functions dedicated to each IRL for assigning input model semantics in the CRM. They point out that these functions may be adapted since semi-formal languages include semantic variability points. Our composition mechanism conforms to these ideas. In our approach, these functions are sets of interpretation rules and can be easily adapted because we have a rule-based approach. The major difference with this work is that we describe a concrete composition mechanism.

Kompose [4] is a meta-modeling approach to compose metamodels, in particular aspects capturing capacities. It is built on the Kermeta language. Similar to our approach, fusion semantics is described by a set of rules. Equivalences between objects are automatically calculated with regards to object structures and resolved. The main differences with our composition mechanism are threefold: (i) Kompose take as inputs models expressed in one language, (ii) complex compositions can not be specified since equivalent objects must be instance of the same type and (iii) specification of resolution directives are not allowed.

The approach [3] targets the identification of crosscutting concerns in viewpoints in order to measure influences between concerns and viewpoints and help negotiations between stakeholders. This is done by using composition rules, which does not allow to obtain a global model of the requirements but to state how concerns are related.

6 Conclusion

In this paper we studied a problem of models composition, (1) when metamodels and their semantics have to be composed into one core metamodel, and (2) when partial instances of this core metamodel have to be composed to produce a complete model. This problem occurs when dealing with requirements models but we believe that this is a general problem which occurs each time a meta-modeling platform is built with a product line approach. The core metamodel (a specific product) is the composition of several concerns and capacities which are needed in a given industrial context to support specific analysis goals.

It is clear that having distinct concerns and distinct capacities implies another complexity i.e. to resolve their overlaps when composing them to enable models composition at the instance level. We believe that the composition process presented here can help to handle this complexity. In future work, it could be interesting to study in more details the relationship between fusion semantics defined at the meta-meta- and meta-level. A formalization of this relationship may be helpful to detect automatically conflicts between fusion rules obtained after the composition at the meta-level. It is indeed an important point with regard to the scalability of the approach.

References

1. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer 39(2), 25–31 (2006)
2. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices 35(6), 26–36 (2000)
3. Rashid, A., Moreira, A., Araújo, J.: Modularisation and Composition of Aspectual Requirements. In: Aspect Oriented Software Development, AOSD 2003, Boston, Massachusetts, USA (2003)
4. France, R., et al.: Providing support for Model Composition in Metamodels. In: IEEE EDOC (2007)
5. Sommerville, I., Sawyer, P.: Viewpoints: principles, problems and practical approach to requirements engineering. Annals of Software Engineering 1997(3), 101–130 (1997)
6. Brottier, E., et al.: Producing a Global Requirement Model from Multiple Requirement Specifications. In: Entreprise Distributed Object Computing Conference, IEEE EDOC, Annapolis (2007)
7. Baudry, B., Nebut, C., Le Traon, Y.: Model-driven Engineering for Requirements Analysis. In: Entreprise Distributed Object Computing Conference, EDOC, Annapolis (2007)
8. The Requirements to Analysis platform website (2006), `http://www.irisa.fr/triskell/Softwares/protos/r2a/` (cited 2008)
9. Nebut, C., et al.: Automatic Test Generation: A Use Case Driven Approach. IEEE Transactions on Software Engineering (2006)
10. Lamsweerde, A.v.: Goal-Oriented Requirements Engineering: From System Objectives to UML Models to Precise Software Specifications. In: International Conference on Software Engineering (2003)
11. Lugato, D., et al.: Automated Functionnal Test Case Synthesis from THALES industrial Requirements. In: RTAS 2004, Toronto, Canada (2004)
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering (1999)
13. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
14. Lamsweerde, A., Darimont, R., Letier, E.: Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Transactions on Software Engineering 24(11) (1998)
15. Sabetzadeh, M., Easterbrook, S.: An Algebraic Framework for Merging Incomplete and Inconsistent Views. In: International Conference on Requirements Engineering (2005)
16. Ainsworth, M., et al.: Viewpoint Specification and Z. Information and Software Technology 36(1), 43–51 (1994)
17. Day, N.A., Joyce, J.J.: A Framework for Multi-Notation Requirements Specification and Analysis. In: IEEE International Conference on Software Engineering (2000)
18. Zave, P., Jackson, M.: Conjunction as Composition. Transaction on Software Engineering and Methodology 2(4), 379–411 (1993)

Managing Variability in Workflow with Feature Model Composition Operators

Mathieu Acher[1], Philippe Collet[1], Philippe Lahire[1], and Robert France[2]

[1] University of Nice Sophia Antipolis, France
I3S Laboratory (CNRS UMR 6070),
06903 Sophia Antipolis Cedex, France
{acher,collet,lahire}@i3s.unice.fr
[2] Computer Science Department,
Colorado State University, USA
france@cs.colostate.edu

Abstract. In grid-based scientific applications, building a workflow essentially involves composing parameterized services describing *families of services* and then configuring the resulting workflow product line. In domains (e.g., medical imaging) in which many different kinds of highly parameterized services exist, there is a strong need to manage variabilities so that scientists can more easily configure and compose services with consistency guarantees. In this paper, we propose an approach in which variable points in services are described with several separate feature models, so that families of workflow can be defined as compositions of feature models. A compositional technique then allows reasoning about the compatibility between connected services to ensure consistency of an entire workflow, while supporting automatic propagation of variability choices when configuring services.

1 Introduction

In grid-based scientific collaboration communities, scientists build workflows by assembling services that, in many cases, perform complex tasks [1]. For example, in the grid-based medical imaging community, scientists compose diverse image processing services to create chains that meet their specific needs. To support reuse, services can be parameterized, thus allowing a scientist to tailor a service to a particular context. Current approaches to assembling grid-based services are labour-intensive [2] and require scientists to manually manage knowledge about *i)* the variable points supported by services, and *ii)* the restrictions on how services must be tailored and composed to meet end-to-end Quality of Service (QoS) or other requirements. When a wide variety of parameterized services exists, the tasks of identifying, tailoring and composing services become tedious and error-prone [3], especially in grid-based medical imaging. As identified in previous work [4, 5], the difficulty of provisioning and composing such services stems from the lack of mechanisms for managing variabilities within and across services. The above problems give rise to the following challenges. The first challenge is to provide mechanisms that enable service providers (e.g., research

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 17–33, 2010.
© Springer-Verlag Berlin Heidelberg 2010

scientists, workflow or grid experts) to capture the commonalities and variabilities in parameterized services that are offered on the grid. The second challenge is concerned with providing support for tailoring and composing services such that service consumers can ensure the consistency of resulting workflows with well-defined properties.

In order to meet these challenges, we describe in this paper a rigorous approach to composing parameterized services into workflows. The approach utilizes Software Product Line (SPL) and Aspect-Oriented Modeling (AOM) techniques. The goal of SPL engineering is to produce reusable artifacts that can be used to efficiently build members of a software product family [6]. The reusable artifacts encapsulate common and variable aspects of a family of software systems in a manner that facilitates planned and systematic reuse. A parameterized grid-based service can be viewed as an SPL. We observe that the variabilities in a parameterized service can be described along a variety of dimensions. For example, in a medical imaging service, three commonly used dimensions concern QoS features, image formats and communication protocols for data transmission. We rely on prior results [7] to separate aspect models which exhibit variability and compose them to produce a comprehensive variability model. This modular technique allows applying separation of concerns principles and thus limits the considered variabilities only to relevant concerns.

In the present approach, a workflow is created by first composing *families of services* and then configuring the resulting workflow product line. Feature models [8,9,10] (FMs) are used to describe the common and variable features in a tailorable service. The variable points in a parameterized service are described by multiple separate FMs, where each FM describes a set of variable points in a particular dimension. A set of composition operators is used to *i)* insert a concern with variability into the description of services and *ii)* merge variability models of connected services. The FM composition operators are used to reason, at the FM level, on services' dependencies specified by the user and identified as active in the workflow. Using these operators, it is possible *i)* to analyze the entire workflow by checking the consistency of families of dependent services and *ii)* to infer variability information and propagate user choices according to variability information described in each family of services. We also consider the impact that workflow constructs (sequence, concurrency, if-then-else condition) have on service composition. The approach assists users with their decision-making process and can largely reduce the sets of configurations to be considered when tailoring and composing services.

2 Motivation and Overview of the Approach

Scientific workflows are increasingly used for the integration of existing, legacy tools and algorithms to form larger and more complex applications, e.g., for scientific data analysis or computational science experiments. In the medical imaging area (e.g., see [11]), scientific workflows are deployed on grids for addressing the computing and storage needs arising from manipulation of large fragmented medical data sets on wide area networks. Service-oriented architectures (SOA) are especially suited to such a domain: There is a need for *reusable self-contained*

services that provide standardized interfaces for calling application code as well as information exchange protocols [12]. In SOA, services are atomic entities that are composed to produce complex (business) processes implementing workflows.

Managing Many Concerns. Many scientific services have a large number of input ports and parameters, but not exclusively. Individual data items processed by scientific workflows are very much related to each other and there is a need to maintain data cohesion: Dependencies between different services within a workflow system must be managed from several perspectives. More generally, deployed services contain a lot of information related to the environment in which they are deployed and composed. In the case of medical imaging services on the grid, service providers supply basic imaging services, implemented in a variety of languages, packaged with information needed to compose the services with other services. In addition, providers have to manage the numerous non-functional properties that are exploited during deployment or runtime, in order to meet quality of service (QoS) goals. The overall issue for users of the workflow is to deal with services' dependencies in the workflow while addressing a large amount of concerns.

An SPL Approach. Rather than providing services in hopes that opportunities for reuse will arise during the design of a workflow, a proactive strategy is to apply SPL principles and plan which characteristics of a service are likely to be systematically reused. The ability to efficiently create many variations of a service and capitalize on its commonalities can improve its composability and increase the extent to which service logic is sufficiently generic so that it can be effectively reused. Our previous work indicates that there is significant *variability* in medical imaging services on the grid [4,5]. For example, a service is able to read and process several medical image formats; some services use network protocols that do not support the transmission of a "receive acknowledge" to indicate that the packet has been received whereas some other services do have this capability. A medical imaging service that exhibits variability can thus be treated as an SPL or *family of services*. For each concern of a service, there are several alternatives that the user has to consider and choose from to derive an

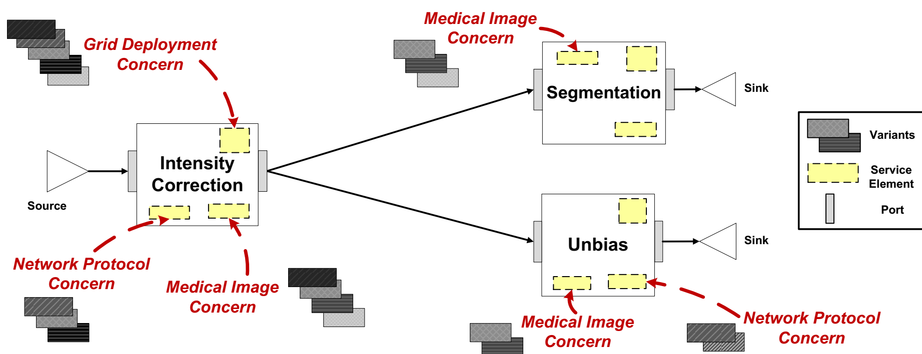


Fig. 1. Workflow, Services, Concerns and Variants

actual service. Adopting an SPL approach, a family of services is described from a variety of *variable concerns*, i.e., a concern with variation points, and thus can be represented as a set of variants.

We believe that separation of functional and non-functional concerns with variability can improve the reusability of services. In Figure 1, three concerns are woven into elements of the Intensity Correction service to augment its description. *Medical Image* describes the medical images input formats that the service is able to process. *Grid Deployment* provides information about service deployment on the grid, e.g., the operating system needed to run the service on the grid. *Network Protocol* represents the network protocol used by the service. When the service Intensity Correction is connected to the services Segmentation and Unbias, several concerns of Intensity Correction may be related to several concerns of Segmentation and Unbias. For instance, users may require that the medical image output of Intensity Correction be compatible with the medical image input of Segmentation and Unbias; or that the network protocol used by Intensity Correction be consistent with the network protocol used by Unbias.

Key Issues. The goal of the SPL approach promoted in the paper is to manage not only the variability of the family of services but also the variability of the resulting composed services. The following key challenges are targeted by our approach. A first challenge is *to cope with multiple dimensions of a family of services* by providing mechanisms to augment the service description with variants from the concerns dimension. These variants can be woven in at several points (e.g., port, interface) in the service description. A second challenge is *to ensure that families of services are consistently composed in the workflow.* At the workflow level, the links between services and their semantics exist in various forms (complex dependencies, input/output dataflow, provided/required interfaces compatibility, etc.). A developer must identify and specify how concerns with variability are treated when services are composed. The actual selection of variants in a large and complex SPL can be a tedious and error-prone task [13]. A third challenge is *to assist the user in selecting the right variant* for each family of services and for each dimension. These choices should be consistent for the entire workflow and should not violate the specified restrictions on concerns.

3 Modeling Concerns with Variability in Workflow

A medical imaging service should have high *variability*, that is, a user should be able to efficiently extend, change, customize or configure the service in a particular context [13]. The medical image format provides an example: some formats can anonymize the medical image by removing all patients metadata; some can compress and/or reduce the image size while the format header may differ. Services often must address several concerns. In this paper, the term *concern* is used in a broad sense and may range from high-level requirements to low-level implementation issues, refer to measurable properties or to the behaviour of the system.

Separation of Concerns (SoC). The number of variants and choices for each concern of a service can be extremely large and can be a threat to scalability:

variability descriptions quickly become too complex to manage, evolve or analyze by users. When modeling variants of a service, applying SoC principles and providing support to the modularisation of variability description can make them scale better. In our approach, the SoC is twofold. Firstly, concerns are associated with and described according to precisely defined elements of a service (e.g., Dataport). Secondly, we support the separation of variability models instead of the use of a large and monolithic variability model: The variability description of a service can be modularized, where each modular model focuses on a well-identified concern.

Modeling Workflow and Service. We first need to model what is a service, what its elements are and how services are assembled in workflows. Figure 2 shows a metamodel[1] that describes the form of services supported by our approach.

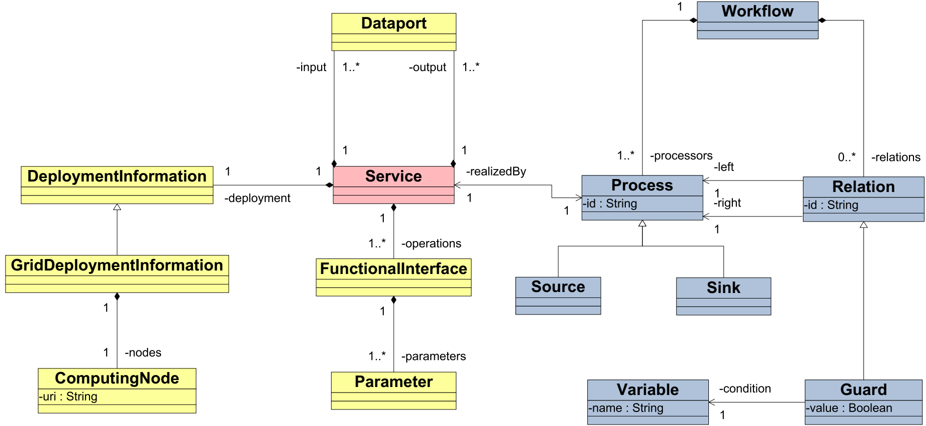


Fig. 2. Metamodel of Service and Workflow

A service describes data items in Dataports. Input dataports hold references to the data to be processed and output data ports contain references to the data produced by a service. FunctionalInterface represents the interfaces (legal operations with their parameters) exposed by a service. DeploymentInformation provides information about the deployment of the service, and a specialized GridDeploymentInformation references the ComputingNodes on which services are deployed. In addition, the metamodel describes how services can be connected in a workflow represented as a set of Processes. Two special processor nodes are also defined: Sources produce data to feed the workflow and Sinks collect the data produced. For the purpose of the paper, we consider that a Process is bound to one and only one Service in the sense that a service *realizes* a process. The connection between processes is specified as a partial ordering: The right part of

[1] While there exists more comprehensive metamodels for service descriptions, we chose to use a simple metamodel that shows only the service concepts needed to understand our approach.

Relation is a process that must wait for the end of the left part to start its own execution. If-then-else conditions can be expressed with a Guard which evaluates a predicate on objects of Variable.

Modeling Variability. A concern (e.g., *medical image format*) of a service can exhibit a set of variable points (e.g., alternatives, optionality). We choose to describe its variability with a Feature Model (FM). FMs are widely used to model a family (e.g., an SPL) in terms of common and variable features. Several definitions of *feature* appear in the literature, ranging from "anything users or client programs might want to control about a concept" [8] to "an increment in product functionality" [9]. These definitions indicate that FMs, like concerns, are not only relevant to requirement engineering but they can also be applied to design or implementation [14].

In Figure 3, a family of medical images is represented by a FM and has two mandatory features, ModalityAcquisition and Format, which imply that each valid configuration of a medical image should include these two features. An optional feature is Anonymized, which states whether all patients metadata of the medical image are included or not. There are also three alternatives of medical image format: Nifti, DICOM or Analyze features form a Xor-group. It means that at least and at most one feature must be selected. Finally, an MRI medical image has either the parameter T1 or T2 or both of them: T1 and T2 form an Or-group. A FM thus describes the set of valid feature combinations. Every member of a family is represented by a unique combination of features. For instance, each valid feature combination of a FM representing a family of requirements corresponds to an actual requirement. In the remainder of the paper, *a combination of selected features is called a configuration of a FM and is represented as a set of features*. In Figure 3, a valid configuration of the FM is as follows: {MedicalImage, ModalityAcquisition, Format, CT, DICOM}. A configuration is valid if all features contained in the configuration and the deselection of all other features are allowed by the semantics of FM [9,10].

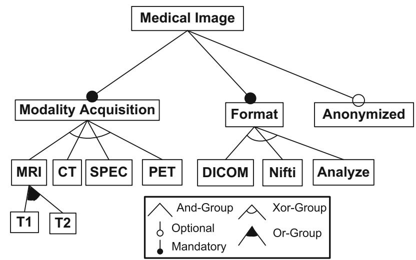


Fig. 3. Medical Image FM

Weaving Concern. Figure 4 describes how concerns which exhibit variability (called VariableConcerns) can be composed in a service description. We consider that a FM (resp. configuration) is an abstract view of a variable concern (resp. variant): A VariableConcern is described with a FeatureModel where each Configuration of a FeatureModel corresponds to a concrete Variant.

There is need to specify *where* the concern is inserted in the description of a service. A mechanism is required to weave a concern into a service model that conforms to the service metamodel of Figure 2. We propose that each model element of the service metamodel (e.g., Dataport, FunctionalInterface, etc.) can inherit from JoinPoint. Join points represent well-defined places in the structure of a service where additional behaviour can be attached. In our case, a

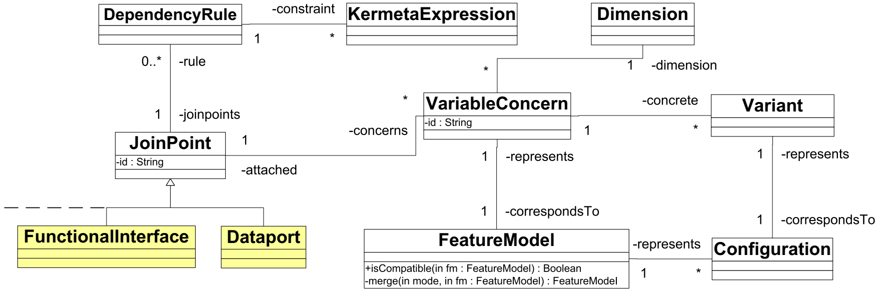


Fig. 4. Join Point Modeling

VariableConcern can be attached to any JoinPoint. For example, a concern dealing with the description of medical images supported by a service can be attached to the Dataport. Another concern can be attached to Dataport, e.g., to describe the security of data. It is also possible to describe how medical images will be stored on the grid. As a result, several concerns along several Dimensions can be associated to a JoinPoint. The actual weaving of a VariableConcern in a specific JoinPoint can be achieved using AOM composition approaches. In the rest of the paper, a dotted arrow which links a FM to a dashed border box means that we weave the FM into an instance of a JoinPoint. For example, in Figure 5 of Section 4, FM_{o1}, is woven to an actual output Dataport of $FService_1$. (Output Dataport is the name mentioned in the box and is a shortcut to name an instance of an output Dataport of a service.)

4 Reasoning on Workflow Concerns

Services are composed in the workflow while several concerns can be weaved into various elements of services. For some reasons, mainly due to the interconnection of services in the workflow, elements of services may be dependent. As a result, concerns attached to these elements may, in turn, be dependent on each other. This typically occurs when concerns belong to the same dimension.

Dependency Modeling. For instance, the medical image output format of a service S_1 is considered to be compatible with the medical image input format of another connected service S_2 in the workflow (see Figure 5). We need to express, at the model level, that an output Dataport of S_1 has to be compatible with an input Dataport of S_2 if they are to be connected. We define some DependencyRule(s), which are associated to JoinPoint elements of the service metamodel and that restrict in some way[2] the VariableConcerns.

The compatibility relation between two concerns vc_1 and vc_2 only considers concerns that belong to the same dimension. It is defined as follows: For at least one Variant of vc_1, there is an equal Variant in vc_2 (and vice-versa). The

[2] The use of constraints between FMs is not considered in the paper (see Section 6).

same relation applies to all dependency rules. In our implementation [15] of the approach, we formalize these rules using the Kermeta language [16] and the compatibility relation corresponds to the function *checkVariableConcerns*. **Rule 1** defines the compatibility between S_1 and S_2 Dataports as described above. It is expressed in Kermeta as follows:

```
s1.output.each{op |
    if op.connectInput.size > 0 then
        // some input ports (of service s2) are connected to op
        op.connectInput.each{ ip |
          // merge concerns of output port op and input port ip of s2
          checkVariableConcerns(op.concerns, ip.concerns)
        }
    end
}
```

Another rule, **Rule 2**, states that when two services are connected, the concerns associated to their FunctionalInterfaces are to be compatible. If a service S_1 does not support HTTP whereas the service S_2 only supports HTTP, users can be prevented from an inconsistency of service S_1 and service S_2. It is also possible to define dependency rules between concerns *without* considering the connection between services in the workflow. For instance, **Rule 3** states that if services are deployed on an equal computing node (a resource on the grid), all concerns must be compatible with each other. The concern can refer to the set of operating systems in which the services can be deployed and executed. In this case, if a service S_1 can only run on the Linux operating system whereas another service S_2 can only run on the BSD operating system, the two services cannot be deployed on the same computing node.

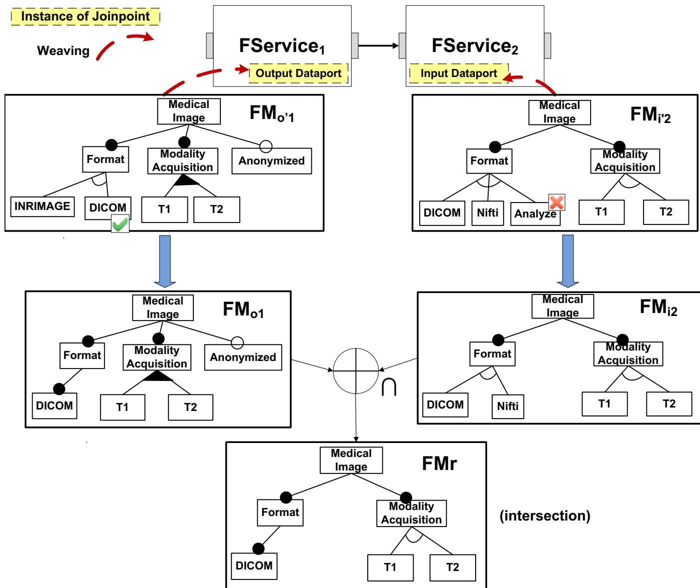


Fig. 5. Consistency checking concerns while shrinking variability choices

As a summary, DependencyRules express restrictions on the variants that can be derived from concerns attached to families of service elements. This boils down to ensuring the consistency of each set of Variant(s) associated with a Join Point of a Service. To implement the reasoning, we rely on our previous work, defining a set of composition operators for FMs. In [7], the semantics of each operator has been given in terms of the expressed configurations. Here, we focus on the merge operator which is dedicated to the composition of FMs that represent variable concerns along the same dimension.

Merge Operator. When two FMs share several features and are different viewpoints of a concern, the goal of the merge operator is to merge the overlapping parts of the two FMs to obtain an integrated model of the system. For example, users want to merge two medical image descriptions represented by two FMs, FM_{I_2} and FM_{0_1}, depicted in Figure 5. The merge operator uses a name-based matching: two features match if and only if they have the same name. The merge process starts from the root features of FM_{I_2} and FM_{0_1}. Medical Image of FM_{I_2} and FM_{0_1} match. Then, when two features have been merged, the whole process proceeds with their children features. Two modes are defined for the merge operator. The *intersection* mode is the most restrictive option: The merged FM, FMr, expresses the common valid configurations of FM_{I_2} and FM_{0_1}. The *union* mode is the most conservative option: the merged FM, FMr, can express either valid configuration of FM_{I_2} or valid configuration of FM_{0_1}. The variability information associated to features in the merged FM is set according to the defined rules. These rules (see [7] for more details) are different according to the merge mode and the properties that one may want to preserve.

We now formalize some properties of the merged FM with respect to the sets of configurations of input FMs. Let f be a FM and $[\![f]\!]$ denotes its set of configurations. The relationship between a merged FM $Result$ in intersection mode and two input FMs FM_1 and FM_2 is denoted $FM_1 \oplus_\cap FM_2 = Result$. It can be expressed in terms of sets of configurations:

$$[\![FM_1]\!] \bigcap [\![FM_2]\!] = [\![Result]\!] \quad (M_1)$$

According to the example of Figure 5, a valid configuration of the merged FM, FMr, is valid in FM_{I_2} and in FM_{0_1} at the same time. The DICOM feature is always part of any valid configuration of FM_{I_2} and FM_{0_1} whereas the Nifti feature cannot be part of any valid configuration of FM_{0_1}. As a result, DICOM is a mandatory feature of the merged FM while the Nifti feature is not part of the merged FM. The following relation can be shown to hold: $[\![FMr]\!] = [\![FM_{I_2}]\!] \bigcap [\![FM_{0_1}]\!]$.

In the union mode, we want to obtain a merged FM that represents the set of configurations of FM_{I_2} and FM_{0_1}. The merge operator in the union mode is denoted $FM_1 \oplus_\cup FM_2 = Result$. In the same way, we define the relationship between a merged FM $Result$ and two input FMs FM_1 and FM_2 in terms of sets of configurations:

$$[\![FM_1]\!] \bigcup [\![FM_2]\!] = [\![Result]\!] \quad (M_2)$$

Using FMs, the user can configure a family and thus derive an individual product (see Section 3). Observing that a FM in which there is no variability represents exactly one configuration, we decide to consider that a configuration of a FM *is* a FM. The rationale behind *considering configuration as an FM* is that it allows one to use the merge operator at each step of the configuration process.

Reasoning on Dependencies. In Figure 5, we focus on the medical image format concern. To illustrate our approach, we consider a very simple workflow where two processes are executed in sequence. $FService_1$ is connected to $FService_2$. FM_{O_1} (resp. FM_{I_2}) represents the medical image format information of $FService_1$ (resp. $FService_2$) and is associated to the output (resp. input) dataport. In this context **Rule 1** applies: The connection between $FService_1$ and $FService_2$ implies that FM_{O_1} and FM_{I_2} must be compatible. It is thus necessary to check if the set of configurations of FM_{O_1} is equal or included in the set of configurations of FM_{I_2} (and vice versa). Our technique is to compute the merge in intersection mode between FM_{0_1} and FM_{I_2}. If the merged FM does not represent an empty set of configurations, then there must be at least one configuration that is valid in FM_{O_1} and FM_{I_2}. The consistency checking can thus be achieved. In the example, such an FM exists (see FMr).

The benefits of computing the merged model are threefold. (1) The restriction on the concerns shrinks the variability choices in FM_{O_1} and FM_{I_2}. This restriction is represented by the merged FM. In this case, there is no longer need to consider the Nifti feature in FM_{I_2} or Anonymized feature in FM_{O_1}. (2) The user can use the merged FM to configure FMs of $FService_1$ and $FService_2$ at a time. One configuration of the merged FM corresponds to the same configuration in $FService_1$ and $FService_2$. For example, if the user selects $T1$ in the merged FM, then it implies that the feature $T1$ associated to $FService_1$ and $FService_2$ are also selected. (3) The merged FM can be the basis for reasoning on the compatibility with another FM. Let us now consider that $FService_1$ is also connected to another service $FService_3$ (see Figure 7(a), Section 5). The output dataport of $FService_1$ is thus dependent on the input dataport of $FService_3$. As a result, the new restriction on FM_{O_1}, represented by the merged FM, should be used to reason on the compatibility between $FService_1$ and $FService_3$.

5 Consistent Workflow Configuration

5.1 Impact of Workflow Constructs

We have seen in Section 4 how we can reason on two services that are connected. Workflows usually have more than two services executed in sequence, and others with parallel computations and branching through if-then-else constructs. It is necessary to ensure the consistency of concerns configurations considering the various workflow constructs (e.g., sequence, concurrency, condition).

Sequence. Figure 6 shows three services $FService_1$, $FService_2$ and $FService_3$ connected in sequence. In this example, checking each pair of connected services independently may not be enough. Let us address two situations illustrated in Figure 6.

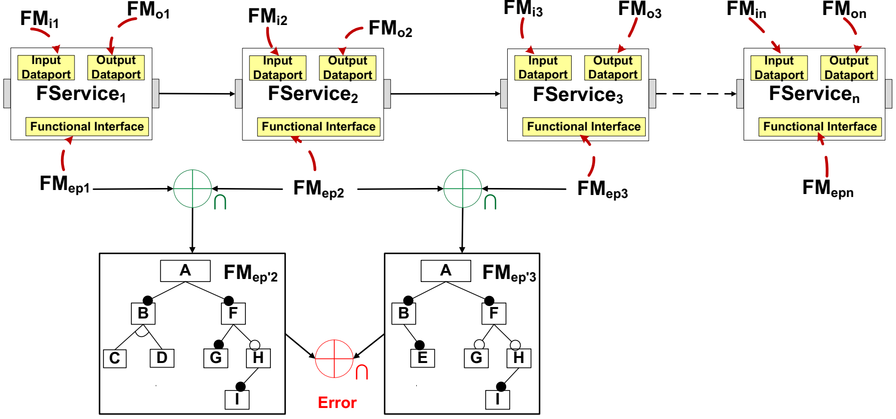


Fig. 6. Sequence of services

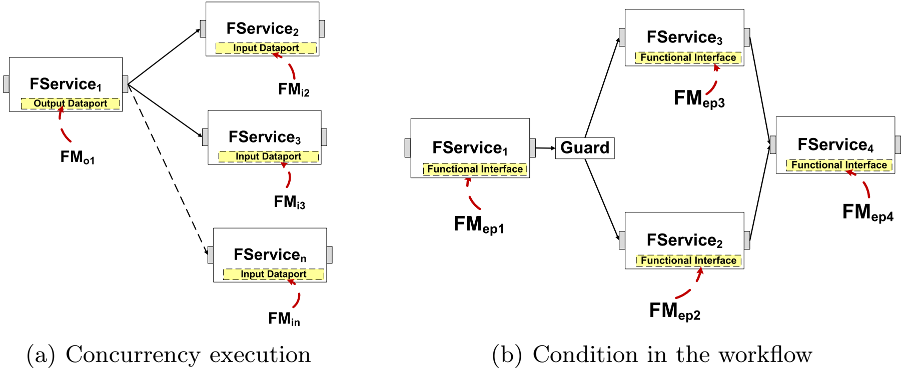
When the join point is the Dataport, the dependency between services is driven by **Rule 1**. The output dataport of $FService_i$, which is connected to the input dataport of $FService_{i+1}$, has to be compatible for $i \in 1...n$. In this case, the reasoning applies on a pair of services independently from the others. When the join point is the Functional Interface, **Rule 2** defined in Section 4 requires that the exchange protocol associated to $FService_i$ must be compatible with the ones of $FService_{i+1}$ for $i \in 1...n$. This requires the following checks to be made: *i)* FM_{ep1} and FM_{ep2} are consistent and also that *ii)* FM_{ep2} and FM_{ep3} are consistent. Let us now explain why it is necessary to reason globally on the entire sequence for this case. If we apply the same strategy as in Figure 5, we obtain:

$$FM_{ep'2} = FM_{ep1} \oplus_{\cap} FM_{ep2} \quad (a)$$
$$FM_{ep'3} = FM_{ep2} \oplus_{\cap} FM_{ep3} \quad (b)$$

However, the composition (a) has a side effect: some features of FM_{ep1} or FM_{ep2} may no longer be available. It is then possible that some features FM_{ep2} may not be involved in composition (b). Both compositions are therefore dependent on each other and should be addressed as a whole. Not following the above technique leads to an error, as shown in the bottom part of Figure 6: FM_{ep1} and FM_{ep2} are consistent as well as FM_{ep2} and FM_{ep3}, but the results of the two compositions are not compatible. We can generalize and state that: A sequence of $FService_1$, $FService_2$, ..., $FService_n$ is consistent according to a concern if and only if $SCR = ((FM_{ep1} \oplus_{\cap} FM_{ep2}) \oplus_{\cap} (FM_{ep2} \oplus_{\cap} FM_{ep3}) \oplus_{\cap} \ldots \oplus_{\cap} (FM_{ep(n-1)} \oplus_{\cap} FM_{epn})) \neq nil$, nil being the empty FM.

The merge operator properties M_1 and M_2 defined in Section 4 rely on the intersection or union of sets of configurations. In set theory, for the operations of intersection and union, associative, commutative and idempotent laws notably hold. The expression SCR can thus be simplified as follows:

$$SCR = (FM_{ep1} \oplus_{\cap} FM_{ep2} \oplus_{\cap} FM_{ep3} \ldots \oplus_{\cap} FM_{epn}) \neq nil$$



(a) Concurrency execution (b) Condition in the workflow

Fig. 7. Other Workflow Constructs

Concurrency. When two services are concurrently executed, the same situation occurs and it may not be sufficient to reason on pairs of services independently. In Figure 7(a), $FService_1$ is connected to $FService_2$ and $FService_3$ which are concurrently executed. The medical image output supported by $FService_1$ is described with FM_{o1} which is attached to Output Dataport. The medical image input supported by $FService_2$ (resp. $FService_3$) is described with FM_{i2} (resp. FM_{i3}) which is attached to Input Dataport. We are considering that the medical image of $FService_1$ is transmitted to $FService_2$ and $FService_3$ (an output Dataport of $FService_1$ is connected to an input Dataport of $FService_2$ and an input Dataport of $FService_3$). In this case, the **Rule 1** applies but, as in the previous example, it is not sufficient to independently check each pair of services. Ensuring the satisfiability of the following formula is not sufficient:

$$FM_{o1} \oplus_{\cap} FM_{i2} \neq nil \wedge FM_{o1} \oplus_{\cap} FM_{i3} \neq nil$$

since the restrictions on the set of configurations of FM_{o1}, due to the merge of FM_{o1} and FM_{i2}, are not considered when composing FM_{o1} and FM_{i3}. As a result, the following relation must hold:

$$FM_{o1} \oplus_{\cap} FM_{i2} \oplus_{\cap} FM_{i3} \neq nil$$

It can be extended to n concurrent services as follows:

$$FM_{o1} \oplus_{\cap} FM_{i2} \oplus_{\cap} FM_{i3} \ldots \oplus_{\cap} FM_{in} \neq nil$$

Condition. When a condition is present in a workflow, different execution paths can be followed. This impacts the way (variable concerns of) services are dependent and thus consistency checking must be adapted. In Figure 7(b), the connection between services in the workflow means that $FService_1$ is executed, followed by an if-then-else condition: If the condition is true (resp. false), then $FService_2$ (resp. $FService_3$) is executed. In addition, $FService_2$ and $FService_3$ are connected to $FService_4$. The **Rule 2** applies between $FService_1$ and $FService_2$, $FService_1$ and $FService_3$, $FService_2$ and $FService_4$, as well as $FService_3$ and $FService_4$. There are two alternative paths (mutually exclusive) considering the

execution flow: *i)* the execution of $FService_1$, then $FService_2$ and $FService_4$ or *ii)* the execution of $FService_1$, then $FService_3$ and $FService_4$. As a result, the following relation must hold:

$$P_1 = (FM_{ep1} \oplus_{\cap} FM_{ep2} \oplus_{\cap} FM_{ep4}) \neq nil \wedge P_2 = (FM_{ep1} \oplus_{\cap} FM_{ep3} \oplus_{\cap} FM_{ep4}) \neq nil$$

We propose to compute these restrictions as new FMs associated to each service. The new FM, $FM_{ep'1}$, associated to $FService_1$ is the union of the two paths in terms of sets of configuration: $FM_{ep'1} = P_1 \oplus_{\cup} P_2$. Then, the new FM, $FM_{ep'4}$, associated to $FService_4$ is also the union of the two paths in terms of sets of configuration: $FM_{ep'4} = P_1 \oplus_{\cup} P_2$. Finally, new FMs associated to $FService_2$ (resp. $FService_3$) are $FM_{ep'2} = P_1$ and $FM_{ep'3} = P_2$.

6 Assessment

Benefits and Strengths. If the variability manipulated by the user leads to some inconsistency but is considered to be more important than the workflow structure, the user has to correct the workflow itself. Using our approach, such inconsistencies can be systematically detected and several correction strategies can be applied. The separation of concerns provides the ability to precisely *locate the source of errors and to give information to assist users* in correcting the workflow. Hence, users can identify which specific services assembled in the workflow are causing inconsistency. In this case, a straightforward strategy is to choose another service. Another solution is to identify and correct inadequate concerns, either by relaxing some variability description of services or by configuring differently some services (e.g., choosing a feature instead of another in a Xor-group). Another option is to detect and solve the shimming problem [17] by introducing intermediary workflow processes, called *shims*, that act as adapters between otherwise incorrectly wired services. The implementation of shims can solely focus on the inadequate concerns previously detected.

Collaborative and distributive development can also be implemented, e.g., several grid and medical imaging experts can *independently* and *incrementally* configure services and associated concerns with respect to their know-how. The merge operator deals with synchronizing choices and guarantees their coherence at each step.

Properties of the merge operator can then be exploited. The various compositions of FMs may be performed in any order because of the associativity property of the merge operator. Heuristics, such as merging larger FMs first, can thus be planned to detect an earlier source of errors. The idempotent and commutative properties can reduce the number of merge calls: In Figure 6, for n services sequentially executed, there are $n-1$ calls before simplification instead of $2*n-1$. The merge between FMs contributes to decrease the number of remaining variability choices.

An additional property of the merge in intersection mode is as follows: The number of features of the resulting FM is lesser than or equal to the number of features commonly shared by input FMs. This property can dramatically reduce the set of configurations to be considered by the user during workflow configuration. As a result, the *amount of time and effort needed during the configuration*

process can be reduced. For instance, let us consider that the average of the number of features of each FM FM_{o1}, FM_{i2}, FM_{i3} and FM_{in} of Figure 7(a) is 30 and the number of features commonly shared by FMs is 20. After applying the merge operator, the new computed FM has a number of features which is necessary lesser than or equal to 20. As a result, a user only has to consider less than 20 features instead of $30 * 4 = 120$ features. Such benefits can also be observed for other workflows.

Current Limits and Threats. Currently we do not handle constraints between FMs whether they are internal or between several FMs. This is useful when concerns related to FMs are not independent, e.g., the QoS provided by a medical imaging service can be dependent on the kind of input images manipulated. More generally, the *feature interaction* problem is still an open and hard research challenge [14]. Constraints between FMs and feature interactions are threats to incremental and modular development, as well as to independent reasoning on FMs. They may cancel out some of the benefits presented above.

In the current proposal, we make the assumption that FMs to be merged have the same granularity, e.g., they share the same hierarchical structure. Given the open nature of the grid and the autonomy of the data and service providers, users may want to *align* concepts (features) and/or to negotiate some parts of FMs that are not present in another like in the viewpoints approach.

7 Related Work

Feature models. A few other approaches use multiple FMs during the SPL development. In [18], separate FMs are used to model decisions taken by different stakeholders or suppliers. The authors recognize the need to compose and merge FMs during multi-stage and multi-step configuration process, but do not achieve it. In [19], several FMs are used to separate feature descriptions related to requirements, problem world context and software specifications. Constraints then inter-relate features of FMs. Metzger et al. proposed a formal approach for separating PL variability (e.g., economical-oriented variability) and software variability, thereby enabling automatic analysis [20]. The two kinds of variability can be considered as concerns of an SPL. Previous contributions do not consider FMs or concerns that are sharing some features. This can happen when concerns along the same dimension interact, when multiple perspectives on a concern needs to be managed or when SPLs are composed with SPLs. A few works [10,21,22,23] suggested the use of a merge operator: Our proposal goes further in this direction and clarifies the semantics of the merge and, most importantly, shows how this operator can be used in practice. In [24], an algorithm is designed to compute the kind of relations between two FMs. We have shown that reasoning on more than two FMs can happen for some constructs of the workflow and then, why the merge operator is required. In [25], the configuration process is represented as a workflow and different stakeholders are configuring the same FM. The first difference with our work is that the term workflow used in the approach does not refer to a processing pipeline, but to the activities completed during configuration. The second difference is that only a single FM is considered during the whole configuration process.

Multiple SPLs. In our case study, a medical imaging service can be seen as an SPL provided by different researchers or scientific teams. The entire workflow is then a multiple SPL in which different SPLs are composed. In many domains, organizations or architectures, the need to "shift from variation to composition" and to support multiple SPLs (also called product populations) is more and more patent [26]. Van der Storm considered not only variability at the level of one software product, but also each variable component as an entry-point for a certain software product (obtained through component composition) [27]. Hartmann and Trew dealt with multiple SPLs and identified several compositional issues in the context of software supply chains. They notably recognized that "merging FMs, especially when they are overlapping, requires a significant engineering activity" [23]. They did not provide a set of operators, a semantics nor a mechanism to automate this task. Reiser and Weber proposed to use multi-level feature trees consisting of a tree of FMs in which the parent model serves as a reference FM for its children [28]. Their purpose is mostly to cope with large diagrams and large-scale organizations, rather than different concerns.

Service composition. A large amount of work exists in (automatic) service composition (e.g. see [29]). To the best of our knowledge, there is no specific approach combining separation of concerns while managing variability in the same kind of context. In [30], AO4BPEL promotes a well-modularized specification of concerns and dynamic strategy for web service composition. Our work focuses on how to ensure in a processing chain, at design time, consistency between concerns with respect to variability. Work in [31] focused on how to map a FM to a business process model described in BPEL; each feature of a FM corresponds to a business process. The motivation of our work is rather to describe the variability within a process; we also consider that the processing chain is fixed.

8 Conclusion and Future Work

Creating workflows from many different kinds of highly parameterized services is a cumbersome and error-prone task as important variabilities have to be managed by the user. In this paper, we have presented an approach that organizes services as a product line architecture and that uses feature models (FMs) to structure necessary information in terms of service variabilities. In the proposed approach, a family of services is defined as a set of concerns which exhibit variability, each being represented with one or several FMs. To reason on these artifacts, we rely on several related metamodels, reifying services, their dependencies in workflows and the join point related concepts. To enable the multiple composition of the concerns while taking variability into account, we have proposed a set of composition operators. Using these operators, we have defined consistency rules that enable the reasoning about the compatibility between families of connected services. Moreover, the consistency checking process makes it possible to automatically propagate variability choices into the whole workflow and thus to assist the user in selecting tailor-made services.

Future work aims at tackling current restrictions: *i)* handling inter- or intra-constraints between FMs in the composition process; *ii)* providing mechanisms to enable users to align FMs. The building of a large SPL dedicated to medical imaging services on the grid has already started. The services are part of a service-oriented architecture in which data-intensive workflows are built to conduct numerous computations on very large sets of images [4, 5]. The construction of such an SPL gives us an opportunity to obtain validation elements and feedback on the approach.

References

1. Taylor, I., Deelman, E., Gannon, D., Shields, M.: Workflows for e-Science. Springer, Heidelberg (2007)
2. Vigder, M., Vinson, N.G., Singer, J., Stewart, D., Mews, K.: Supporting scientists' everyday work: Automating scientific workflows. IEEE Software 25, 52–58 (2008)
3. McPhillips, T., Bowers, S., Zinn, D., Ludäscher, B.: Scientific workflow design for mere mortals. Future Generation Computer Systems 25(5), 541–551 (2009)
4. Acher, M., Collet, P., Lahire, P.: Issues in Managing Variability of Medical Imaging Grid Services. In: Olabarriaga, S., Lingrand, D., Montagnat, J. (eds.) MICCAI-Grid, New York, NY, USA, p. 10 (2008)
5. Acher, M., Collet, P., Lahire, P., Montagnat, J.: Imaging Services on the Grid as a Product Line: Requirements and Architecture. In: Service-Oriented Architectures and Software Product Lines (SOAPL 2008) workshop at 2008. IEEE, Los Alamitos (2008)
6. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
7. Acher, M., Collet, P., Lahire, P., France, R.: Composing Feature Models. In: Gašević, D. (ed.) SLE 2009. LNCS, vol. 5969, pp. 62–81. Springer, Heidelberg (2010)
8. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
9. Batory, D.S.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
10. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. 51(2), 456–479 (2007)
11. Germain, C., Breton, V., et al.: XIX. In: Grid Analysis of Radiological Data. Handbook of Research on Computational Grid Technologies for Life Sciences, Biomedicine and Healthcare, p. 30 (2009)
12. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, GGF (June 2002)
13. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. Journal of Systems and Software 74(2), 173–194 (2005)
14. Apel, S., Kästner, C.: An overview of feature-oriented software development. Journal of Object Technology (JOT) 8(5), 49–84 (2009)
15. `http://modalis.polytech.unice.fr/softwares/manvarwor`
16. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)

17. Lin, C., Lu, S., Fei, X., Pai, D., Hua, J.: A task abstraction and mapping approach to the shimming problem in scientific workflows. In: SCC 2009: International Conference on Services Computing, pp. 284–291. IEEE, Los Alamitos (2009)
18. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice 10(2), 143–169 (2005)
19. Tun, T.T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P.: Relating requirements and feature configurations: A systematic approach. In: SPLC 2009, pp. 201–210. IEEE Computer Society, Los Alamitos (2009)
20. Metzger, A., Pohl, K., Heymans, P., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: RE 2007, pp. 243–253 (2007)
21. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: GPCE 2006, pp. 201–210. ACM, New York (2006)
22. Segura, S., Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated merging of feature models using graph transformations. Post-proceedings of the Second Summer School on GTTSE 5235, 489–505 (2008)
23. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: SPLC 2008, pp. 12–21. IEEE, Los Alamitos (2008)
24. Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In: ICSE 2009. IEEE Computer Society, Los Alamitos (2009)
25. Hubaux, A., Classen, A., Heymans, P.: Formal modelling of feature configuration workflows. In: SPLC 2009, pp. 221–230. IEEE Computer Society, Los Alamitos (2009)
26. van Ommering, R., Bosch, J.: Widening the scope of software product lines - from variation to composition. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 328–347. Springer, Heidelberg (2002)
27. van der Storm, T.: Variability and component composition. In: Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, pp. 157–166. Springer, Heidelberg (2004)
28. Reiser, M.O., Weber, M.: Multi-level feature trees: A pragmatic approach to managing highly complex product families. Requir. Eng. 12(2), 57–75 (2007)
29. Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web Grid Serv. 1(1), 1–30 (2005)
30. Charfi, A., Mezini, M.: AO4BPEL: an aspect-oriented extension to BPEL. World Wide Web 10(3), 309–344 (2007)
31. Schnieders, A., Puhlmann, F.: Variability modeling and product derivation in E-Business process families. In: Technologies for Business Information Systems, pp. 63–74. Springer, Heidelberg (2007)

Composition and Compositionality in a Component Model for Autonomous Robots

Olena Rogovchenko and Jacques Malenfant

Université Pierre et Marie Curie-Paris 6, CNRS, UMR 7606 LIP6,
104 avenue du Président Kennedy, F-75016 Paris, France
{Olena.Rogovchenko,Jacques.Malenfant}@lip6.fr

Abstract. Component models for autonomous robots control architectures are much more constrained than traditional ones: obeying strict timing constraints, coping with a large spectrum of rapidly changing hardware (e.g. sensors and actuators), etc. Beyond introducing new concepts into components themselves, composition in such models must go much farther than the standard connection through method signature interfaces. Viewing components as full-fledged sensori-motor behaviors, our model follows the concept of rich interfaces introduced by Henzinger *et al.* to attach to each component all the necessary syntactical and behavioral information to make them externally composable. This paper presents two kinds of composition, parallel and by modes, their semantics, their compositionality properties and the impact of these on the composition model. A prototype implementation in Java is backed by a constructive semantics defined as a constraint system solved in this prototype with the ECLIPSe constraint programming system.

1 Introduction

As autonomous robot control architectures grow ever more complex, means to build them incrementally from existing pieces of software are becoming a key issue. Component-based architectures appear as a promising approach, yet no component model deals with all the constraints from autonomous robotics such as obeying strict timing constraints, mixing computationally intensive computations with hard real-time tasks, taking into account a large spectrum of rapidly changing hardware such as sensors and actuators and allowing for coordination with other autonomous robots. This paper addresses the impact of these constraints on the component definition as well as the kinds of compositions to which they can be submitted, and their semantics and implementation.

In our model [1,2], components strive to represent full-fledged sensori-motor behaviors of autonomous robots. For example, simultaneous localisation and mapping (SLAM), a crucial functionality, intertwines distance and positioning measures as well as obstacle identification done in real-time with map construction as a non real-time computation; this map is also used to localize and control movements of the robot in real-time. To implement such sensori-motor behaviors, the model mixes reactive components for real-time and active components for non real-time computations, the two communicating through asynchronous

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 34–49, 2010.
© Springer-Verlag Berlin Heidelberg 2010

message passing. Components provide for an explicit representation of precedence, timing, communication and hardware related constraints by declaring in interfaces all the necessary information to make them externally composable.

The challenge addressed in this paper is the definition of a composition model that produces correct-by-construction deployable reactive component assemblies from these declarative interfaces. To do so, formal composition rules as well as generative programming techniques are developed. Because of the heterogeneity of the deployment contexts (monoprocessor, multiprocessor, networking, etc.), and the domain-specific needs, the composition model aims to offer different composition operators as well as different context-dependent variants, as many of these, like the concurrent one, take different meanings depending on the deployment context. One goal of the composition model is to abstract the programmer away from these to make components more reusable in different contexts.

The rest of the paper is organized as follows. Section 2 presents the background on the current approaches to component models for autonomous robotics, as well as issues related the building of correct control architectures. The section 3 introduces the proposed component model, and then develops the composition model itself as the building blocks for a future full-fledged architecture definition language. The section 4 presents the different composition operators included in the model to date, and ends with a short description of a prototype implementation. Related work (§5) and conclusions are then discussed.

2 Components and Composition for Autonomous Robots Control Architectures

This section presents the required background on component composition.

2.1 Rich Interfaces for Component Composition

Viewing components as sensori-motor behaviors raises central issues when composing for control architectures: schedulability of real-time tasks, resource sharing among behaviors using the same sensors and actuators, etc. Current component models cannot address these issues because traditional interfaces do not represent such behavioral and timing constraints.

Acknowledging the central role of interfaces, de Alfaro and Henzinger [3] have proposed the concept of *rich interfaces* to capture whole aspects of component interactions at their boundaries. In concurrent programming, synchronisation comes along on top of more traditional method signatures concerns. In real-time programming, time plays a central role, introducing temporal dependencies among pieces of code, and time constraints (worst-case execution time, maximal and minimal delays between pieces of code producing and consuming data, frequencies in the triggering of readings on sensors or writing on actuators, etc.). Robotic control architectures exhibit all of these aspects, requiring therefore a much richer model of interfaces than the one of traditional components.

Rich interfaces, at large, expose all the necessary and sufficient information to verify if components can be composed (compatibility) and then to compose them.

The precise definition of component compatibility depends upon the component model and the properties defined in its rich interfaces. Beyond traditional syntactic compatibility, behavior [3], timing [4] and resources consumption [5] have been considered. Beyond the predictability of assemblies, rich interfaces allow for the incremental design of applications as well as independent implementation of components, two of the strengths of component-based approaches.

The introduction of required and offered rich interfaces allows compositions to make early decisions based on timing, resources and hardware requirements for the deployment context expressed in the required interfaces of components. At deployment-time, the validation of these decisions, and therefore of the assembly, amounts to checking the compatibility of the required interfaces of the assembly with the offered interfaces of the containers.

2.2 Composition Semantics and Compositionality

Real-time programs are built from elementary tasks with bounded execution times that execute periodically under strict constraints. Tasks also produce and consume data, or signals. Hence, the composition semantics must take several aspects into account:

- communication, by producing connections to send and receive signals;
- scheduling, by producing a schedule that respects all of the precedence and timing constraints of the different components;
- resource sharing, by synchronizing code accessing them; and
- hardware dependencies, by connecting abstract required hardware interfaces to available (offered) concrete ones.

Scheduling is in fact concerned with several other aspects of composition and plays a central role. Signals introduce precedence constraints between producer and consumer tasks, which can be solved by ordering them correctly in the schedule. Tasks can explicit the resources they use, and provided that they release the acquired resources before they finish, synchronization amounts to mutual exclusion constraints among them to be observed by the schedule. Besides ensuring the compatibility of the actual hardware to the required one, hardware dependencies also introduce timing constraints when several sensori-motor behaviors need the same sensors and actuators but at different times and frequencies. The scheduling of sensor readings and actuator writings must provide for compatible frequencies among the different users while respecting the maximal and minimal frequency constraints imposed by the hardware.

Scheduling challenges the definition of a compositional semantics for composition operators. The semantics of a construct is compositional if it is directly constructed from the semantics of its subconstructs. Compositionality enables proofs by structural induction. Pragmatically, it also eases the production of reusable deployable components by simplifying their late-composition. Non-compositionality arises when the semantics of subconstructs influences directly one another. Proofs then become much more involving, since they must proceed from subconstructs to subconstructs along the dependencies. Processor

time management is the key issue when composing over schedules. As components in assemblies compete for the access to processors, and therefore depend on each others scheduling decisions, compositionality is hard to obtain. A possible solution is to preallocate processor time slots to the components, but with the *caveat* of suboptimal processor usage and overconstrained feasible assemblies.

3 The Composition Model

This section introduces the proposed component and composition model.

3.1 Component Model

Reactive components are periodic, executing repeatedly the same code with a period p so to respect their time constraints. Their code consists of elementary pieces called *codels* (*code elements*). Codels have a bounded execution time (WCET), explicit communication, and no synchronization instruction to ensure that they can be scheduled to run to completion when activated. All the precedence, time and resource usage constraints of codels must be provided explicitly:

- a precedence constraint between two codels indicates that one codel must finish its execution before the second can start;
- a minimum (resp. maximum) delay constraint between two codels indicates that a minimum (resp. maximum) delay must be enforced between the finish time of the first codel and the start time of the second;
- a minimum (resp. maximum) frequency constraint on a codel indicates that a maximum (resp. minimum) delay must be enforced between two successive activations of the codel;
- a resource usage constraint on a codel indicates that the codel access that resource during its execution; if the resource is not sharable, two codels using the same resource will have a mutual exclusion constraint between them;
- a communication constraint between two codels establishes a signal producer/consumer relationships between them.

More formally, a *reactive component* is a tuple $\mathcal{C}(n, \Pi, cdls, csts, R)$ where

- $n \in I_{\mathcal{C}}$ is a unique component identifier,
- Π is the set of communication ports, themselves tuples $\pi(n, \tau, i_c, \delta)$ where
  - $n \in I_\pi$ is a unique port identifier,
  - $\tau \in \{in, out\}$ is the kind of port,
  - i_c is a communication interface,
  - $\delta \in \{int, float, ...\}$ is the type of data transmitted;
- $cdls$ is a set of codels, themselves tuples $\kappa(n, \sigma_{in}, \sigma_{out}, w, \rho)$ where
  - $n \in I_\kappa$ is a unique codel identifier,
  - σ_{in} and σ_{out} are the sets of input and ouput signals respectively,
  - $w \in \mathbb{N}^+$ is the worst-case execution time of the codel (or $WCET$),
  - ρ is a set of resources used by the codel, themselves tuples $\rho(n, \mathcal{T})$ where
    - $n \in I_\rho$ is a unique resource identifier,
    - $\mathcal{T}$ is the resource type.

  with $\sigma_{in} \cap \sigma_{out} = \emptyset$,

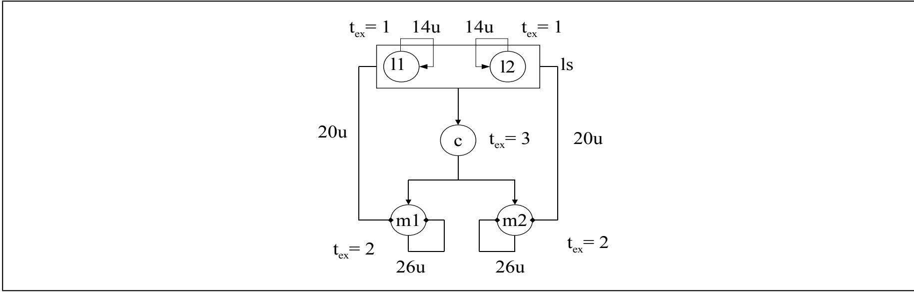


Fig. 1. A graph representation of the exemple : the nodes represent the codels and the arcs the time constraints. The non-valued arcs represent precedence dependencies between codels. The worst case execution times t_{ex} are attached to the nodes.

- $csts$ is a set of constraints, themselves tuples $cst(ct, n_s, n_t, e)$ where
  - $ct \in \{temporal, timed, ...\}$ is the type of constraints,
  - $n_s \in I_\kappa$ is the source codel identifier,
  - $n_t \in I_\kappa$ is the target codel identifier,
  - e is the constraint expression, depending upon the type of constraint;
- R is the resource interface, itself a tuple $R(procs, ct)$, where
  - $procs \in \mathcal{P}(I_{Procs})$, is a set of processor identifiers,
  - $ct : I_{Procs} \times I_{Procs} \rightarrow \mathbb{N}$, the worst-case communication times.

Signals are tuples $\sigma(n, \delta)$ where

- $n \in I_\sigma$ is a unique signal identifier,
- $\delta \in \{int, float, ...\}$ is the type of data transmitted;

Example. *The robot must follow a black line on a clear surface. At each cycle, it reads two sensors for reflected light, and correct its trajectory accordingly using the two motors (see also Figure 1). The graph shows examples of precedence (e.g. between c and m1) and timed constraints (e.g. maximum frequency on m1 or maximum delay between l1 and m1). In the formal model, it gives:*

$\mathcal{C}(LineFollower,$	
$\{\ \kappa(l1, \emptyset, \{\sigma_1\}, 1, \{\rho_1\}),$	codel $l1$, 1st light sensor
$\kappa(l2, \emptyset, \{\sigma_2\}, 1, \{\rho_2\}),$	codel $l2$, 2nd light sensor
$\kappa(c, \{\sigma_1, \sigma_2\}, \{\sigma_3, \sigma_4\}, 3, \emptyset),$	codel c, computation module
$\kappa(m1, \{\sigma_3\}, \emptyset, 2, \{\rho_3\}),$	codel $m1$, actuator, 1st motor
$\kappa(m2, \{\sigma_4\}, \emptyset, 2, \{\rho_4\})\},$	codel $m2$, actuator, 2nd motor
$\emptyset,$	no communication ports
$\{\ cst(temporal, l1, c, \cdot),$	$l1$ produces the signal $s1$ consumed by c
$cst(f_max, l1, 14),$	$l1$ may be launched at most once every 14 units
...	...
$R(\{p1, p2\}, \cdot)$	a list of processors available to the component

$$\frac{inBounds(p, \mathcal{IC}) \qquad ValidSched(\mathcal{IC})}{ValidCompInst(\mathcal{IC}(\cdot,\cdot,\cdot,\cdot,\cdot,p,\cdot,\cdot))} \tag{1}$$

$$\frac{(\forall i\kappa(n,o,s,f) \in iCodels(\mathcal{IC}))\ p \geq f}{inBounds(p, \mathcal{IC})} \tag{2}$$

$$\frac{\begin{array}{l}(\forall cst \in csts)\ IsValid(cst, icdls) \\ \quad (\forall i\kappa \in icdls)\ Proc(i\kappa) \in procs \wedge CValid(i\kappa, cdls, n_mc) \\ \qquad (\forall i\kappa(n,o,s,f), i\kappa(n',o',s',f') \in icdls)\ n \neq n' \Rightarrow o \neq o'\end{array}}{ValidSched(\mathcal{IC}(\cdot,\cdot,cdls,csts,R(procs,f),\cdot,n_mc,icdls))} \tag{3}$$

$$\frac{(\exists!\, \kappa(n',\cdot,\cdot,wcet,\cdot) \in cdls)\ n = n' \ \wedge\ f = s + wcet \ \wedge\ no \leq n_mc}{CValid(i\kappa(n,s,f,no,p), cdls, n_mc)} \tag{4}$$

Fig. 2. Single deployable component instance

where $\sigma_1 = \sigma(s1, int)$ *and* $\sigma_2 = \sigma(s2, int)$ *signal the intensity of light read while* $\sigma_3 = \sigma(s3, int)$ *and* $\sigma_4 = \sigma(s4, int)$ *send out an order to the motors in terms of a rotation angle, and the ressources are* $\rho_1 = \rho(sens_l1, light_sensor)$, $\rho_2 = \rho(sens_l2, light_sensor)$, $\rho_3 = \rho(act_m1, motor)$ *et* $\rho_4 = \rho(act_m2, motor)$.

When ready to be deployed, a *reactive component instance* is a tuple $\mathcal{IC}(n, \Pi, cdls, csts, R, p, n_mc, icdls)$ which adds the following to the component:

- $p \in \mathbb{N}^+$ is the period
- $n_mc \in \mathbb{N}^+$ is the number of microcycles this component needs to match the period of the embedding component (equals to 1 if not embedded),
- $icdls$ is a set of codel instances, themselves tuples $i\kappa(n, o, s, f, \rho)$ where
  - $n \in I_{cdl}$ is the unique identifier of the instantiated codel,
  - $o \in \mathbb{N}^+$ is the occurrence number of the codel in the macro-cycle,
  - $s \in \mathbb{N}^+$ is start time of the codel within the component period,
  - $f \in \mathbb{N}^+$ is finish time of the codel within the component period,
  - ρ is the set of ressources used by the codel.

The algorithm to construct a component instance from the more abstract component description is implementation-dependent. On the other hand, formal rules can express the constraints and properties that a valid component instance must obey. For the sake of conciseness, the rules of Figure 2, 3, 5 and 6 concentrate on behavioral properties rather than syntactic correctness ones like the unicity of identifiers and so on. Checking the validity of a component instance then amounts to prove:

$$(\exists p)(\exists n_mc)(\exists icdls)\ ValidCompInst(\mathcal{IC}(n, \Pi, cdls, csts, R, p, n_mc, icdls))$$

A constructive proof for this sentence provides for a valid deployable component instance, where p, n_mc and $icdls$ are found so to satisfy the constraints.

$$\frac{(\forall i\kappa_1(n_1, o, s_1, f_1, \cdot), i\kappa_2(n_2, o, s_2, f_2, \cdot) \in icdls)\ s_1 < s_2}{IsValid(prec(n_1, n_2, \cdot), icdls, \cdot)} \tag{5}$$

$$\frac{(\forall i\kappa_1(n_1, o, s_1, f_1, \cdot), i\kappa_2(n_2, o, s_2, f_2, \cdot) \in icdls)\ s_1 + d \leq s_2}{IsValid(max_delay(n_1, n_2, d), icdls, \cdot)} \tag{6}$$

$$\frac{(\forall i\kappa_1(n_1, o, s_1, f_1, \cdot), i\kappa_2(n_2, o, s_2, f_2, \cdot) \in icdls)\ s_1 + d \geq s_2}{IsValid(min_delay(n_1, n_2, d), icdls, \cdot)} \tag{7}$$

$$\frac{(\forall i\kappa_1(n, o_1, s_1, f_1, \cdot), i\kappa_2(n, o_1', s_1', f_1', \cdot) \in icdls)\ o_1 + 1 = o_1' \ \wedge\ s_1 + d \geq s_1'}{IsValid(min_freq(n, \cdot, d), icdls, \cdot)} \tag{8}$$

$$\frac{(\forall i\kappa_1(n, o_1, s_1, f_1, \cdot), i\kappa_2(n, o_1', s_1', f_1', \cdot) \in icdls)\ o_1 + 1 = o_1' \ \wedge\ s_1 + d \leq s_1'}{IsValid(max_freq(n, \cdot, d), icdls, \cdot)} \tag{9}$$

$$\frac{\begin{array}{r}(\forall i\kappa_1(n_1, o, s_1, \cdot, p_1), i\kappa_2(n_2, o, s_2, \cdot, p_2) \in icdls)\ s_1 + f(p_1, p_2) \leq s_2 \\ IsValid(prec(n_1, n_2, \cdot), icdls, R(procs, f))\end{array}}{IsValid(comm(n_1, n_2, \cdot), icdls, R(procs, f))} \tag{10}$$

Fig. 3. Constraints for a valid schedule

3.2 Composition and Its Semantics

A reactive component can be executed either on a single processor or on multiple processors, such as modern multicores. To enable external composition, codel identifiers, precedence, timed and resource usage constraints forms rich interfaces used in component composition. Altogether, this information forms what is called the *membrane* of the component. When bound to the actual codels, provided with a schedule for running them and bound to the required hardware, a *component instance* is obtained, ready to be deployed. A component instance is valid if it is bound to suitable resources such as required sensors and actuators, and if its execution period p as well as the start and finish times of all of its codel constitutes a valid schedule with respect to the component constraints.

When composed with other reactive components with different execution periods, a unique period must be sought for the composite. Hence, several replicated execution periods of a component may need to be unfolded to match side-by-side the periods of other components and form a composite component with a unique overall execution period. The period of each composed component is called its *microcycle period*, while the overall period of the composite is called its *macrocycle period*. Each inner component has its own *number of microcycle occurrences* in the composite macrocycle, which also corresponds to the number of occurrences of its codels in the composite macrocycle.

3.3 Operator Hierarchies

One goal of the composition model is to abstract the programmer away from these to make components reusable in different contexts. Operators are therefore

$$\frac{(\exists period \in \mathbb{N})\ (samePeriod(period, e_1) \wedge samePeriod(period, e_2)) \quad compose(e_1 \mid\mid e_2 : cst)}{globalCompose(e_1 \mid\mid e_2 : cst)} \tag{11}$$

$$\frac{(\exists period \in \mathbb{N})\ (\forall i \in [1,n])(samePeriod(period, e_i) \quad compose(\texttt{mode}\ e^* : cst)}{globalCompose(\texttt{mode}\ e^* : cst)} \tag{12}$$

$$\frac{samePeriod(period, e_1) \quad samePeriod(period, e_2)}{samePeriod(period, e_1 \mid\mid e_2)} \tag{13}$$

$$\frac{(\forall i \in [1,n])\ samePeriod(period, e_i)}{samePeriod(period, \texttt{mode}\ [e_1, ..., e_n])} \tag{14}$$

$$\frac{period = p \times n_mc}{samePeriod(period, \mathcal{IC}(\cdot,\cdot,\cdot,\cdot,\cdot,p,n_mc,\cdot))} \tag{15}$$

$$\frac{Par(e_1, e_2, cst))}{compose(e_1 \mid\mid e_2 : cst)} \qquad \frac{Mode(e^*, cst)}{compose(\texttt{mode}\ e^* : cst)} \tag{16}$$

Fig. 4. Composition expressions

organized in hierarchies, which specialization goes from context-independent operators used by programmers to context-dependent ones. The derivation of the latters from generic ones is done automatically in the process of producing deployable assemblies. For exemple, a generic concurrent composition will be turned into a multiprocessor one and then a distributed and furthermore a tightly-coupled one if the deployment context offers many processors interconnected through a time-guaranteed network.

4 Composition Operators

This section presents the composition operators included in the model to date.

4.1 Composition Expressions and Composites

Component instances are composed to form composites using composition expressions which current abstract grammar is (ic is a component instance):

$$e ::= e \mid\mid e\ :\ cst\ \mid\ \texttt{mode}\, e^*\ :\ cst\ \mid\ ic$$

In this paper, only parallel and mode compositions are addressed. Each of these can impose composition constraints cst (communication, precedence, time, resource sharing, ...). Parallel compositions can be done over a single processor, or over multiple processors with known communication delays. Mode composition allows for the definition of composites with mutually exclusive operating modes represented by subcomponents and triggered by internal state transitions.

In the current compositions, the same macrocycle period is imposed to all components to provide for the satisfaction of inter-component communication and

resource sharing cross-constraints. Hence, as shown by the composition rules 11 and 12, verifying the composition amounts to impose the same period (see rules 13 to 15) and then to apply the composition rule for the operator (see Figure 5), which are detailed in the next subsections.

Although rules 13 to 15 explicit the constraints that must be observed, finding a common period is far from trivial. In general, this can be done in two steps: (1) computing lower and upper bounds on the period of each of the components and then (2) determining of a composite execution cycle length. Deduced from its constraints, the duration of each component microcycle can be expressed as an interval $l_{cyc} = [d_{min}, d_{max}]$, where $d_{max}, d_{min} \in [0, \propto]$ and $d_{max} \geq d_{min}$. The cases $d_{min} = 0$ and $d_{max} = \propto$ represent the absence of constraints on the minimum and maximum cycle lengths respectively. The case $d_{min} = d_{max}$ corresponds to a unique possibility for the cycle length. The period length of a component is a multiple of the number of clock ticks and is therefore always an integer. The macrocycle length of the composite is deduced from its subcomponents microcycle lengths. For two components A and B, two cases are possible (the idea generalizes to n components):

1. There exists a macrocycle length $d_{macro} \in \mathbb{N}^+$ such that :
   - $d_{macro} > d_{ex}(A) + d_{ex}(B)$ where d_{ex} is the cycle length,
   - $d_{macro} \in l_{cyc}(A)$ and $d_{macro} \in l_{cyc}(B)$,
   - there exists a combined schedule for A and B that fits inside the time allotted for the macrocycle and respects all the individual component constraints as well as those imposed by the composition.
2. If such a cycle length does not exist, a combination of n cycles of A and of m cycles of B, such that it satisfies the three conditions above and the constraints of A and B when unfolded (repeated) n and m times respectively to form a macrocycle, must be found.

To compute the length of the macrocycle, microcycle lengths for A and B must be chosen and the number of microcycles in the macrocycle must be calculated. The cycle length for a component, whether simple or composite, is represented by an interval, that contains one or more acceptable integer bounds, as long as the composition for that component is not completed. At the end, a unique cycle length for that component is set and a concrete schedule is found.

4.2 Concurrent Composition: The Monoprocessor Case

Given that a common macrocycle period has been imposed globally on the composite, applying a parallel composition amounts to the handling of subcomponents and cross-component constraints. In this paper we deal with two types of cross-component constraints: precedence constraints and communication constraints. The precedence constraint $prec(A.a, B.b)$ states simply that the codel a of component A must finish its execution before codel b of component B can start its execution (Figure 6). Communication is done through ports which are declared by each component as entrance and exit points. The communication constraints are explicitly specified in the composition expression, therefore the

$$\frac{\sharp(Procs(e_1) \cup Procs(e_2)) = 1 \quad ParMono(e_1, e_2, cst)}{Par(e_1, e_2, cst)} \tag{17}$$

$$\frac{\sharp(Procs(e_1) \cup Procs(e_2)) > 1 \quad ParMulti(e_1, e_2, cst)}{Par(e_1, e_2, cst)} \tag{18}$$

$$\frac{\begin{array}{l} AllDisjoint(iCodels(e_1) \cup iCodels(e_2)) \\ \quad base(e_1) \Rightarrow ValidSched(e_1) \qquad base(e_2) \Rightarrow ValidSched(e_2) \\ \quad\quad (\forall c \in cst)\ IsValid(c, iCodels(e_1) \cup iCodels(e_2)) \\ \quad\quad\quad \neg base(e_1) \Rightarrow compose(e_1) \quad \neg base(e_2) \Rightarrow compose(e_2) \end{array}}{ParMono(e_1, e_2, cst)} \tag{19}$$

$$\frac{\begin{array}{l} base(e_1) \Rightarrow isSetProc(e_1) \wedge ValidSched(e_1) \\ base(e_2) \Rightarrow isSetProc(e_2) \wedge ValidSched(e_2) \\ \quad (\forall c \in cst)\ IsValid(c, iCodels(e_1) \cup iCodels(e_2)) \\ \quad\quad (\forall p\ in Procs(e_1) \cup Procs(e_2)) \\ \quad\quad\quad AllDisjoints(\{i\kappa \in iCodels(e_1) \cup iCodels(e_2) \mid Proc(i\kappa) = p\}) \\ \quad\quad\quad \neg base(e_1) \Rightarrow compose(e_1) \quad \neg base(e_2) \Rightarrow compose(e_2) \end{array}}{ParMulti(e_1, e_2, cst)} \tag{20}$$

$$\frac{\begin{array}{l} (\forall i \in [1,n])\ N_mc(e_i) = 1 \qquad (\forall i \in [1,n])\ ValidSched(e_i) \\ \quad (\forall c \in cst)\ IsValid(ic, \bigcup_{i \in [1,n]} iCodels(e_i)) \\ \quad\quad (\forall i \in [1,n])\ \neg base(e_i) \Rightarrow compose(e_i) \end{array}}{Mode([e_1, ..., e_n], cst)} \tag{21}$$

$$\frac{\begin{array}{l} (\forall i\kappa(n, o, s, f, proc), i\kappa(n', o', s', f', proc) \in icdls) \\ \quad n \neq n' \ \Rightarrow \ ((s < s' \ \wedge \ f \leq s') \ \vee \ (s > s' \ \wedge \ s \geq f')) \end{array}}{AllDisjoints(icdls)} \tag{22}$$

Fig. 5. Composition operators

in and out ports to be connected do not need to have the same name. They do however need to have the same datatype and the same kind of communication protocol (Figure 6).

In the case where the number of microcycles of A and B are the same, the enforcement of these constraints is trivial and is translated simply into a constraint on the order of execution of the codels. If the number of microcycles of A and B is different, then special constructs called converters are deployed. In the simplest representation, converters are codels with a negligible length that are placed on each of the communicating components, between the emission port and the codels. They have the following behaviour:

- if the frequency of emission is higher than the frequency of consumption, then the emission side converter will discard the extra data;
- if the frequency of consumption is higher that the frequency of emission, then the reception side converter will reiterate the last message received the necessary number of times.

$$\frac{(\forall i\kappa(n,\cdot,\cdot,\cdot,proc') \in icdls)\ proc = proc'}{IsValid(assign_proc(n,\cdot,proc), icdls)} \tag{23}$$

$$\frac{\begin{array}{l}(\forall i\kappa_1(n_1,o,s_1,f_1,\cdot), i\kappa_2(n_2,o,s_2,f_2,\cdot) \in icdls)\\ \qquad (s_1 < s_2\ \wedge\ f_1 < s_2\ \vee\ s_2 < s_1\ \wedge\ f_2 < s_1)\end{array}}{IsValid(mutually_exclusive(n_1,n_2,\cdot), icdls)} \tag{24}$$

$$\frac{\begin{array}{l}(\forall ic(n_c,\Pi,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot), ic(n_c',\Pi',\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) \in ics)\\ \qquad (\exists!\,\pi(n_\pi,\tau,\cdot,\delta) \in \Pi)(\exists!\,\pi(n_\pi',\tau',\cdot,\delta') \in \Pi')\ \tau = \tau'\ \wedge\ \delta = \delta'\end{array}}{IsValid(communication(n_c,n_\pi,n_c',n_\pi'), ics)} \tag{25}$$

Fig. 6. Additional rules for validity of constraints

Since the length of these codels is negligible and they do not need to be taken into account in schedule calculation. This mechanism is the default one for converters, but it can be replaced by more sophisticated user-defined mechanisms. The user must then provide in the communication constraint the converter to be used for that communication and the implementation of this converter. In the case where the user-defined converter has a non-negligible length, it will be scheduled like any other codel.

Given the imposed macrocycle length and the obtained composition, a new set of constraints is calculated by unfolding the necessary number of cycles of A and B. Both the individual and the cross-constraints must also be unfolded. A feasible schedule for this constraint set is then calculated.

4.3 Concurrent Composition: The Homogeneous Multiprocessor Case

To extend concurrent composition to include the multiprocessor case, homogeneous processors defined solely by their names are introduced into the component model. These are introduced in three ways in the component definition process:

- for individual components to be made deployable, a list processors will appear in (currently) simplified resource interfaces, where every codel can be executed indifferently on any processor;
- for any given codel, the use of processors can be forced by adding ressource constraints:
  - $cst(assign_processor, n_\kappa, \cdot, p_{id})$, where n_κ is a unique codel identifier and p_{id} is a unique processor identifier, restricts the scheduling of the codel n_κ to the processor p_{id},
  - $cst(mutual_exclusion, n_\kappa, n_\kappa', \cdot)$, where n_κ and n_κ' are unique codel identifiers, sets a mutual exclusion constraint between codels n_κ and n_κ';
- for composition expressions, processor use can also be restricted through cross-constraints *assign_processor* and *mutual_exclusion* with a syntax similar to that of the above constraints, but with codel names that must be prefixed by the names of their components.

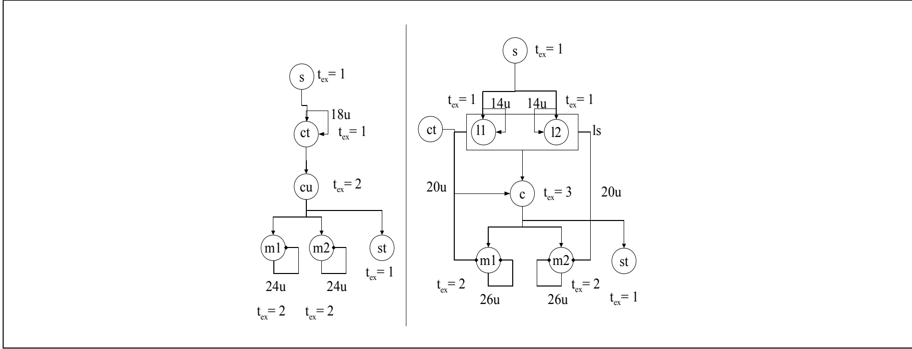


Fig. 7. The representation of the line follower and obstacle avoider componenets modified for mode composition

The communication between codels running on parallel processors depends on the type of medium. For processors communicating through a shared memory, the communication time is instantaneous, whereas for processors communicating by a bus or a network, the connection time can be taken into consideration by annotating the resource interface with the communication times for every couple of processors. Once the processors are set for all the codels, the communication times are represented by automatically generating a *min_delay* constraints between the emitting and the receiving codels.

The algorithm for common period length computation is adapted to estimate the periods with respect to the number of processors available and the enforcement of the processor constraints is verified by the rules presented in Figure 6.

4.4 Operating Modes Composition

Mode composition combines two or more components out of which only one runs in at any given cycle. Modes can represent different patterns:

- competitive behaviors such as the example presented in this paper, where one behavior inhibits another one, and the mode change is triggered by the inhibition signal
- stages of a plan, such as moving until a target is located, collecting samples and getting back. In this case the mode change corresponds to a certain stage of the plan being attained
- addressing rare events (sensor failure, panic mode, ...). A mode change can only happen at the end of a cycle though, therefore if a really quick reaction to an error in necessary, it must be scheduled into the same mode
- a transitory mode, when a mode change requires adaptation (ex : sensor or actuator reconfiguration) a transitory mode containing the instructions necessary to sustain the minimum necessary robotic activity and execute the necessary changes can be defined. This mode is run for a limited number of cycles, triggering the target mode as soon as the changes are in place.

– roles in a plan, in the context of distributed robotics, robots are often devised to be able to take on different roles in their mission, and according to the configuration they must be able to switch between these roles, taking care to respect all the temporal and ressource constraints of the system.

A mode composition requires the modules to share a signal that contains the state of the composite. At the beginning of each cycle a codel common to all modes is run to determine which mode needs to be activated for that cycle. During the execution of the cycle, a new signal must be emitted to signify that a mode change is necessary. A mode change can be triggered by an external event (a message from another node, data from a sensor) or by an internal change (as a result of evaluations, an automatic transition).

To be compatible for mode composition, components must:

– have a common cycle length
– satisfy a ressource interface, by providing the minimum and maximum frequencies required on the ports specified in the interface in every mode. If a minimal/maximal frequency is specified on a port of the interface, then for every mode a schedule emitting the signal with at least/ at most the required frequency must be found. If no frequency is specified, this means that the post is not required to be present in all the modes.

The resulting component will use the union of the time slots occupied by the different modes satisfying the ressource interface provided.

Example. *The example presented in the previous sections can be also used to illustrate composition by modes. The default behaviour (mode) is to follow the line. As soon as an obstacle is detected, the component switches into obstacle avoider mode until the obstacle is succesfully avoided, when it switches back to "line follower" mode. Figure 7 presents the changes in the models of the components. The codel s, whose role is to switch between modes based on the value of a shared signal, is added at the beginning of each mode. Another codel, st is added to the end of each cycle to update the value of the signal used to switch modes. The result of the composition in the formal model is given in Figure 8.*

In the presented example, the time required to switch between the two modes is negligible, and can therefore be considered as included in the execution time of c. In other cases to make all the necessary changes a transition through an intermediary mode might be required.

4.5 A Prototype Java/ECLIPSe Implementation

A prototype implementation of the component model and the accompanying composition model has been done using Java as backbone language for component representation. Each component is represented by a class, and the different constraints and rich interfaces forming the membrane of the component are represented as annotations. A Javassist program has been written to load such components and generate code required by the calculation kernel to generate

$$
\begin{array}{ll}
\mathcal{C}_{mode}(& \mathcal{C}(ObstacleAvoider, \\
\quad \{\mathcal{C}(LineFollower, & \quad \{\kappa(s,\{\sigma_8\},\emptyset,1,\emptyset), \\
\quad\quad \{\kappa(s,\{\sigma_7\},\emptyset,1,\emptyset),\ \kappa(l1,\emptyset,\{\sigma_1\},1,\{\rho_1\}), & \quad\ \kappa(ct,\emptyset,\{\sigma_9\},1,\{\rho_3\}), \\
\quad\quad\ \kappa(l2,\emptyset,\{\sigma_2\},1,\{\rho_2\}),\ \kappa(ct,\emptyset,\{\sigma_3\},1,\{\rho_3\}), & \quad\ \kappa(cu,\{\sigma_9\},\{\sigma_{10},\sigma_{11},\sigma_{12}\},2), \\
\quad\quad\ \kappa(c,\{\sigma_1,\sigma_2,\sigma_3\},\{\sigma_4,\sigma_5,\sigma_6\},3,\emptyset), & \quad\ \kappa(m1,\{\sigma_{10}\},\emptyset,1,\{\rho_4\}), \\
\quad\quad\ \kappa(m1,\{\sigma_4\},\emptyset,2,\{\rho_4\}),\ \kappa(m2,\{\sigma_5\},\emptyset,2,\{\rho_5\}), & \quad\ \kappa(m2,\{\sigma_{11}\},\emptyset,1,\{\rho_5\}), \\
\quad\quad\ \kappa(st,\{\sigma_6\},\{\sigma_7\},1,\emptyset)\}, & \quad\ \kappa(st,\{\sigma_{12}\},\{\sigma_8\},1,\emptyset)\}, \\
\quad\quad \emptyset, & \quad \{cst(temporal,s,ct,\cdot), \\
\quad\quad \{cst(temporal,s,l1,\cdot), & \quad\ ... \\
\quad\quad\ ... & \quad\ cst(max_freq,ct,18), \\
\quad\quad\ cst(max_delay(l1,m1,20), & \quad\ cst(min_freq,m1,24), \\
\quad\quad\ ... & \quad\ cst(min_freq,m2,24)\}, \\
\quad\quad\ cst(max_freq,l1,14), & \quad R(\{\rho(p1,proc)\},\cdot),24,1, \\
\quad\quad\ cst(max_freq,...),...\}, & \quad \{i\kappa(s,1,0,1,\{\rho(p1,proc)\}), \\
\quad\quad R(\{\rho(p1,proc)\},\cdot),24,1, & \quad\ ...\})\}). \\
\quad\quad \{i\kappa(s,1,0,1,\{\rho(p1,proc)\}), & \\
\quad\quad\ ...\ \}), &
\end{array}
$$

Fig. 8. The mode composition example in the formal model

the schedule and composition instructions that are fed back into Java, allowing to construct the deployable assembly.

All the calculations are done by a kernel based on the formal component model, independent of the chosen implementation and modelling languages. The high-level declarative component representation is thus transformed into a formal model description of the constraint used to calculate a correct by construction formal icomponent model. The current implementation of the kernel is written in ECL^{i}PS^e (an extension of Prolog for constraint programming, chosen for the ease it allows in expressing the component constraints) using resolution, backtracking and labelling techniques to build constructive proofs of composite validity.

5 Related Work

Even though several component models for robotics have been proposed (see [2,6] for a recent wide coverage of the area), few models take a component as sensori-motor behavior point of view. CoSARC [7] or the recent work on G^{en}oM with BIP controllers developed at Verimag [8] are notable exceptions. Yet, these do not explicitly address composition problems. CoSARC uses OO Petri-nets as basic schedulers and simply connects components through tokens exchanged among the differents schedulers. G^{en}oM/BIP defines components under the synchronous hypothesis where compositions always produce centralized (monoprocessor) schedulers. Schlegel looks at the composition problem in the context of robotics, but not beyond the mere connection of components, by postulating that they can be made of only a limited number of patterns providing for low coupling, management of heterogeneity and dynamic wiring [9].

Related works in the field of component-based real-time systems do not cover all the issues addressed in this paper. Dominguez and Day [10] propose compositional rules to reason about liveness properties of communication protocols port-based components. Maraninchi and Bouhadiba [11] study the composition of components under the synchronous hypothesis where no time constraints are given. Gu and He [12] concentrate on scheduling under time constraints but do not address resource usage constraints, nor do they consider the composition issues *per se*. Lipari et al. [13] compose components having time constraints but imposing their own scheduler; hence, they concentrate on verifying assemblies rather than producing valid ones. Sandström et al. [14] also address time-constrained scheduling but not in the context of different composition operators. We share with Wuyts et al. [15] the use of CLP in producing schedules under time constraints, but complement this work by better tackling resource usage constraints and by fitting the approach in a composition-oriented setting.

6 Conclusion

Composition of autonomous robotic software components is still a grand challenge. Beyond the traditional compatibility of method signatures, several other aspects must be taken into account, such as the communication, precedence, timing and resource usage constraints. We have developed a kernel of composition operators over a component model adapted to autonomous robotics. This model aims to produce correct-by-construction assemblies from components which declares in rich interfaces all the information needed to perform the composition. Our contributions are threefold:

– an extensible set of composition operators aiming to cover the domain-specific needs for assembling components as full-gledged sensori-motor robot behaviors;
– the organization of operators into hierarchies from generic to deployment context-dependent, which enables an automatic process taking programmer-written composition expressions to turn them into fully deployable assemblies given context information declared by containers; and
– a formal semantics for these composition operators in the form of composition rules and constraints, which constructive proofs produce the necessary information to generate the deployable assemblies.

Short term future work includes developing more composition operators and studying the compositionality of their semantics to address large-scale assemblies. On a medium term, properties of operators as an algebra, which commutativity, associativity and distributivity properties need a deeper study. On a longer term, a programming methodology and associated tools (e.g. better and more robust implementation algorithms) will be needed to manage the lifecycle of these components and assemblies, and more precisely taming the progressive introduction of the precise deployment context information in order to tune the generation of assemblies to this context.

References

1. Rogovchenko, O., Malenfant, J.: Composants et composition pour les architectures de contrôle de robots. Revue des systèmes série JESA. Journal Européen des Systèmes Automatisés 42, 423–438 (2008)
2. Rogovchenko, O., Malenfant, J.: Components for Control Architectures of Autonomous Robots (2010) (submitted for publication)
3. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Engineering theories of software intensive systems. NATO Science Series, vol. 195, pp. 83–104. Springer, Heidelberg (2005)
4. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed Interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)
5. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource Interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
6. Brugali, D. (ed.): Software Engineering for Experimental Robotics. Springer Tracts in Advanced Robotics, vol. 30. Springer, Heidelberg (2007)
7. Passama, R., Andreu, D., Dony, C., Libourel, T.: CoSARC, un langage de composants pour l'ingnierie des architectures robotiques. Revue des systèmes série JESA, Journal Européen des Systèmes Automatisés 42 (2008) 439–458
8. Gallien, M., Gargouri, F., Kahloul, I., Krichen, M., Nguyen, T.H., Bensalem, S., Ingrand, F.: D'une approche modulaire à une approche orientée composant pour le développement des systèmes autonomes: Défis et principes. In: Proceedings of Control Architectures of Robots, CAR 2008 (2008)
9. Schlegel, C.: Communication patterns as key towards component interoperability. [6]
10. Dominguez, A.L.J., Day, N.A.: Compositional Reasoning for Port-Based Distributed Systems. In: Proceedings of ASE 2005, pp. 376–379. ACM, New York (2005)
11. Maraninchi, F., Bouhadiba, T.: 42: Programmable Models of Computation for a Component-Based Approach to Heterogeneous Embedded Systems. In: Proceedings of GPCE 2007, pp. 53–62. ACM, New York (2007)
12. Gu, Z., He, Z.: Real-Time Scheduling Techniques for Implementation Synthesis from Component Software Models. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 235–250. Springer, Heidelberg (2005)
13. Lipari, G., Gai, P., Trimarchi, M., Guidi, G., Ancilotti, P.: A Hierarchical Framework for Component-Based Real-Time Systems. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 209–216. Springer, Heidelberg (2004)
14. Sandström, K., Fredriksson, J., Akerholm, M.: Introducing a Component Technology for Safety Critical Embedded Real-Time Systems. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 194–208. Springer, Heidelberg (2004)
15. Wuyts, R., Ducasse, S., Nierstrasz, O.: A data-centric approach to composing embedded, real-time software components. The Journal of Systems and Software 74, 25–34 (2005)

Event-Specific Software Composition in Context-Oriented Programming

Malte Appeltauer[1], Robert Hirschfeld[1], Hidehiko Masuhara[2], Michael Haupt[1], and Kazunori Kawauchi[2]

[1] Hasso-Plattner-Institute, University of Potsdam, Germany
first.last@hpi.uni-potsdam.de
[2] Graduate School of Arts and Science, University of Tokyo, Japan
{masuhara,kazu}@graco.c.u-tokyo.ac.jp

Abstract. Context-oriented programming (COP) introduces dedicated abstractions for the modularization and dynamic composition of crosscutting context-specific functionality. While existing COP languages offer constructs for control-flow specific composition, they do not yet consider the explicit representation of event-specific context-dependent behavior, for which we observe two distinguishing properties: First, context can affect several control flows. Second, events can establish new contexts asynchronously. In this paper, we propose new language constructs for event-specific composition and explicit context representation and introduce their implementation in JCop, our COP extension to Java.

1 Introduction

With the increasing demand for personalization and mobility of applications, context awareness gains growing relevance as a distinguishing feature of software systems. To meet the challenges of developing and managing context-specific behavior, several approaches have emerged, each providing its own definition of context. We adopt a notion where context is constituted by a set of predicates and a set of variation modules. The former are evaluated to determine the context's presence, and the latter are composed based on the result of predicate evaluation. Variation implementations are often scattered over application source code and can so be characterized as crosscutting concerns. With that, a major task of context representation is the modularization of such crosscutting concerns. In addition to modularization, context-specific crosscutting concerns require means for dynamic composition.

Context-oriented programming [22] (COP) is an approach to representing context-specific concerns, focusing on dynamic composition of control flows. COP allows for the definition of *layers*, modules that crosscut object-oriented decomposition and encapsulate the implementation of behavioral variations. For instance, a security layer can extend various methods with access control features without affecting the original method declarations. Depending on the execution context, layers are *composed* into a system at run-time. A layer composition defines the order in which layers adapt the base system. This way, COP separates

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 50–65, 2010.
© Springer-Verlag Berlin Heidelberg 2010

the *definition* of adaptations from their *composition*, distinguishing it from alternative multi-dimensional modularization techniques such as aspect-oriented programming [26] (AOP), Mixins [12], or Classboxes [11]. The aforementioned security layer could be applied for specific control flows, while at the same time other computations can be executed with the basic functionality.

In the following, we distinguish between the separation of *adaptation code* and *composition code* from the base code. While the former is adequately handled by layers, the latter deserves better language support. The COP languages implemented so far [2] support selective activation and deactivation of layer compositions, expressing programmatically when the application enters and leaves certain contexts. It is, however, not enough to regard context as being entirely under programmer control; instead, context can impose itself on the running application "from the outside". Based on this observation, we distinguish control-flow specific from event-specific contexts. Two key properties characterize them:

1. Event-based context can overlap several control flows, unlike control-flow specific context, which is confined to a single control flow. For instance, context change in graphical user interface (GUI) applications can affect the behavior of several event handler methods at once.
2. Event-based context entry and exit often cannot be localized at fixed points in the control flow. Instead, context entry depends on asynchronous events independent from main control flow. Moreover, a certain context is often active until another event changes the composition. Any kind of sensor data, such as localization or temperature, are examples of independent context information that may asynchronously trigger system recomposition.

The former property implies that event-based context (de)activation leads to layer composition statements' being scattered over several locations, each of which corresponds to one of the affected control flows. The latter property implies that it is impossible to determine the locations where to place layer composition (de)activation. Also, asynchronous composition can lead to inconsistent system state within a control flow. With the abstractions of state-of-the-art COP languages, event-based context (de)activation cannot be represented without scattering layer composition statements over the program. Instead, first-class support for contexts is required, enabling declarative description of events that constitute context entry and exit. In addition, a possible solution must take composition consistency of asynchronous context change into account.

Contribution. In this paper, we motivate the need for explicit representation of event-specific context-dependent composition along a case study that we conducted using ContextJ [5,3], our earlier COP extension to the Java programming language. We present appropriate abstractions adopted from AOP to cope with event-based behavioral variations. We introduce the *JCop* programming language extension that supports these constructs while preserving composition consistency as defined by COP. As a proof of concept, we apply JCop to our case study and discuss its expressiveness.

Outline. The rest of the paper is structured as follows. Section 2 introduces COP and describes our case study in which we developed a context-aware event-based GUI application using ContextJ. We discuss our experience concerning the case study in Section 3. Section 4 introduces JCop, Section 5 discusses the expressiveness of JCop with respect to the GUI implementation. Section 6 presents related work, while Section 7 summarizes the paper.

2 Event-Specific Behavioral Variations

Any computation in a program flow is executed within a specific context, such as system state or user-specific configuration, that can influence system behavior. The COP approach provides a first-class representation of context-specific behavioral variations that can be dynamically composed for a specific control flow. COP focuses on control-flow specific composition of behavioral variations and omits providing dedicated abstractions for event-specific composition, which we will address in the next sections.

2.1 Context-Oriented Programming

COP extends object-oriented programming with first-class abstractions for behavioral variations that can be composed into a system depending on execution context. COP assumes *context* to be *everything that is computationally accessible*, such as object state, network bandwidth, or user interaction.

COP provides *layers* [9] as a modularization concept that can crosscut an object-oriented decomposition and encapsulate context-specific behavioral variations, represented as *partial method definitions*. COP extends object-oriented method dispatch with dynamic composition of crosscutting concerns. To distinguish between the different kinds of method definitions, we introduce the terms *plain method definition* and *layered method definition*. A plain method is one whose execution is not affected by layers. Layered methods consist of a *base method definition*, which is executed when no active layer provides a corresponding partial method, and at least one partial method definition.

Layers can be activated and *composed* with others at run-time. When activated, layered method calls are dispatched to the partial method provided by the layer. Partial methods can be executed before, after, or around the base method definition. In a composition, multiple layers may provide partial definitions of the same method. In that case, a partial method can *proceed* to the next partial definition in the composition or, if none exists, to the base method definition. This feature has been previously introduced by other languages such as Common Lisp [24] and AspectJ [25]. Layer composition is controlled *per thread* and is by default scoped to the dynamic extent of a block of statements.

ContextJ[1] [2,3] is a COP implementation for Java. It supports layer declaration within classes and explicit layer composition. In the following, we will explain ContextJ's main features along an example.

[1] ContextJ is available for download at `http://www.hpi.uni-potsdam.de/swa/cop`

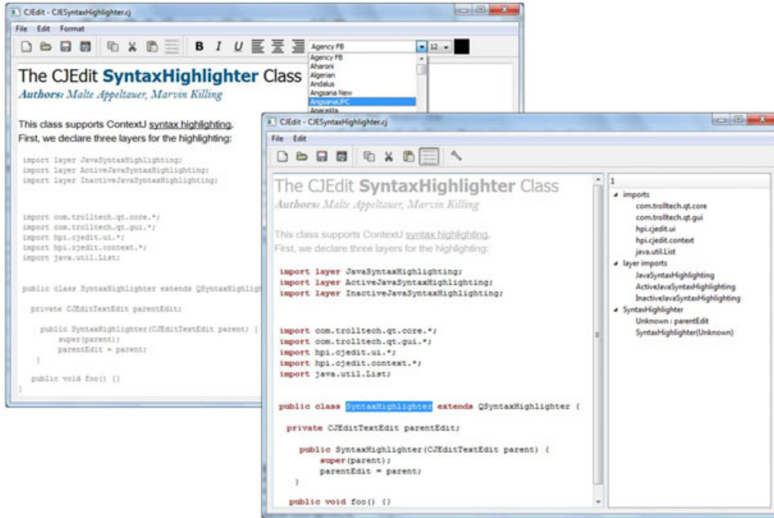


Fig. 1. Screenshots of CJEdit. *Left*: rich-text editing with format toolbars and menus. *Right*: program development is supported by an outline and focus on source code blocks.

2.2 Case Study: CJEdit

As a case study [4], we have developed a little IDE using ContextJ, whose GUI provides context-specific behavior.

Figure 1 shows two screenshots of our *CJEdit* application, a simple programming environment that provides different UI elements and behavior for the user-driven activities *programming* and *commenting*. The left-hand screenshot presents the application's commenting mode in which the toolbar offers various text formatting actions. The right-hand image shows the programming mode, where the editor comes with an outline and a different toolbar. To support focusing on source code, any rich text within the document is displayed in gray. The editor supports syntax highlighting, an outline view, a compilation/execution toolbar, and *rich text commenting features*, such as font and color modifications. Based on the user's actual task (i. e., context), the UI only offers relevant tools, menus, and widgets. The UI is recomposed upon context switches, which are either directly triggered by the user, or by text cursor changes. To enter programming context, the user can push a toolbar button. Moreover, context is changed whenever the text cursor moves from text to code and vice versa. CJEdit's core is implemented using ContextJ and the *Qt Jambi GUI Framework*[2]. The editor consists of approximately 3,000 lines of code in 38 classes.

Figure 2 shows the implementation of the *programming* activity-specific widgets using layers. In ContextJ, layers, denoted by the keyword `layer`, can be

[2] Nokia Corporation, Whitepaper: A Technical Introduction to Qt, 2008 `www.qtsoftware.com`

```
1  import layer CodeWidgets;
2  import layer Outline;
3  import layer RTFWidgets;
4
5  public class CJEditWindow extends QMainWindow {
6    ...
7    private void drawToolBars() {...}
8    private void drawMenus() {...}
9    private void drawWidgets() {... drawMenus(); drawToolbars(); }
10
11   layer CodeWidgets {
12     // partial methods
13     after private void drawToolBars() {...}
14     after private void drawMenus() {...}
15     // auxiliary members
16     private CodeToolBar codeToolBar;
17     private Menu codeMenu;
18     private CodeToolBar createToolBar() {...}
19     private Menu createMenu() {...}
20   }
21   layer Outline {
22       ...
23   }
24   layer RTFWidgets {
25       ...
26   }
27 }
```

Fig. 2. Layered specification of task-dependent GUI Widgets

defined in classes[3] and contain partial method definitions that are executed—depending on their modifiers—before, after, or around their base method.

The same layer can be partially defined in multiple classes; for instance, `CodeWidgets` can also provide partial methods for `CJEditTextEdit`, which implements the text editor widget. The layers shown in Figure 2 provide partial methods responsible for drawing UI elements, and auxiliary methods accessible from within the layer only to create these objects.

Each text block object of the underlying document tree holds a list of layers that should be activated when its text is focused by the user. A focus is set by text cursor selection. By default, text blocks refer to the layers responsible for *rich text commenting* behavior. If the user switches to the *programming* activity (by pressing the code button in the toolbar), subsequently created text blocks are linked with programming environment-specific layers.

The application is recomposed and its GUI redrawn whenever the current block type switches. The dynamic composition of our previously specified layers is depicted in Figure 3. For layer composition, ContextJ provides a `with` statement that specifies the layers to be activated, and the dynamic extent for which the composition is valid. To explicitly disable a layer for a control flow, the `without` statement can be used. Recomposition can be triggered by the `onCursorPositionChanged` event handler that checks if the block type of the

[3] Note that this is a design decision in ContextJ and not a general restriction of COP. The JCop language introduced in Section 4 allows for the specification of layers as top-level elements.

```
1  public class CJEditWindow extends QMainWindow {
2    private List<Layer> getLayersOfCurrentBlock() {
3      if (currentBlock.getType() == BlockType.TEXT)
4        // returns RTFWidget
5      if (currentBlock.getType() == BlockType.CODE)
6        // returns CodeWidget and Outline
7    }
8    private boolean blockTypeChanged() {
9      // true if the focused block has a different type than its predecessor
10     ...
11   }
12   void onCursorPositionChanged() {
13     if (blockTypeChanged()) {
14       with (getLayersOfCurrentBlock()) { drawWidgets(); }
15     }
16   }
17   void onPrint() {
18     with (getLayersOfCurrentBlock()) { ... }
19   }
20   void onSave() {
21     with (getLayersOfCurrentBlock()) { ... }
22   }
23   void onFileNew() {
24     with (getLayersOfCurrentBlock()) { ... }
25   }
26 }
```

Fig. 3. Dynamic composition in CJEdit

previously focused block is different to that of the current block. If so, the method calls `drawWidgets` to update the UI using the current block's layer composition.

3 Lessons Learned

Although CJEdit is a relatively small application with only a few context-dependent concerns, its ContextJ-based implementation eases the development process compared to a plain Java solution. From a structural point of view, layers allow for a better separation of concerns. Base methods only have to care for the editor's default behavior, while layers completely encapsulate their context-specific variations. In our scenario, context-specific behavior is strongly coupled with private state of extended classes, thus layer declaration within classes is the appropriate strategy for layer implementation. Dynamic GUI adaption is also expressed naturally by layer compositions.

These benefits aside, some characteristics of GUI-based programming had to be considered that led to additional challenges for the ContextJ-based implementation. In the following, we discuss the two most important findings.

3.1 Problems

Scattered Composition Statements. User interaction with a GUI is event-driven rather than control flow-centric. This complicates dynamic extent-based layer composition as originally proposed by COP. Figure 4 depicts an execution sequence in CJEdit, where user interaction triggers several event handlers, such

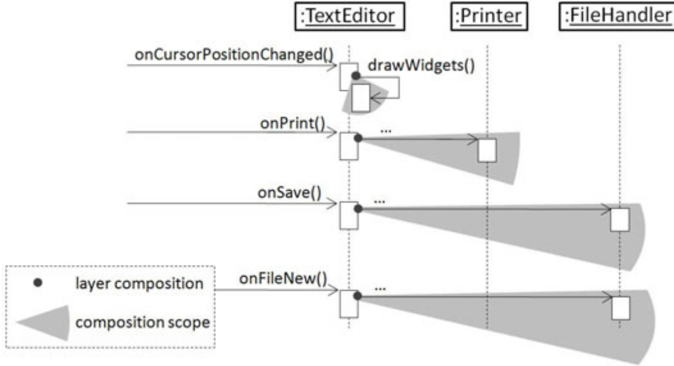


Fig. 4. Scattered layer compositions for event handlers in CJEdit

as printing the document, writing new text, or moving the text cursor through the document. Each event handler activates the layers of the currently focused text block for their respective control flows. In the source code, this issue is manifested in the form of identical `with` statements occurring in several event callback methods, as shown in Figure 3. More formally, we identify two different kinds of cross-cutting concerns, according to [1]: The actual behavioral variations implemented using layers are *heterogeneous* concerns, therefore they should be defined close to their respective objects. ContextJ serves this purpose well. Conversely, layer composition statements in CJEdit constitute a *homogeneous* cross-cutting concern that is not modularized by COP abstractions.

Event-specific Context Representation. COP generally defines context as *everything that is computationally accessible*, meaning that any event a system can recognize can influence the current layer composition. With the intention to avoid further restrictions, COP does not provide means for explicitly describing when context influences an execution. In general, it is impossible to globally describe the circumstances under which a composition should be used, since, for different executions, these properties can entirely change. The explicit `with` statement in COP exists due to this fact. However, the nature of what we denote as *event-specific context* is more predictable. In CJEdit, the *programming* context is constituted by the fact that the focused text contains source code; for the commenting context, the text must contain rich text elements. However, it is impossible to simply express these properties using the original COP abstractions provided by ContextJ. Instead, the application must provide information about the composition to be activated and must check for the context change itself.

3.2 Our Solution

To address the aforementioned problems, we propose an alternative to explicit composition statements. A declarative specification supporting the description

of control flow entry points helps avoid scattered composition statements. For event-based composition, we suggest a declarative definition of the event condition and its respective layer composition.

4 Event-Specific COP with JCop

From our CJEdit experiments, we conclude that scattered `with` statements and event-based layer composition deserve appropriate lingual abstractions. In this section, we present the JCop language that provides new constructs for declarative and event-based layer composition and a first-class event-based context representation. We discuss issues regarding module consistency within a dynamic extent and explain how JCop ensures consistent event-specific adaptations.

4.1 JCop Overview

The JCop language combines COP features provided by its predecessor ContextJ with alternative layer declaration and composition features.

Layers can either be defined within the classes for which they provide behavioral variations (*layer-in-class*), or in a dedicated top-level layer similar to an aspect (*class-in-layer*)[4] [22,2]. Besides the structural differences of the two declaration styles, layer-in-class can access and extend the host object's internal state and methods, we restrict class-in-layer to public interfaces in order to sustain encapsulation. Developers can decide per situation if they prefer to define a layer within its enclosing class, allowing private member access, or to declare all partial definitions of a layer as one layer module to reduce scattering. For layer composition, JCop provides the control-flow specific `with` and `without` statements known from ContextJ.

The JCop compiler is implemented based on the *JastAdd* [20] meta-compiler framework and extends the Java 1.5 specification *JastAddJ* [16]. In addition, we adopted the AspectJ method pattern grammar used by the `abc` compiler [7]. It compiles JCop source code to Java 1.5 byte code.

4.2 Declarative Layer Composition

As stated above, event-based systems can handle multiple events whose behavior depends on identical layer composition. A layer composition spanning several control flows requires an explicit composition statement at the beginning of each of them. Thus, layer composition can be a homogeneous crosscutting concern applying the same functionality at several points in the system.

JCop introduces a *declarative layer composition* statement. It consists of a logic concatenation of *predicates* and a *composition block*. A composition

[4] If both styles are used to define the same layer, the compiler avoids ambiguities by asserting that a partial method must not be defined in both a class-in-layer and a layer-in-class declaration simultaneously.

```
1 in(CJEditWindow win) &&
2 (
3   on(* CJEditWindow.onPrint(..)) ||
4   on(* CJEditWindow.onSave(..)) ||
5   on(* CJEditWindow.onFileNew(..)) ||
6   on(* CJEditWindow.drawWidgets(..))
7 )
8 {
9   with(win.getLayersOfCurrentBlock());
10 }
```

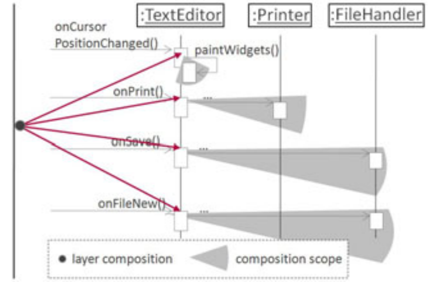


Fig. 5. Using the `on` predicate in CJEdit

block contains a `with` and/or `without` statement specifying the layers to be (de)activated[5]. Like for the general `with` statement, any expression returning a layer or a list of layers is a valid parameter, so layer compositions can also be computed. Declarative `with` statements are re-evaluated for every execution of the methods they are bound to.

Quantify Over Control Flows. To address the issue of scattered `with` statements, we adopted features of AOP [26, 18], where scattered functionality is expressed by *advice* blocks bound to *pointcuts* quantifying over a set of *join points*, well-defined events in the execution graph. JCop introduces a pointcut designator denoted `on`. It contains an AspectJ-like method pattern [25] specifying those methods to whose dynamic extent a layer composition is to be applied. We restrict the pattern to describe only those methods visible to the declaration without breaking encapsulation rules. The optional `in` predicate allows for binding the object on which the composition declaration should be evaluated.

Figure 5 presents a declarative layer composition that specifies the scope of layer activation for CJEdit. For all event handler callbacks and `createWidgets`, the composition of the currently focused block is used.

Conditional Composition. For a clearly specifiable set of method executions participating in a layer composition, the `on` predicate is our preferred means. For more complex structures, however, the explicit specification of control-flows becomes increasingly verbose.

In addition to `on`, JCop allows for a more implicit description of composition time independent of the actual execution in the main control flow. We support the fact that a context activation event is reflected in the change of some property that *is computationally accessible* and provide a `when` predicate that allows for the specification of a Boolean expression evaluating this property. The predicate is evaluated before the execution of any layered method that is potentially

[5] If a layer is referred by both lists, it is activated and deactivated at the same time and thus will be ignored for composition.

```
1 import layer RTFWidgets;
2 import layer CodeWidgets;
3 import layer Outline;
4
5 context Commenting {
6   in(CJEditWindow win) &&
7   when(win.getCurrentBlockType() ==
8        BlockType.Commenting)
9   {
10      with(RTFWidgets);
11      without(CodeWidgets, Outline);
12  }
13 }
14 context Programming {
15   in(CJEditWindow win) &&
16   when(win.getCurrentBlockType() ==
17        BlockType.Programming)
18   {
19      with(CodeWidgets, Outline);
20      without(RTFWidgets);
21   }
22 }
```

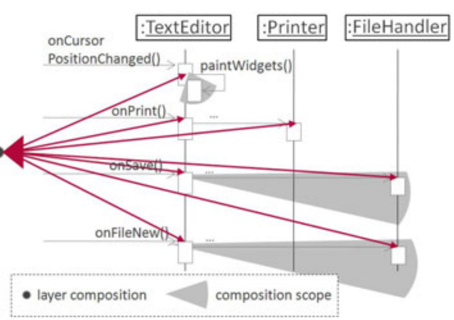


Fig. 6. Using the `when` predicate in CJEdit

affected by the respective layer composition. If the `when` predicate is evaluated to `true`, these layered methods are executed using the composition.

Figure 6 shows two declarations of event-based composition for CJEdit (Lines 6–12, 15–21). In both statements, the `when` predicates specify the current block type required to activate the composition. The predicate expressions are evaluated on a `CJEditWindow` instance that is bound by an `in` designator.

The `when` predicate completely relieves the developer from having to specify *where* a layer composition should begin and only requires the declaration of *when* composition takes place. Nevertheless, the combination of `on` and `when` is useful to restrict the scope in which `when` should be evaluated.

4.3 First-Class Context Representation

Since declarative and event-based composition statements are independent of specific objects, they should be defined in a dedicated location. For this reason, JCop provides a first-class `context` construct. Like layers, contexts are special singleton types that cannot be instantiated. The construct can host both declarative and event-based composition statements and auxiliary methods and fields.

Figure 6 presents context declarations for our programming and commenting contexts. The contexts contain an event-based composition statement that declares when the context changes and which layers are composed.

4.4 Composition Consistency in a Dynamic Extent

Programming languages and frameworks that support dynamic recompositions offer extended expressiveness. This, in turn, can lead to inconsistent and unintuitive control flows. A typical example is the recomposition of a method while

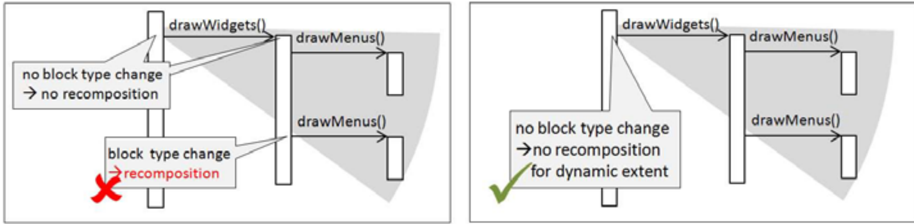


Fig. 7. Layer composition scopes and activation

it is being executed. To aid the developer in avoiding such undesired behavior, the original COP approach restricts layer composition to a dynamic extent. Our event-based composition evaluates the `when` predicate every time a method invocation is potentially dispatched to a layer involved in the composition. Without additional restrictions, this approach cannot guarantee that a dynamic extent is executed with a consistent layer composition.

Figure 7(a) exemplifies this issue for CJEdit. Assume that text focus can be changed asynchronously to the UI `drawWidgets` operation, which, among others, calls `drawMenus`. Both methods are layered and draw the UI according to the current context. If we apply the contexts declared in Figure 6 using the strategy described above, and block focus changes during `drawWidgets`, the redrawn GUI would partially consist of both programming and commenting context parts.

Besides the fact that such behavior is obviously undesired, the implicit and asynchronous composition activation is hard to debug. Tracing this kind of failures is tedious. JCop prevents such inconsistencies by ensuring that, once `when` is evaluated, the predicate will not be re-evaluated within the dynamic extent originating from this evaluation, as depicted in Figure 7(b). This strategy conforms to the original context-oriented programming model: once a composition has been activated, it is consistent and valid until its `with` block terminates. In our extension, this assumption holds: a composition is valid until the control flow returns to the point at which the composition has been created.

5 Discussion

In Section 3, we identified some issues concerning the representation of event-based behavioral variations using programming language abstractions provided by the original COP approach. First, if layer compositions range over several control-flows, their respective composition statements must be repeated at any of their potential entry points. Second, event-specific composition cannot be explicitly declared but must be handled by application logic. In the following, we discuss how our new JCop language constructs solve these issues.

Composition Declarations to Solve Scattering. The phenomenon of scattered functionality that requires code repetition is well known as crosscutting

concern for which AOP provides encapsulation mechanisms. In JCop, we fuse COP with some concepts of AOP to address this problem.

JCop introduces context types that contain a declarative composition statement similar to a pointcut-advice construct. Declarative compositions allow for the layer composition of several control flows. Scattered composition statements can be avoided using the declarative composition in combination with the `on` predicate, which allows for the specification of all method executions to be included in the scope of a layer composition.

When-Declarations for Event-based Composition. In addition to `on`, JCop provides a `when` predicate for composition declarations. It describes the property that must be fulfilled for the activation of a composition and thus reliefs the application logic from managing compositions and their respective events. One of the key properties of COP is the consistency of a layer composition within a dynamic extent. JCop ensures that this property is not violated by event-based composition, as described in Section 4.4.

Context Types Encapsulate Context Specification. JCop's context types release the application logic from handling layer compositions and event-specific context changes. Besides composition declarations, context types can contain auxiliary members to compute layer compositions or store relevant context information.

6 Related Work

Other COP Languages. Most COP extensions have been developed for dynamic languages, such as Lisp [13,14] Smalltalk [21], Python [31,23], and the *delMD-SOC* kernel [29]. They all implement the original semantics of COP, based on meta-programming facilities of their respective host language. A detailed comparison of COP language features is provided in [2]. JCop is the first language that fuses COP with AOP for a more declarative composition scope specification. Except for the Python extension *PyContext* [31] that provides implicit layer activation, none of the mentioned languages support event-based layer activation. The *Ambience* language is another approach to context-orientation. Based on the *Ambient Object System* [19], it supports behavior adaptations with partial method definitions and context objects, which correspond to COP layers. Ambience does not support implicit context activation based on the evaluation of an expression as supported by JCop's `when` predicate.

Aspect-oriented Programming. The main distinction between AOP and COP (including JCop) is that the former allows for a joint specification of *when* in the execution flow *what* kind of functionality should be used, while COP separates *when* (using explicit `with` statements) from *what* (using layers and partial methods). JCop exceeds COP by introducing declarative composition statements.

AspectJ [25] is a popular Java language extension that established the notion of *join points*, well-defined events in the execution of a program that can be described by *pointcut* predicates and can be adapted by *advice* blocks. JCop's `on` predicate is equivalent to AspectJ's `execution` pointcut, except that the former's

method patterns are restricted to public methods to preserve encapsulation. AspectJ's `if` pointcut contains an expression that is evaluated at a join point. It is of use only when concatenated with other pointcuts that provide the set of join points on which `if` is evaluated. JCop's `when` predicate is similar to `if` as it dynamically evaluates a condition. However, `when` uses an implicit set of join points, namely all executions of layered methods.

Most AspectJ-like languages do not support dynamic aspect weaving that could simulate COP layer activation. However, they can mimic COP behavior using pointcuts and advice, though in an unwieldy manner since the pointcut specifications get complex. In some languages, such as *CaesarJ* [6], aspects can be deployed for a dynamic extent at run-time, much like explicit `with` statements. However, CaesarJ does not provide first-class context and behavioral variations but rather supports variability at a different level of abstraction.

Alternative Adaptation Techniques. Modularization approaches such as *traits* [30,15] and *mixins* [12] allow for an additional inheritance relationship next to the class hierarchy, but do not offer dynamic adaptation like layers. Feature-oriented programming (FOP) [10] and its Java-based implementation AHEAD [8] provide layer-like modules to specify adaptations of methods and classes (and other software artifacts). However, FOP and AHEAD apply compile-time composition of feature variations in contrast to run-time composition as provided by COP and JCop. The *Classbox/J* [11] module system extends Java's packaging and scoping mechanism. A classbox is an explicitly named scope in which classes and their members can be defined. Besides common subclassing, Classbox/J supports local refinement of imported classes by adding or modifying their features without affecting the originating classbox, much like layers and partial methods. However, it does not provide means for dynamic composition.

Event-based Programming. An important difference between event-based programming and event-based context (de)activation deserves to be highlighted. Event-based programming supports the synchronous or asynchronous trigger of *action* as events are signaled. Conversely, event-based context (de)activation triggers *recomposition*, which influences the binding of actions at interfaces. Obviously, context (de)activation events have a certain influence on action characteristics, but this is expressed only in terms of bindings of actions to interfaces; actions are not immediate (synchronous or asynchronous) results of events.

The CaesarJ extension *ECaesarJ* [27] supports the definition of context as a class implementing two events representing context entry and exit. Unlike JCop, ECaesarJ does not provide a layer-like representation and composition mechanism of behavioral variations. Moreover, objects must explicitly handle context change, whereas event-based context implicitly changes the composition.

EventJava [17] models events as asynchronous methods and compound events by correlation patterns. Event-specific behavior is encapsulated in method bodies of correlation patterns that allow access to application-specific data and to implicit context information of the event, which can be customized for application-specific purposes. The execution of event methods can be restricted through

predicates specified in a `when` clause. Contrary, JCop's `when` construct specifies the constraints under which an event is triggered.

In *Ptolemy* [28], code blocks are bound to events, similar to pointcut-advice binding in AOP. Classes can contain binding definitions to such events or to compositions of multiple events. Events are explicitly announced, contrary to JCop's implicitly evaluated `when`. Ptolemy's event handling mechanism allows for the immediate execution of functionality on event announcement, while JCop ensures that event-based layer compositions wait until the execution stack has reached a safe point for recomposition.

7 Summary and Conclusion

We discussed the requirements for context-oriented programming languages to support event-specific context-dependent behavioral variations along a case study implemented using a conventional COP language. For a better separation of layer composition from application logic, we adopted pointcuts from AOP and developed declarative composition statements. As an implementation of these concepts, we presented JCop, our new language extension to Java. We applied JCop to our case study to show that our new language abstractions allow for a more declarative and intuitive specification of event-based behavioral variations.

As we have shown in this paper, different types of context require different programming language representations in its support. In future work, we will conduct a thorough analysis of variations of possible layer composition scopes beyond control-flow and event-based scope.

References

1. Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Transactions on Software Engineering 34(2), 162–180 (2008)
2. Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A Comparison of Context-oriented Programming Languages. In: COP 2009: International Workshop on Context-Oriented Programming, pp. 1–6. ACM Press, New York (2009)
3. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ - Context-oriented Programming for Java (2009) (submitted)
4. Appeltauer, M., Hirschfeld, R., Masuhara, H.: Improving the Development of Context-dependent Java Applications with ContextJ. In: COP 2009: International Workshop on Context-Oriented Programming, pp. 1–5. ACM Press, New York (2009)
5. Appeltauer, M., Hirschfeld, R., Rho, T.: Dedicated Programming Support for Context-aware Ubiquitous Applications. In: UBICOMM 2008: Proceedings of the 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, Washington, DC, USA, pp. 38–43. IEEE Computer Society Press, Los Alamitos (2008)
6. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: Overview of CaesarJ. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)

7. Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: An Extensible AspectJ Compiler. In: AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, pp. 87–98. ACM Press, New York (2005)
8. Batory, D.: Feature-Oriented Programming and the AHEAD Tool Suite. In: ICSE 2004: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, pp. 702–703. IEEE Computer Society, Los Alamitos (2004)
9. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engeneering Methodologies 1(4), 355–398 (1992)
10. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering 30(6), 355–371 (2003)
11. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: controlling the scope of change in Java. In: OOPSLA 2005: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, And Applications, pp. 177–189. ACM Press, New York (2005)
12. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA 1990: Proceedings of the European Conference on Object Oriented Programming Systems Languages and Applications, pp. 303–311. ACM, New York (1990)
13. Costanza, P., Hirschfeld, R.: Language Constructs for Context-oriented Programming: An Overview of ContextL. In: Proceedings of the 2005 Symposium on Dynamic Languages, pp. 1–10. ACM Press, New York (2005)
14. Costanza, P., Hirschfeld, R., De Meuter, W.: Efficient Layer Activation for Switching Context-dependent Behavior. In: Lightfoot, D.E., Szyperski, C. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 84–103. Springer, Heidelberg (2006)
15. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A Mechanism for Fine-Grained Reuse. ACM Trans. Program. Lang. Syst. 28(2), 331–388 (2006)
16. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, pp. 1–18. ACM Press, New York (2007)
17. Eugster, P., Jayaram, K.R.: EventJava: An Extension of Java for Event Correlation. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 570–594. Springer, Heidelberg (2009)
18. Filman, R.E., Elrad, T., Clarke, S., Aksit, M. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
19. Gonzlez, S., Mens, K., Cdiz, A.: Context-Oriented Programming with the Ambient Object System. Journal of Universal Computer Science 14(20), 3307–3332 (2008)
20. Hedin, G., Magnusson, E.: JastAdd: An Aspect-oriented Compiler Construction System. Science of Computer Programming 47(1), 37–58 (2003)
21. Hirschfeld, R., Costanza, P., Haupt, M.: An Introduction to Context-Oriented Programming with ContextS. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007 II. LNCS, vol. 5235, pp. 396–407. Springer, Heidelberg (2008)
22. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. Journal of Object Technology 7(3), 125–151 (2008)
23. Hirschfeld, R., Perscheid, M., Schubert, C., Appeltauer, M.: Dynamic contract layers. In: 25th Symposium on Applied Computing, Lausanne, Switzerland. ACM DL, New York (2010)
24. Steele Jr., G.L.: Common LISP: The Language, 2nd edn. Digital Press, Newton (1990)

25. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
26. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
27. Núñez, A., Noyé, J., Gasiũnas, V.: Declarative Definition of Contexts with Polymorphic Events. In: COP 2009: International Workshop on Context-Oriented Programming, pp. 1–6. ACM Press, New York (2009)
28. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
29. Schippers, H., Haupt, M., Hirschfeld, R., Janssens, D.: An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In: Proc. SAC PSC. ACM Press, New York (2009)
30. Smith, R.B., Ungar, D.: Programming as an experience: The inspiration for self. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 303–330. Springer, Heidelberg (1995)
31. von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented Programming: Beyond Layers. In: Demeyer, S., Perrot, J.-F. (eds.) ICDL 2007: Proceedings of the 2007 International Conference on Dynamic Languages. ACM International Conference Proceeding Series, vol. 286, pp. 143–156. ACM Press, New York (2007)

Predicated Generic Functions
Enabling Context-Dependent Method Dispatch

Jorge Vallejos[1], Sebastián González[2], Pascal Costanza[1], Wolfgang De Meuter[1], Theo D'Hondt[1], and Kim Mens[2]

[1] Software Languages Lab – Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{jvallejo,pascal.costanza,wdmeuter,tjdhondt}@vub.ac.be
[2] Département d'ingénierie informatique – Université catholique de Louvain
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium
{s.gonzalez,kim.mens}@uclouvain.be

Abstract. This paper presents predicated generic functions, a novel programming language abstraction that allows the expression of context-dependent behaviour in a declarative and modular manner, providing fine-grained control of method applicability and method specificity. Methods are guarded by predicates with user-defined orderings, thereby increasing the expressiveness of existing method dispatching approaches. We have prototyped our proposal in Lambic, an extension of the standard Common Lisp Object System. We illustrate and motivate our approach by discussing the implementation of a collaborative graphical editor.

1 Introduction

The lack of linguistic support for encoding context-dependent behaviour forces programmers to scatter these dependencies throughout application code in the form of conditional statements. In object-oriented programming, ad hoc polymorphism alleviates this problem by means of dynamic method dispatch, enabling behavioural variations based on a *receiver* argument. Methods and their overriding relationships are driven by the inheritance hierarchies of the objects received as arguments. Although an improvement, object-oriented dispatch has been found to be limiting in many situations. A number of solutions have been proposed, ranging from design patterns and multiple dispatch to those based on metaobject protocols and aspects. Still, most approaches do not offer linguistic abstractions to influence the semantics of method dispatch based on more general criteria, while still preserving the benefits of encapsulation and polymorphism —with one remarkable exception, *predicate dispatch* [11].

Predicate dispatch offers fine-grained control on method applicability by means of logical predicates. Logical implication between predicates defines the overriding relationship between corresponding methods. However, predicate dispatching has limited capacities to resolve method overriding ambiguities since the logical implications between predicates cannot be decided in the general case. Thus, the set of method predicates must be restricted to a well-chosen subset

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 66–81, 2010.
© Springer-Verlag Berlin Heidelberg 2010

and thus can be statically analysed. This leads to a viable approach, but can be limiting in some circumstances, since users cannot extend predicate dispatching with their own arbitrary predicates in a straightforward way. Predicates without logical implications can still be added but they are treated as black boxes and the overriding relationship between two syntactically different expressions is considered ambiguous [5,11].

This paper proposes a generic function-based multiple dispatch mechanism, called *predicated generic functions*, that alleviates the restrictions of predicate dispatching to cope with method overriding ambiguities. Instead of requiring a logical implication order between predicates, this model fosters the definition of context-specific priorities. Predicated generic functions enable users to establish a priority order between logically unrelated predicates. These priorities are specified in a per-generic-function basis: predicated generic functions contain not only the methods with a common name and argument structure (as in standard generic function models [3,7]), but also the predicates on which such methods can be specialised. A method is selected for execution when its predicate expression is satisfied, and the order of the predicates specified in a generic function determines the order of applicability of its methods.

We implement the mechanism of predicated generic functions in *Lambic* [23], a prototype extension of the Common Lisp [20] programming language. We illustrate the benefits of predicated generic functions by developing a scenario of context-aware computing. Lambic allows application developers to modularise behavioural adaptations in methods and declaratively specify the context conditions for these adaptations as predicates. Manual definition of predicate priorities in generic functions provides developers with fine-grained control over the composition of adaptations, ensuring that the "most suitable" composition of behaviour is selected by the dispatch mechanism for any given method call.

2 Motivation: Context-Dependent Behaviour

Context dependency is the ability of software services to perceive and react to changes in their execution environment, adapting their behaviour accordingly [15]. This ability is already an integral part of some business applications, but it is becoming even more critical in application domains such as mobile and ubiquitous computing, in which context adaptability requirements play a central role.

In this section we show the need for language constructs that ease the expression of context-dependent behaviour, discussing the suitability of existing object-oriented approaches to cope with this requirement. We use as running example a distributed graphical editor called *Geuze*. This editor can be used in different hosts to work collaboratively on a same graphical document. For the sake of simplicity, in this section we focus only on the graphic user interface of Geuze, and discuss other cases of context dependency we have found in the implementation of this editor, in Section 5.

Table 1. Context conditions for Geuze operations

Context	GUI events		
	Mouse down	**Mouse move**	**Mouse up**
Painting shape	shape found, brush selected	—	—
Moving shape	shape found, brush not selected	drag status found, brush not selected	drag status found, brush not selected
Drawing shape	shape not found, brush selected	drag status not found	line status found
Drawing selection	—	brush not selected, drag status not found	brush not selected, drag status not found
Selecting shape	shape found	—	—
Deselecting shape	shape not found	—	—

2.1 Handling User Interface Events

Consider some of the main graphical operations that Geuze can perform. The editor allows creating, selecting, moving and painting shapes in a canvas. Each operation has its own interaction pattern defined in handlers for Graphical User Interface (GUI) events. A pattern can require a combination of GUI events, and the same GUI event can be used in several patterns. For instance, the operation for painting a shape is defined in a handler for the *mouse-down* event (clicking inside a shape with the mouse pointer paints the shape). The operations for drawing and moving a shape follow a drag-and-drop pattern, which is specified in handlers for the *mouse-down*, *mouse-move* and *mouse-up* events.

The context of use plays a key role, as it determines the operation that handles an event. This is illustrated in Table 1. For example, the operation that should be executed upon a *mouse-down* event depends on context information such as the x and y coordinates of the mouse pointer, on the *shape* found at these coordinates (if any) and on the state of the editor's *brush* (whether the brush button is selected or not). Depending on this information, the operation corresponding to the *mouse-down* gesture might be painting, moving and so forth.

The mode of operation (*painting*, *moving*, etc.) is also part of the context. Pressing the mouse button will trigger a different set of actions depending on the currently active operation, as illustrated in Table 2. For instance, a same *mouse-move* event provokes a displacement of the shape when the editor is in *moving* mode, whereas a line is drawn —a completely different behaviour— when the mode is *drawing*. Further, there are cases in which two operations correspond to the same *mouse-down* event. If a deselected shape is painted, the shape is first selected, and then painted. This order of operations, although not apparent in Tables 1 and 2, is integral part of the normal behaviour of the editor and needs to be properly encoded. We will come back to this when we discuss the implementation in Lambic.

Table 2. Actions for Geuze operations

Context	Actions		
	Mouse down	**Mouse move**	**Mouse up**
Painting shape	paint shape	—	—
Moving shape	set drag status	update drag status, move shape	delete drag status
Drawing shape	set line status, draw initial point	update line status, draw line	delete line status, create a shape object out of the drawn line
Drawing selection	—	draw selection square, select found shape	remove selection square
Selecting shape	select shape	—	—
Deselecting shape	deselect shape	—	—

2.2 Design Analysis of Running Example

At first sight, programming a basic graphical editor such as the one described previously is straightforward. Nevertheless, a detailed analysis of the possible solutions reveals that the programming tools we have at hand today fall short of expressiveness to allow the production of a cleanly modularised solution.

Naive solution. The most immediate implementation of the handler for the *mouse-down* event is through conditional statements. Such a handler would squeeze all the information contained in Tables 1 and 2 into one monolithic piece of code: both the conditions that are necessary for execution of context-specific behaviour, and the behaviour itself, for every operation of the editor that is concerned by the *mouse-down* event. The downsides are clear —context conditions would be hard coded using conditional statements, and the implementations of the different context actions would be tangled in the handler. This solution is unacceptable, as it hinders maintainability and code reuse [16].

Object-oriented solution. In traditional object-oriented systems, method invocation is triggered by messages being sent to objects where the objects then decide which method to execute based on a mapping from message signatures to actual methods. Typically, such mappings are fixed for specific types of objects, which means that the dynamic state of a running system cannot (easily) influence the dispatch mechanism for a particular object anymore.

One solution is to use forwarding, which means that an object that receives a message forwards it to another object, based on some arbitrary criteria. A popular example of that approach is the State pattern [12], which enables separation of method definitions according to the state of a particular receiver object. Fig. 1 shows a diagram for a possible use of the State pattern in our example. Using this architecture, the editor may behave differently according to whether it is in the state *painting*, *moving*, and so forth.

A drawback of message forwarding is that it introduces object identity problems: the `self` or `this` reference in the method corresponding to the current state is not the original receiver of the message (note in Fig. 1 that the first

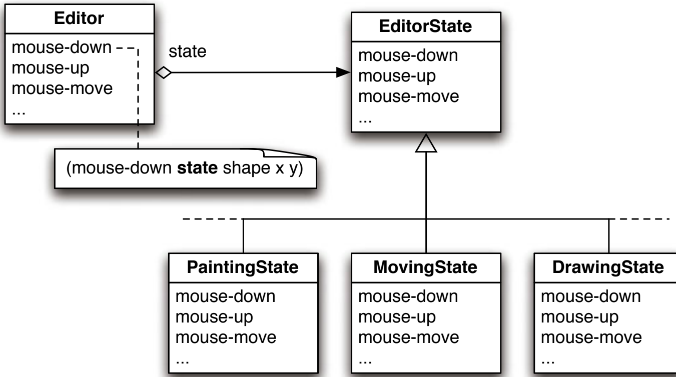


Fig. 1. Architecture of the Geuze editor using the State pattern

argument passed to *mouse-down* is the `state` attribute of the editor). This is typically referred to as the object schizophrenia problem [6]. There are a number of suggestions to solve certain aspects of this problem, for example by rebinding `self` or `this` to the original receiver in delegation-based languages [17], or by grouping delegating objects in *split objects* and letting them share a common object identity [1]. However, the core problem remains, namely that it is not straightforward to unambiguously *refer* to a single object anymore. The programmer must ensure that the right object in a delegation chain is being referred to, and even in split objects, the correct *role* of an object has to be selected.

Predicate dispatching. Predicate dispatching [11] is very convenient for the declaration of *method applicability* constraints: the conditions —or context— under which a method *can* be invoked. However, it falls short in helping to express *method specificity* in the general case —that is, when methods *should* be invoked, taking precedence over other applicable methods. Precedence by logical implication is a natural choice, but it sometimes cannot be established by mere static analysis of the predicates. Unfortunately, predicate dispatching does not support user-defined orderings of predicates for cases that cannot be decided solely on the grounds of the structure of the predicates. Furthermore, automatic disambiguation of methods by means of logical implication does not always yield the desired semantics. For instance, in Table 1 the condition for *moving shape* is stronger than (implies) that of *selection*. Predicate dispatching will thus consider moving behaviour more specific than selection behaviour. However, in Geuze the selection behaviour must be performed *before* the moving behaviour.[1] Lambic allows to encode this semantics, as will be shown in the explanation of Listing 1.

In this section we have shown that it is difficult to cleanly encode context-dependent behaviour within the traditional object-oriented paradigm. We have identified predicate dispatching as a possible solution path, but have discussed

[1] This requirement has to do with the distributed part of the editor, not explained in this paper. The selection of a shape is used for control access. Selecting a shape means obtaining a lock from the leader that coordinates the interaction in the network.

its shortcomings when it comes to defining method specificity. Next we introduce our answer to still use the underlying idea of predicated applicability, albeit in an adapted form that also allows fine-tuning of specificity.

3 Predicated Generic Functions

Predicated generic functions are an extension of the generic function-based mechanism of the Common Lisp Object System (CLOS) [3]. CLOS supports multiple dispatch semantics by detaching methods from classes, allowing developers to specialise methods on the classes of all received arguments, as opposed to only the first argument in singly dispatched languages. We extend such mechanism by enabling methods to also specialise on predicates. In this section, we briefly explain the syntax and informal semantics of predicated generic functions, showing a small example of use. To this end, we use *Lambic* [23], our extension of the Common Lisp programming language that implements predicated generic functions. In the remainder of this paper we refer to predicated generic functions and to Lambic interchangeably.

3.1 Syntax and Semantics

This section describes the very essentials of Lambic's syntax and semantics for the definition and invocation of predicated generic functions and methods. Further details of this mechanism are available in Section 5.

Defining Generic Functions and Methods. In Lambic, as in CLOS, a generic function is a container of methods with a common name and a parameter list the methods can specialise. Additionally, Lambic's generic functions can contain a list of predicate declarations. The definition of such generic functions follows the syntax:

(**defgeneric** *function-name function-parameters*
[(**:predicates** {*pred-symbol* | (*pred-name pred-params pred-body*)}*)])

A generic function is defined with the `defgeneric` construct which receives as arguments a name, a parameter list and an optional list of predicate declarations (denoted with the `:predicates` keyword). Predicates are standard (CLOS) functions with a boolean-valued expression as body, following an arbitrary user-defined specificity order represented by the order of the predicate declarations in the generic function: the first predicate of the list is the most general predicate and the last one the most specific. A predicate can be defined either outside or inside the generic function. In the former case it suffices to indicate only the symbol associated to the predicate's definition (which can correspond to an already existing function). Internally defined predicates, on the other hand, should indicate the predicate's name, parameter list and body. These internal predicates are available exclusively for the methods belonging to the generic function.

Methods are defined independently from their containing generic functions, using the `defmethod` construct:

(**defmethod** *method-name method-parameters*
[(**:when** {(*pred-name arguments*)}*)]
method-body)

A method is defined with a name, a parameter list and an optional predicate expression (specified with the `:when` keyword). This expression is composed of one or more invocations to the predicates declared in the method's generic function, using one or more parameters of the method as arguments.

Invoking Generic Functions. In Lambic, as in CLOS, object-oriented programs are written in terms of generic function invocations rather than messages exchanged between objects. Yet, both approaches result in the invocation of a method or a method chain. When a generic function is called with particular arguments, it selects the methods to be executed —known as the *applicable methods*— by evaluating each of the method's predicate expressions. This evaluation occurs in the lexical environment of the method, augmented with the generic function's parameters bound to the received arguments. The applicable methods are those whose predicate expression evaluates to `true`, and the methods that do not specify any predicate.

The execution order of the applicable methods is determined by the specificity of their predicates: the method with the most specific predicate is executed first. A method without predicate expression is considered more general than any method with predicates and thus it is always executed at last.

Finally, the most specific method is called. The other methods can be invoked by the programmer inside of method definitions by way of `call-next-method`, much like with super calls in other object-oriented languages.

3.2 Example of Use

Consider as illustrative example the factorial function. In this function we want to distinguish between negative and positive numbers, and the number zero. We therefore define a `factorial` generic function using as predicates the functions for aritmethic comparison `<`, `=`, and `>`. Since these are already defined in Common Lisp, we just need to declare them as predicates for the `factorial` generic function, indicating the corresponding symbols as follows:

```
(defgeneric factorial (n)
  (:predicates < = >))

(defmethod factorial (n)
  (:when (> n 0))
  (* n (factorial (- n 1))))

(defmethod factorial (n)
  (:when (= n 0))
  1)

(defmethod factorial (n)
  (:when (< n 0))
  (error "Factorial not defined for negative numbers."))
```

Each method indicates one of the predicates declared in the generic function using the `:when` keyword. The first method is called if the argument `n` is a positive number and computes the general case of the factorial function. The second method is called if `n` is 0 and returns 1. The third method will be called if `n` is a negative number and signals an error.

4 The Geuze Editor in Lambic

We now describe the development in Lambic of the scenario introduced in Section 2, the Geuze drawing editor. First, we focus on the implementation of a GUI event handler to illustrate the benefits of the predicated generic functions in terms of modularity and reusability. We then show that by supporting method dispatch based on the state of received arguments, Lambic enables expressing State-like patterns without object identity problems.

4.1 Context-Dependent Event Handlers

Listing 1 shows the implementation in Lambic of the actions that handle the *mouse-down* event in the Geuze drawing editor. Each of these actions correspond to (part of) a graphical operation, which is selected depending on a number of context conditions, detailed in Table 1. In Lambic, we can cleanly separate such actions in methods specialised on predicates representing the different graphical operations. These methods act as *context-dependent* handlers which can conveniently associate actions to context conditions. For instance, the first method definition in Listing 1 describes how to handle the *mouse-down* event when the editor is in the *painting* context (indicated by means of the `painting` predicate).[2]

All the methods in Listing 1 are contained in the `mouse-down` generic function which declares the predicates with the context conditions for the graphical operations. Listing 2 shows the definition of such generic function with the `moving` and `drawing` predicates defined internally. The other three predicates are defined as external functions as in the case of the `selecting` predicate shown in Listing 2. Section 4.2 explains the reason for this difference in the declaration of the predicates.

Combining Context-dependent Handlers. In Section 2, we discuss the situation in which an event is handled by more than one operation, e.g. painting a deselected shape results in the shape first being selected and then painted. In Lambic, the selection and proper combination of methods is internally computed in accordance to the predicates and their order of declaration in the generic function. Hence, in the Geuze editor, such case is transparently handled by the `mouse-down` generic function, which selects for execution the methods defined for selecting and painting shapes (shown in Listing 1), denoted with the `selecting` and `painting` predicates respectively. Since the `selecting` predicate is declared

[2] For the sake of clarity, the handling action in this case is reduced to the invocation of a `paint-shape` method.

Listing 1. Untangled GUI event handlers in Lambic. Application concerns are indicated on the right side.

```
(defmethod mouse-down (editor shape x y)
  (:when (painting shape editor))                    → context
  (paint-shape editor shape))                        → painting

(defmethod mouse-down (editor shape x y)
  (:when (moving shape editor))                      → context
  (set-drag-status editor x y))                      → moving

(defmethod mouse-down (editor shape x y)
  (:when (drawing shape editor))                     → context
  (set-line-status editor x y)
  (draw-point editor x y))                           → drawing

(defmethod mouse-down (editor shape x y)
  (:when (selecting shape editor))                   → context
  (select-shape editor shape)                        → selection
  (call-next-method))

(defmethod mouse-down (editor shape x y)
  (:when (deselecting shape editor))                 → context
  (deselect-shapes editor))                          → deselection
```

more specific than the `painting` predicate (the latter predicate appears first in the list of predicates of the `mouse-down` generic function), the method for selecting the shape is executed first. Finally, as we explained in Section 3, by default only the most specific method is executed. Therefore, we need to include the `call-next-method` construct in the method specialised on the `selecting` predicate, so that the one specialised on `painting` is invoked next.

4.2 State Pattern without Object Schizophrenia

Lambic enables developers to express State-like patterns without object identity problems. As example, assume that the graphical operations of the Geuze editor represent its different states, just like in the original State pattern. Each state groups the behaviour required by an operation to handle one or more GUI events, as detailed in Table 2 of Section 2. In Lambic, this corresponds to describing a state as a number of methods using the same predicate expression. Simple examples are the states representing the operations for selecting, deselecting and painting shapes, which only require one method definition to handle the *mouse-down* event, shown in Listing 1. A bigger example is the *moving* state presented in Listing 3 which defines its behaviour in the methods `mouse-down` (to set a drag status used during the move), `mouse-move` (to move the shape) and `mouse-up` (to remove the drag status at the end of the move).

Notice how this way of specifying the behaviour of the Geuze editor cleanly separates the definition of its several states (embodied by the predicates) from the behaviour corresponding to those states. This State-like idiom avoids any

Listing 2. The `mouse-down` generic function.

```
(defgeneric mouse-down (editor shape x y)
  (:predicates painting
               (moving (shape editor)
                 (and shape (not (brush-active? editor))))
               (drawing (shape editor)
                 (and (not shape) (brush-active? editor)))
               selecting
               deselecting))

(defun selecting (shape)
  shape)
```

Listing 3. The *moving* state.

```
(defmethod mouse-down (editor shape x y)
  (:when (moving shape editor))
  (set-drag-status editor x y))

(defmethod mouse-move (editor shape x y)
  (:when (moving shape editor))
  (move-shape shape editor x y))

(defmethod mouse-up (editor shape)
  (:when (moving shape editor))
  (delete-drag-status editor))
```

object identity problems: a particular editor always retains its identity, no matter what state it is in. Since the state of the editor is automatically derived from the current context conditions, one does not have to worry about managing an explicit state with explicit state switches in the corresponding *mouse-down*, *mouse-move* and *mouse-up* event handlers.

Finally, note in Section 2 that Table 1 identifies different context conditions that characterise the operation for moving a shape at the different mouse events. In particular, there is a set of conditions for the *mouse-down* event and another set for the *mouse-move* and *mouse-up* events. However, we can still define the *moving* state in terms of the three methods using the `moving` predicate. In Lambic, this is possible by enabling predicates to be defined inside the generic functions which are only available to the generic function's methods. Thus, the `mouse-move` and `mouse-up` generic function refer to a globally defined `moving` predicate (as shown in Listing 4) whereas the `mouse-down` generic function defines its own version of such predicate (shown in Listing 2). This example makes clear that the State pattern in Lambic is mostly a naming convention for the predicate used to identify the state. Further, this example shows the fine-grained control that developers have to influence the applicability of methods based on the context.

Listing 4. Reuse of the `moving` predicate.

```
(defun moving (shape editor)
  (and (drag-status? editor) (not (brush-selected?))))

(defgeneric mouse-move (editor shape x y)
  (:predicates ... moving ...))

(defgeneric mouse-up (editor shape x y)
  (:predicates ... moving ...))
```

5 Validation and Discussion

Lambic's predicated generic functions allow methods to specialise on programmer-defined predicates, providing fine-grained control of method applicability, as predicate dispatching also does. Additionally, method dispatch is driven by the predicates' specificity order declared in generic functions which avoids the problems caused by potential ambiguities when comparing arbitrary predicates that do not designate instance subsets of each other.

We have illustrated the benefits of predicated generic functions by discussing the implementation of the graphical user interface of the Geuze editor. However, this is only part of our current implementation of Geuze.[3] As mentioned in Section 2, Geuze is an application for collaborative edition that enables its users to create peer-to-peer drawing sessions. Predicated generic functions have contributed significantly to define context-dependent behaviour for this application, allowing a clean separation of the code required for collaborative work from the plain editor logic, a clear distinction between the behaviour for the roles required for the coordination of the session (the session leader and the rest of the participants), and the modularisation and dynamic composition of the graphical operations (which is discussed in this paper). In the implementation of Geuze we have used 4 instances of the State-like idiom described in Section 4.2 to handle events related to network connection (to deal with network failures), synchronisation (to control edition locks), replication (to ensure a consistent collaborative edition), and graphic user interface (partially shown in Section 4.2). None of these instances require additional infrastructure beyond the methods associated to states by means of context predicates. Geuze is composed of 44 methods grouped in 16 generic functions. None of these methods contain entangled concerns in their body.

5.1 Predicated Generic Functions and CLOS

In the current implementation of predicated generic functions in Lambic we have adopted a rather conservative approach which preserves the method dispatching

[3] The full implementation is available at `http://soft.vub.ac.be/lambic`.

semantics of CLOS. For instance, Lambic still enables the methods to specialise on the classes of the arguments, and to use qualifiers (e.g. `:around`, `:before` and `:after`). We achieve this compatibility by reflectively introducing our predicate dispatch mechanism as an internal step in the method selection and ordering process, leaving unchanged the rest of the semantics of CLOS that computes applicable methods and their ordering. Actually, predicate expressions correspond to implicit generic function arguments which are added at the end of the parameter list. Each predicate is associated to a type, and the predicate ordering is encoded by means of subtyping. Given this encoding, behaviour selection is performed following normal CLOS dispatch semantics. In particular, a generic function containing methods specialised on classes and predicates, selects and sorts the methods according to the classes first, and then according to predicates. Our current implementation of Geuze[3] illustrates this case.

5.2 Limitations

Although Lambic can help in tackling some of the challenges for modelling generic functions with context-specific predciates, a number of challenging issues needs to be further explored. Currently, predicated generic functions are implemented using the Meta-Object Protocol (MOP) of CLOS, and can be used only in the LispWorks® Enterprise Edition[4] development environment for Common Lisp. We have not considered efficiency issues in detail yet, and have not explored more general implementation techniques. However, efficient implementation techniques for generalised predicate dispatch have been investigated in detail in the past [5] and can probably be adapted to the implementation of predicated generic functions as well.

In this paper, we propose an alternative to logical implication order used by existing predicate dispatching approaches to disambiguate method overriding. However, there are situations in which such approach would still be desired, e.g. to disambiguate methods using the same predicate expression. For instance, using predicated generic functions the method definitions

```
(defmethod foo (n)
  (:when (> n 1))
  (print "Number bigger than 1"))

(defmethod foo (n)
  (:when (> n 2))
  (print "Number bigger than 2"))
```

would lead to an ambiguous situation if `foo` is invoked with `n` greater than 2, as both methods would be selected for execution but none of them is more specific than the other. While Lambic avoids this problem by accepting only one method definition with the same predicate expression, this is clearly a case in which the inclusion of logical implication in Lambic would increase its expressiveness.

[4] See http://www.lispworks.com.

6 Related Work

We divide related techniques in two categories, according to whether they promote flexible method dispatch or flexible software composition to enable behaviour adaptability.

6.1 Flexible Behaviour Selection Schemes

Predicated generic functions build on previous work on filtered dispatch [9], which in turn draws inspiration from specialisation-oriented programming [19]. Filtered dispatch introduces filter expressions that map actual arguments to representatives. Filtered arguments are then used in place of the original arguments for method selection and combination. The chosen method is invoked using the original arguments.[5] As Lambic's predicates, filters are associated to generic functions. One of the motivating examples for developing filtered dispatch has been the implementation of an interpreter for a Lisp dialect [14]. In this interpreter, values of the interpreted language are represented in the standard way, as wrapped values in the implementation. However, the object-oriented implementation of the interpreter needs to dispatch on the unwrapped values, not on the wrappers themselves. Using filtered dispatch, this can easily be expressed by turning the unwrap function into a filter.

Predicated generic functions and filtered dispatch are closely related, albeit there are examples which are easily expressed in Lambic that cannot be expressed using filtered dispatch, and the other way around. Firstly, even though many filters can be defined for a given generic function in filtered dispatch, corresponding methods can use only one of those filters at a time. As a consequence, each possible combination of the filters that could prove useful needs to be anticipated and encoded as an additional filter in the generic function. Secondly, filtered expressions are parameterised exclusively on the argument they filter; they cannot depend on the value of other arguments of the method. This restriction renders filtered dispatch less amenable to express context adaptations, because the conditions for applicability (the predicates) cannot harness all available contextual information. On the other hand, the example where an interpreter dispatches on unwrapped interpreter values [9,14], seems to be less straightforward to do in Lambic.

Mode classes [21] enable dispatching on the explicit state of an object. Predicate classes [4] extend this idea by dispatching on computed states. Mode classes correspond to an explicit management of state, while predicate classes compute state implicitly. Predicate classes were a precursor to generalised predicate dispatch [11], discussed in Section 2.2.

Clojure[6] is a recent dialect of Lisp that provides a generic dispatch framework that can accommodate many different semantics. A Clojure multimethod

[5] This corresponds to the *lookup* ∘ *apply* decomposition of method dispatch investigated by Malenfant et al. [18]. In filtered dispatch, *lookup* receives the filtered arguments, and *apply* receives the unfiltered (original) ones.

[6] See `http://clojure.org`.

is a combination of a *dispatching function* and one or more *methods* with the same name. The dispatching function maps passed arguments to arbitrary values which are associated to methods. The ordering of values combines Java's typing mechanism with Clojure's own ad-hoc hierarchy system. When no method dominates the others, Clojure provides a means to manually disambiguate multiple matches. An implementation of predicated generic functions in the dispatch framework of Clojure seems feasible.

Ambience [13] proposes a prototype-based object system that reifies the context as an object, and exploits multiple dispatch to enable selection of context-dependent behaviour. Ambience does not propose however a predefined mechanism to choose which context should be activated; in contrast, in Lambic the conditions that enable different behaviours are readily encoded in the method predicates.

6.2 Dynamic Composition Schemes

Classboxes [2] are a mechanism for lexically-scoped structural *refinements*. Similar to open classes, refinements in classboxes make it possible to extend a class definition from the outside, with new fields and methods, as well as extending methods with a mechanism similar to overriding in standard object-oriented programming. However, while changes made with open classes are globally visible, classboxes introduce a dynamic scoping mechanism based on import relations between modules: a refinement to a class is only visible for all execution that originates from a client of the module in which the refinement is defined.

Aspect-Oriented Programming (AOP) [16] allows the programmer to encapsulate concerns that cross-cut modularisation boundaries (e.g. classes) in a construct called an *aspect*. Aspects are designed to supplement the basic composition mechanisms provided by the host language. There are three main binding times for aspects: compile time, load time and run time. Dynamic aspect weaving occurs at run time. Dynamic aspect weaving can be thought of as a tool for dynamic behaviour adaptation, since it allows for base application logic to change at run time —for instance, to be adapted to non-functional concerns such as security or low power computation. Dynamic aspects can be woven and unwoven according to context. Context-Aware Aspects explore this idea [22]. The idea is to extend pointcut languages with context-specific restrictions, allowing both parameterization of context definitions and exposure of context state to the aspect action. This and other aspect weaving approaches are related to ours insofar as the applicability of advice is determined by dynamically evaluated pointcut definitions —advice is analog to filtered methods, and pointcuts are analog to predicates which limit the applicability of advice to specific contexts.

Costanza [8] argues that the advantages of (dynamic) AOP can be obtained with less conceptual and technical burden thanks to dynamic scoping of functions. This idea has been integrated reflectively into CLOS, and the result is ContextL [10]; ContextL features a form of dynamic scoping of methods to enable adaptation to context.

7 Conclusion and Future Work

In this paper we have introduced *predicated generic functions*, a novel mechanism to allow flexible behaviour selection according to context. This mechanism has been realised in Lambic, an extension of CLOS. Lambic method definitions can be guarded by predicates, which are used to decide on the applicability of the method for a list of actual arguments. If more than one predicated method is applicable, the order in which the predicates are declared in the corresponding generic function is used as tiebreaker. These are the main tools Lambic offers for fine-grained control of applicability and specificity of methods.

The development of Geuze, a non-trivial application for the collaborative edition of graphical objects, has shown us that predicated generic functions allow the modular implementation of various concerns. The modes of operation of the application (whether it is working on *painting*, *moving* or some other mode) can be regarded as contexts, and behaviour is specialised on such contexts.

The assessment of predicated generic functions presented in this paper leads us to believe that they are well suited to the expression of behaviour that depends non-trivially on context. However, some work remains. Among our next steps, we plan to look into efficient implementation techniques to avoid unnecessary computation when deciding method applicability. Also, we will consider the formalisation of the semantics of predicated generic functions. Finally, we will continue exploring the possibilities of predicated generic functions in combination with distribution —one of the flagships of Lambic, offering many possibilities yet to be explored.

Acknowledgements. This work has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), and by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State, Belgian Science Policy. We thank Charlotte Herzeel and the anonymous reviewers for their valuable comments on this paper.

References

1. Bardou, D., Dony, C.: Split objects: A disciplined use of delegation within objects. ACM SIGPLAN Notices 31(10), 122–137 (1996)
2. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: Controlling visibility of class extensions. Journal of Computer Languages, Systems and Structures 31(3), 107–126 (2005)
3. Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., Moon, D.: Common Lisp Object System. specification. Lisp and Symbolic Computation 1(3/4), 245–394 (1989)
4. Chambers, C.: Predicate classes. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 268–296. Springer, Heidelberg (1993)
5. Chambers, C., Chen, W.: Efficient multiple and predicated dispatching. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 238–255. ACM Press, New York (1999)

6. Chandra Sekharaiah, K., Janaki Ram, D.: Object schizophrenia problem in object role system design. In: Bellahsène, Z., Patel, D., Rolland, C. (eds.) OOIS 2002. LNCS, vol. 2425, pp. 1–8. Springer, Heidelberg (2002)
7. Clifton, C., Leavens, G., Chambers, C., Millstein, T.: MultiJava: Modular open classes and symmetric multiple dispatch for Java. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 130–145. ACM Press, New York (2000)
8. Costanza, P.: Dynamically scoped functions as the essence of AOP. ACM SIGPLAN Notices 38(8), 29–36 (2003)
9. Costanza, P., Herzeel, C., Vallejos, J., D'Hondt, T.: Filtered dispatch. In: Proceedings of the Dynamic Languages Symposium. ACM Press, New York (2008), co-located with ECOOP 2008
10. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of the Dynamic Languages Symposium, October 2005, pp. 1–10. ACM Press, New York (2005), co-located with OOPSLA 2005
11. Ernst, M., Kaplan, C., Chambers, C.: Predicate dispatching: A unified theory of dispatch. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 186–211. Springer, Heidelberg (1998)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley, Reading (1995)
13. González, S., Mens, K., Heymans, P.: Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In: Proceedings of the Dynamic Languages Symposium, October 2007, pp. 77–88. ACM Press, New York (2007), co-located with OOPSLA 2007
14. Herzeel, C., Costanza, P., D'Hondt, T.: Reflection for the masses. In: Hirschfeld, R., Rose, K. (eds.) S3 2008. LNCS, vol. 5146, pp. 87–122. Springer, Heidelberg (2008)
15. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (2008)
16. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
17. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. ACM SIGPLAN Notices 21, 214–223 (1986)
18. Malenfant, J., Dony, C., Cointe, P.: A semantics of introspection in a reflective prototype-based language. In: LISP and Symbolic Computation, May 1996, vol. 9, pp. 153–179. Springer, Netherlands (1996)
19. Newton, J., Rhodes, C.: Custom specializers in object-oriented Lisp. Journal of Universal Computer Science 14(20), 3370–3388 (2008)
20. Steele, G.: Common Lisp: The Language, 2nd edn. Digital Press (1990)
21. Taivalsaari, A.: Object-oriented programming with modes. Journal of Object-Oriented Programming 6(3), 25–32 (1993)
22. Tanter, E., Gybels, K., Denker, M., Bergel, A.: Context-aware aspects. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 227–242. Springer, Heidelberg (2006)
23. Vallejos, J., Costanza, P., Van Cutsem, T., De Meuter, W.: Reconciling Generic Functions with Actors. In: ACM SIGPLAN International Lisp Conference, Cambridge, Massachusetts (2009)

Dynamically Adaptive Systems through Automated Model Evolution Using Service Compositions

Adina Mosincat, Walter Binder, and Mehdi Jazayeri

Faculty of Informatics, University of Lugano, Switzerland
{adina.diana.mosincat,walter.binder,mehdi.jazayeri}@usi.ch

Abstract. Runtime adaptability and delivered quality are two important concerns for every system. One way to achieve runtime adaptability is by specifying variants in the system model at design time to allow switching between runtime configurations. The fulfillment of system's quality requirements depends on parameters that can change at runtime. In order to meet its quality requirements, the system must be able to dynamically adapt to changes that affect the delivered quality. We outline our approach to enhance system adaptability through automatic evolution of the system model. Our approach periodically updates the model by re-evaluating the delivered quality based on runtime information. We use a service composition model to represent the system functional requirements and annotate it with delivered quality evaluations. We ensure system runtime adaptability by selecting the variant to execute at runtime based on the evolved model.

1 Introduction

Systems are increasingly required to feature autonomic capabilities. Systems that need to be continuously available and be able to change behavior depending on environmental conditions require a higher degree of autonomic capabilities. These systems must dynamically adapt to changes in the environment, switching between different runtime configurations without disrupting the running system. Software engineers can develop such dynamically adaptive systems by providing variants that determine the runtime configurations, respectively the execution paths of the system.

Quality requirements[1] are important concerns for every system. When reasoning about design decisions, software engineers must also ensure that the system will meet its quality requirements. The engineers face two issues:

1. The variants provided at design time are limited to the knowledge of the engineers and to their capability to foresee changes. Unanticipated changes in the environment or changes demanded by new user needs require modification and redeployment of the system.

[1] In this paper we use the term quality requirements for non-functional requirements, such as performance, reliability, and cost.

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 82–89, 2010.
© Springer-Verlag Berlin Heidelberg 2010

2. To estimate the fulfillment of the quality requirements, software engineers use parameters provided at design time that estimate the behavior of system components in a given environment. In a changing environment, parameter changes can result in quality degradation or violations of the system's quality requirements.

To address these issues, we automatically evolve the model of the system, periodically update it with observed quality information, and use the evolved model to drive the choice of the system runtime configuration. We present a framework for automated model evolution that uses a service composition model to represent the running system. The choice of the model allows us to leverage the advantages of service oriented architectures: flexibility, loose coupling, and ease of integration. Changes in our model are immediately incorporated in the implementation and runtime configuration. By using the model at runtime, our approach can easily integrate new variants and modify existing ones without disrupting the running system.

The Quality of Service (QoS) parameters of a service-oriented system depend on the QoS of its composing services. QoS estimates are expressed as guarantees, called service level objectives (SLO [1]), in service level agreements (SLA [1]). However, given the dynamic and distributed nature of service-oriented systems, QoS can vary in time and the system cannot enforce its composing services to meet their SLOs. Our framework monitors service execution and periodically re-evaluates the QoS parameters of the system based on monitoring information. It then updates the model with the new QoS values that are used for selecting the runtime configuration.

In this paper we outline our approach to enhance runtime adaptation capabilities for dynamically adaptive systems by updating the system model according to runtime information and using the updated model at runtime to drive the system runtime configuration. We give an overview of a possible implementation of our approach using service compositions and the BPEL standard.

2 Approach Overview

An important concept that we use in our approach is the *variant*. A variant represents one possible implementation of a system functional requirement. Variants can increase fault tolerance by specifying alternative ways of fulfilling a functional requirement. Take for example the case of the alarm function in a smart home system. The engineer specifies one variant of fulfilling the functionality by setting off the alarm, and a second variant by blinking the lights.

Fig. 1 presents the conceptual phases in our approach: (1) automatically evolving the system model based on runtime information; and (2) using the updated model at runtime to adapt the running system.

Initially, the delivered quality values of the system are estimated using parameters available at design time, such as QoS guarantees specified by SLAs. Due to system evolution and environmental changes, these parameters might change,

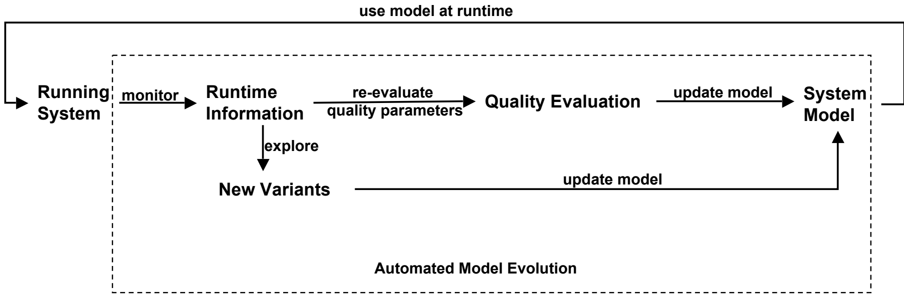


Fig. 1. Approach overview diagram

making the initially estimated quality values obsolete. By monitoring the running system, it is possible to gather information that reflects the current state of the environment. The quality values are re-estimated based on the monitoring information and the model is then updated according to the new estimations.

The algorithms used to estimate the quality values can help detect the system's inability to meet its quality requirements. The estimations can also help detect the cause of a quality degradation, such as a malfunctioning component. The system can improve the delivered quality and prevent violations of quality requirements by leveraging the estimations when selecting the variant to execute.

A second concern for model evolution are changes that provide a new variant, or that invalidate an existing variant. Consider for example the case of introducing a new component. New variants using the component become available. The model is updated by integrating the new discovered variants.

System adaptation is done by using the updated model to decide the system runtime configuration and the execution path for each functional requirement. To use this adaptation technique, a framework implementing our approach must ensure that implementation always conforms to the current model. The framework must provide a way to automatically implement the model changes into the running system without requiring user intervention or system interruption. The choice of the system model plays an important part in achieving the required autonomic capabilities.

3 A Framework Based on Service Compositions

This section presents a framework illustrating the approach introduced above and describes the system model and the components of the framework that play an important part in the automated model evolution and system adaptation.

3.1 Model

Fig. 2 presents the generation of the system model from the developer input model. In the input model the developer defines variability points, that is, functional (sub)requirements of the system which can have different variants. The

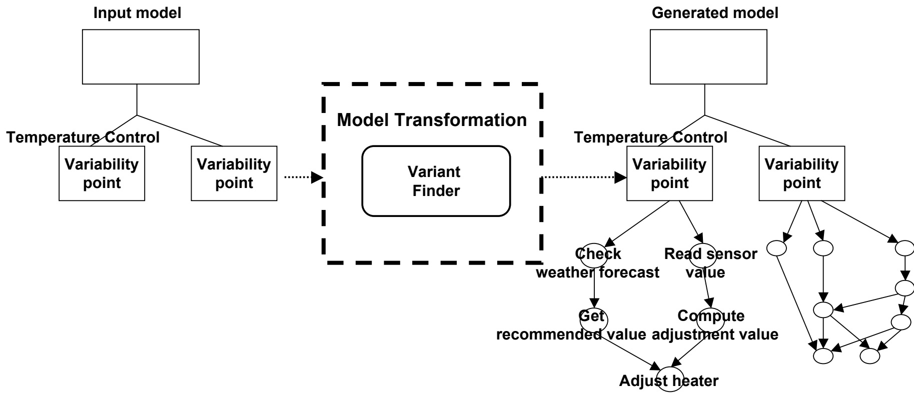


Fig. 2. Model transformation

Variant Finder generates the model of the running system from the input model by finding solutions to the variability points. A solution to a variability point is a variant.

The *Variant Finder* is an automated service composition engine that takes the variability point description and queries the service repository finding the set of service compositions that solve the requirement, such as [2,3]. The variability points must be expressed in a query language understood by the *Variant Finder*, and the services must be semantically annotated. In previous work we have introduced a query language for directories in support for automated service composition [4].

The input model must represent a system that can be implemented as a service composition. Currently we consider the input model to be a goal model with functional goals (functional requirements) as presented in [5]. A goal model is an AND/OR graph showing how higher-level goals are satisfied by lower-level ones (goal refinement) [6]. The AND-refinement link relates a goal to a set of subgoals that must be satisfied in order for the goal to be satisfied. A goal node can be OR-refined into multiple AND-refinements that each represent an alternative, i.e. the parent goal can be satisfied by satisfying the subgoals in any of the alternative AND-refinements. In our input model, the bottom subgoals must be queries in order to be able to automatically generate the system implementation. In this case, the whole system implementation is provided through automatic service composition.

The generated model is an annotated AND/OR graph in which the variability points are expanded with the variants found by the *Variant Finder*. A *variant node* is a node that is parent to at least one variant. All OR-link nodes and nodes corresponding to variability points in the input model are *variant nodes*. *Variant nodes* are annotated with information that is used at runtime to select the variant to execute. A *variant node* contains as data for each child variant a set of estimated QoS values and the deviation percentage from the required QoS. At model generation, the QoS values are computed based on the SLAs of the variant's composing services.

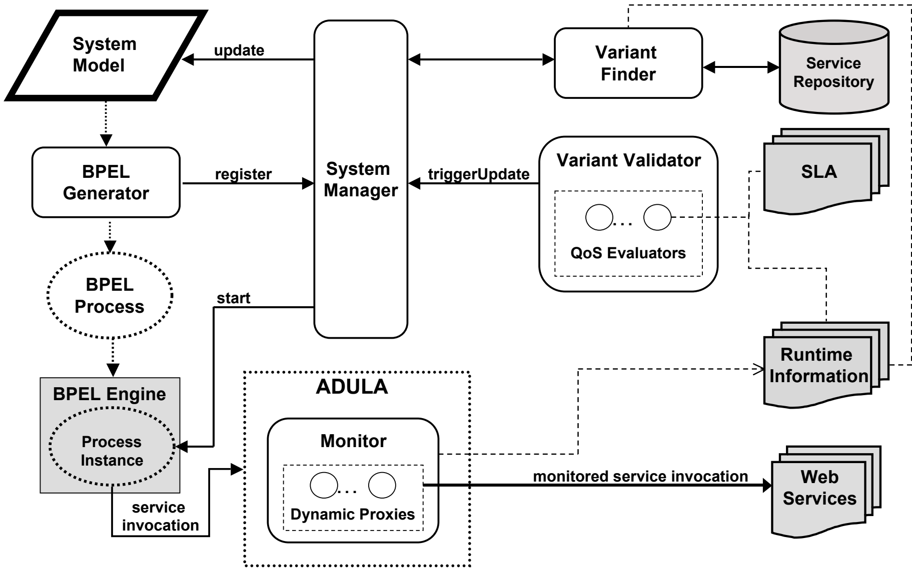


Fig. 3. Framework architecture

In Fig. 2 we show a fragment of the generated model for a smart home system. We detail the graph branch representing the temperature control functional requirement. The requirement is to adapt the temperature in the room according to weather conditions. The *Variant Finder* provides two variants for the temperature control variability point:

1. The system reads the weather forecast from an online service, gets the recommended temperature value for the external temperature forecast and sets the room heater to the recommended value.
2. The system reads the external temperature sensors installed on the building, computes the temperature value using a dedicated function and sets the room heater to the computed value.

3.2 Architecture

Fig. 3 presents the architecture of our framework. BPEL [7] is the de facto standard for service compositions. BPEL process definitions are deployed to BPEL engines that instantiate and execute a process instance when a request for the deployed process arrives. Our framework leverages the BPEL technology to implement the system.

The *System model* is the one introduced in Section 3.1. Our framework automatically generates BPEL processes from the system model. The processes are instrumented to allow monitoring the component services and the system QoS parameters. Based on monitoring information the framework estimates QoS values, periodically updating the model. The updated model is used to decide the runtime configuration.

The *BPEL Generator* creates a BPEL process for every variant in the model. All generated processes are registered to the *System Manager* (the arrow labeled **register** in the figure).

The *System Manager* is responsible with selecting the variants to be executed for fulfilling the system requirements. Some systems that are an aggregation of distributed applications, such as service-oriented systems, can be seen as a composition of functional requirements implemented independently. The system core is a dispatcher that forwards requests to the *System Manager*. We can automatically generate the dispatcher from the model. The developer can choose not to use the fully automated version, but only to forward requests for variability points execution to the *System Manager*.

The *System Manager* keeps a mapping between variants in the model and the corresponding BPEL processes. A variant is mapped to exactly one process. When the execution of a variability point is triggered, the *System Manager* checks the model for the variant to execute. The *System Manager* starts the variant execution by invoking the process corresponding to the selected variant (**start**). This approach allows the system to evolve with the system model, changes in the model being reflected in the system implementation.

Changes in the environment affect the system so that variants can become outdated, or new variants can be found. *Variant Finder* provides new possible variants based on runtime information, such as the availability of a new service. The *Variant Finder* updates the model with new variants, which can be then selected to be executed. In this way, new variants are easily integrated without disrupting the running system.

Our framework uses the monitoring capabilities of ADULA, a framework for fault tolerant execution of BPEL processes introduced in previous work [8,9]. The *Monitor* component of ADULA provides statistics on QoS parameters of services used by processes. Service invocations are redirected through *Dynamic Proxies* that measure the response time of the service. The *Monitor* observes the service execution, collects measurements and provides aggregated performance statistics.

Variant Validators test the fulfillment of the system quality requirements for each variant using statistic methods, such as Bayesian inference [10] or statistical hypothesis testing [11]. In previous work [9] we have used statistical methods to detect SLO violations for BPEL processes.

QoS Evaluators compute the estimated value of the QoS making use of monitoring information. Based on the estimated values, *Variant Validators* compute the deviation for each variant and each QoS parameter, and update the model with the new values (**triggerUpdate**). The deviation value is computed for each QoS using a provided function.

4 Related Work

KAMI [12] is a framework for model evolution by runtime parameter adaptation. KAMI focuses on Discreete Time Markov Chain [13] (DTMC) models that

are used to reason about non-functional properties of the system. The authors adapt the non-functional properties of the model using bayesian estimations to update the parameters that influence the non-functional properties. The estimations are computed based on runtime information, and the updated model allows verification of requirements. Our framework focuses on using the model at runtime to improve a system's autonomic capabilities. We also consider new variants for functional requirements and use the evolved model to dynamically adapt the system.

Similarly to KAMI, the approach in [14] considers the non-functional properties of a system in a web-service environment. The authors provide a language, SLAng, that allows to specify QoS to be monitored.

There are different approaches to provide self-adaptive systems. Models@Run.Time [15] propose leveraging software models and extending the applicability of model-driven engineering techniques to the runtime environment to enhance systems with dynamic adapting capabilities. The system adaptation in our approach leverages this idea using the model to determine the system runtime configuration.

In [16], the authors use an architecture-based approach to support dynamic adaptation. Rainbow [17] also updates architectural models to detect inconsistencies and correct certain types of faults. In [18] the authors implement an architecture-based solution in the context of mobile applications to adapt the system by replacing the implementation of components at runtime. None of these solutions considers the impact of environmental changes on the quality requirements of the system.

A different approach to using models at runtime for system adaptation is taken in [19]. The authors update the model based on execution traces of the system. Our approach provides new execution paths for the system by integrating new and modified variants into the model.

The work in [20] provides a solution to a different issue concerning dynamically adaptive systems, which is the control over the wide number of variants that a system with many variability options can have. The authors introduce a solution to maintain dynamically adaptive systems by using aspects to evolve the model.

5 Conclusion

In this paper we outlined our approach to enhance a system's autonomic capabilities by using an automatically evolving model of the system at runtime. The model is periodically updated with re-evaluated QoS values based on runtime information. The evaluations can be used to predict and prevent QoS violations.

We gave an overview of a BPEL-based framework implementing our approach and introduced a system model leveraging service compositions. Variants represented in our flow graph model are implemented as BPEL processes that can be switched at runtime allowing the system to adapt to changes in the environment. In this way, the system can integrate new variants without requiring interruption of the running system.

Acknowledgements. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "SOSOA: Self-Organizing Service-Oriented Architectures" (SNF Sinergia Project No. CRSI22_127386/1).

References

1. Open Grid Forum: WS-Agreement specification, http://www.ogf.org/documents/GFD.107.pdf
2. Marconi, A., Pistore, M., Traverso, P.: Automated composition of web services: the astro approach. IEEE Data Eng. Bull. 31(3), 23–26 (2008)
3. Klusch, M., Fries, B., Sycara, K.P.: Owls-mx: A hybrid semantic web service matchmaker for owl-s services. J. Web Sem. 7(2), 121–133 (2009)
4. Constantinescu, I., Binder, W., Faltings, B.: Service composition with directories. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 163–177. Springer, Heidelberg (2006)
5. Lamsweerde, A.: Reasoning about alternative requirements options. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 380–397. Springer, Heidelberg (2009)
6. Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Chichester (2009)
7. BPEL: BPEL 2.0 standard specification, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
8. Mosincat, A., Binder, W.: Transparent Runtime Adaptability for BPEL Processes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 241–255. Springer, Heidelberg (2008)
9. Mosincat, A., Binder, W.: Automated Performance Maintenance for Service Compositions. In: WSE, pp. 131–140 (2009)
10. Berger, J.: Statistical Decision Theory and Bayesian Analysis. Springer, Berlin (1999)
11. Romano, J.: Testing Statistical Hypotheses. Springer, Berlin (2005)
12. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: ICSE, pp. 111–121 (2009)
13. Meyn, S.P., Tweedie, R.L.: Markov Chains and Stochastic Stability. Springer, London (1993)
14. Raimondi, F., Skene, J., Emmerich, W.: Efficient online monitoring of web-service slas. In: SIGSOFT FSE, pp. 170–180 (2008)
15. Blair, G.S., Bencomo, N., France, R.B.: Models@ run.time. IEEE Computer 42(10), 22–27 (2009)
16. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)
17. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)
18. Floch, J., Hallsteinsen, S.O., Stav, E., Eliassen, F., Lund, K., Gjørven, E.: Using architecture models for runtime adaptability. IEEE Software 23(2), 62–70 (2006)
19. Maoz, S.: Using model-based traces as runtime models. IEEE Computer 42(10), 28–36 (2009)
20. Morin, B., Barais, O., Nain, G., Jéźequel, J.M.: Taming dynamically adaptive systems using models and aspects. In: ICSE, pp. 122–132 (2009)

Visualizing and Assessing a Compositional Approach of Business Process Design

Sebastien Mosser[1], Alexandre Bergel[2], and Mireille Blay–Fornarino[1]

[1] University of Nice Sophia – Antipolis
CNRS, I3S Laboratory, MODALIS team
Sophia Antipolis, France
{mosser,blay}@polytech.unice.fr
[2] Department of Computer Science (DCC)
University of Chile, Santiago, Chile
www.bergel.eu

Abstract. In the context of Services Oriented Architecture (SOA), complex systems are realized through the design of *business–driven processes*. Since the design of a complete process can be complex, composition tools such as *aspects* and *features* propose to define large systems by composing smaller artifacts, easier to understand. But these techniques shift the system complexity into the definition of composition directives able to build it. At composition time, process designers need support to assist them and assess their designed systems. We propose in this article a set of visualizations to represent *compositions* of business–processes and then identify patterns and categorizations. We use the ADORE framework as the underlying process composition platform. We validate this work by visualizing and assessing a Car Crash Crisis Management system (CCCMS, a comparison referential for Aspect Oriented Modeling techniques). We use these visualizations to assess the CCCMS realization.

Note for the proceeding reader: this paper makes use of colors. Although not mandatory for its understanding, an online (colored) version of this paper will ease the reading.

1 Introduction

An application that follows the Service Oriented Architecture paradigm (SOA, [1]) is an assembly of services that realizes business processes. Business processes are defined by business specialists and typically involve many services that are composed in a variety of ways. Furthermore, the need to extend a SOA application with new business features (to follow market trends) arises often in practice. In the technological context of Web Services, business processes can be implemented as *orchestrations of services* [2]. Existing tools and formalisms related to business processes (*e.g.* BPMN notation [3], BPEL industrial language [4]) use a *design–in–the–large* approach and do not intrinsically provide language constructions and frameworks to support the introduction of new features into existing

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 90–105, 2010.
© Springer-Verlag Berlin Heidelberg 2010

processes. Using these technology, one can define *"composition of services"* to reify business–process and then visualize such entities [5].

In this paper, we address the *"composition of process"* problematic, *i.e.*, the composition of existing composition of service to obtain more complex process at the end. New paradigms such as aspects [6] and features [7] model application in terms of composing smaller units. Assuming that a complex system is difficult to understand by humans, they propose to reduce the complexity by defining several smaller artifacts instead of a single and large one. They identify and encapsulate parts of models that are relevant to a particular concern. A same feature may be shared and integrated into several processes simultaneously. These artifacts are then composed to produce the expected system. These approaches help taming the complexity of business processes design [8,9]. As a consequence, the intrinsic complexity of the system is shifted into the composition directives used to build it. When a system involves many processes, it is necessary to have a holistic point of view on features, compositions and business processes to grasp it. In this paper we examine visualization method to tackle complexity of compositions at the application level.

The paper makes the following contributions and innovations:

- a number of visualizations dedicated to support designers when they define business processes using a compositional approach.
- benefits and design weakness are revealed using a number of patterns to assess compositions quality.
- scalability of the approach is sketched by using a very large case study as running example.

We motivate this contribution by presenting in section 2 our running example. Visualizations used to represent and assess compositions are then described in Section 3. Section 4 briefly presents implementation details. We propose a discussion on the approach benefits (associated to interesting perspectives) in Section 5. Finally, Section 6 describes an overview of related work, and Section 7 concludes this paper.

2 Running Example: Realizing a *CCCMS* Using ADORE

This section presents the running example used to validate the visualization approach defended in this paper. It also presents the composition framework we used to realize this example, and highlights our identified needs of visualization.

2.1 Case Study: A Car Crash Crisis Management System

In Kienzle *et al.* [10], authors propose a common case study (a Crisis Management System, CMS) to compare existing *Aspect Oriented Modeling* approaches between each other. We consider this case study as a reference, and use it as a running example to illustrate the problematic tackled in this paper and the contribution we made. According to the definition given by this case study, a CMS

is *"a system that facilitates coordination of activities and information flow between all stakeholders and parties that need to work together to handle a crisis"*. Many types of crisis can be handled by such systems, including terrorist attacks, epidemics, accidents. To illustrate the case study, they provide an instance of a CMS in the context of car accidents. They define this system as the following:

> "*The Car Crash CMS (*CCCMS*) includes all the functionalities of general crisis management systems, and some additional features specific to car crashes such as facilitating the rescuing of victims at the crisis scene and the use of tow trucks to remove damaged vehicles.*"

The requirement document defines ten use cases, described using textual scenario. Each scenario defines first a *main success scenario* which represents the normal flow of actions to handle a crisis (*e.g.*, retrieve witness identity, contact firemen located near to the crash location). Then, a set of *extensions* are described to bypass the normal flow when specific actions occurs (*e.g.*, witness provides fake identification, firemen are not available for a quick intervention).

2.2 Composition Framework: ADORE

The ADORE framework defines a compositional approach to support complex business processes modeling, using the orchestration of services paradigm. Models describing business–driven processes (abbreviated as *orchestrations*, defined as a set of partially ordered activities) are composed with process fragments (defined using the same formalism) to produce a larger process. *Fragments* realize models of small behavior and describe different aspects of a complex business process. ADORE thus allows a business expert to model these concerns separately and then compose them.

Using ADORE, designers can define *composition units* (abbreviated as *composition*) to describe the way fragments should be composed with orchestrations. The merge algorithm used to support the composition mechanism [11] computes the set of actions to be performed on the orchestration to automatically produce the composed process. Implementation details, environment screenshots and video demonstrations are available on the project web site[1].

We proposed in our previous work [12] a realization of the CCCMS system using ADORE. We realized all the use cases main scenarios as orchestration of services, and extensions as fragments to be integrated into these orchestrations. The complete set of designed models (12 orchestrations & 24 fragments, representing 1216 activities scheduled by 895 relations in terms of implementation) is available on the CCCMS realization web page[2].

2.3 Need for Visualization Techniques

When developing a large system, designers handle a large set of initial orchestrations, and a large set of fragments to apply in these orchestrations. ADORE

[1] http://www.adore-design.org

[2] http://www.adore-design.org/doku/examples/cccms/

tackles the complexity of integrating fragments into orchestrations, at the activity level. We depict in Fig. 1 an extract of the Cccms case study (authentication use case, `#2.10`). The composition algorithm identify several unification (represented using dashed lines) between an initial orchestration (which realize the *"main success scenario"*) and a fragment (realizing a business extension). Such a detailed and focused view of composition is not scalable to support the design of large system (non-functional concerns introduction in the Cccms raises the number of unification to more than a thousand).

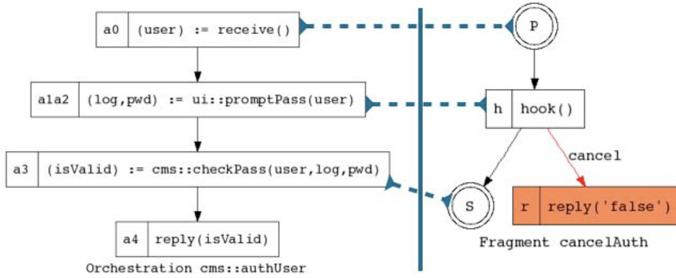


Fig. 1. Detailed visualization of a composition (at the activity level)

Visualization techniques of large data set such as fish–eye [13] can tame the readability problems, but do not reduce the amount of details visualized in this representation. When designing a complete set of business processes using a compositional approach, the objectives of designers are to retrieve a holistic representation of the involved entities to understand easily what they are doing. Such a global visualization is needed at both design and analysis time.

Design Phase. When building a complete system using a compositional approach, designer needs to understand at a coarse–grained level the interactions between the different composition entities they are manipulating. At this step, a designer focuses on the fragments in terms of *impact* (*e.g.*, "the fragment throws a fault") instead of their detailed behavior (*e.g.*, "when a resource takes too much time to reach the crisis location, a timeout fault must be thrown").

Analysis Phase. After the composition directives execution, a *composed* system is obtained. At this step, critical points in the composed system need to be manually identified, for example to design unit tests. This step focuses on the comparison between the *intrinsic complexity* of the original entities and the composed result.

3 Visualizing Compositions Using Mondrian

We describe in this section the polymetrics view techniques, used to define three different visualizations of compositions. These visualizations are then described and applied to the Cccms example.

The common objective of these three visualizations is to provide *dashboards* to designers. Composition dashboards are graphical representations meant to help designers to (*i*) get a scalable and global overview of the compositions present in an orchestration-based application, (*ii*) identity abnormal composition and (*iii*) facilitate the comprehension of a large composition by categorizing compounds. The idea of these dashboards is to enable a better comparison of elements constituting a program structure and behavior.

3.1 Polymetric Views Technique Description

The visualizations we propose are based on *polymetric view* [14]. A *polymetric view* is a lightweight software visualization technique enriched with software metrics information. It has been successfully used to provide “software maps” intended to help software comprehension and visualization. Figure 2 illustrates the principle of polymetric view.

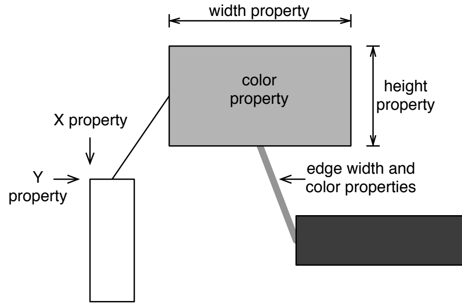


Fig. 2. Principle of polymetric view

Given two-dimensional nodes representing entities, we can map up to 5 metrics on the node characteristics: position properties X and Y, height property, width property and color property:

– *Size*. The width and height of a node can render two measurements. We follow the intuitive notion that the wider and the higher the node, the bigger the measurements its size is telling.
– *Color*. The color interval between white and black may render one measurement. The convention that is usually adopted [15] is that the higher the measurement the darker the node is. Thus light gray represents a smaller metric measurement than dark gray.
– *Position*. The X and Y coordinates of the position of a node may reflect two other measurements.

3.2 Fragments Dashboard *(Design Phase)*

This visualization represents all the *fragments* (as square) involved in a system, focusing on their *impact*. We have identified several impact properties represented as *boxes*: *(i)* hooked variable modification, *(ii)* exception throw, *(iii)* fault handling, *(iv)* initial process execution inhibition and finally *(v)* restricted process inhibition. To illustrate this visualization, we instantiated it on the Cccms example (Fig. 3). It represents the 24 fragments defined to answer to the different use–cases extensions defined in the requirements.

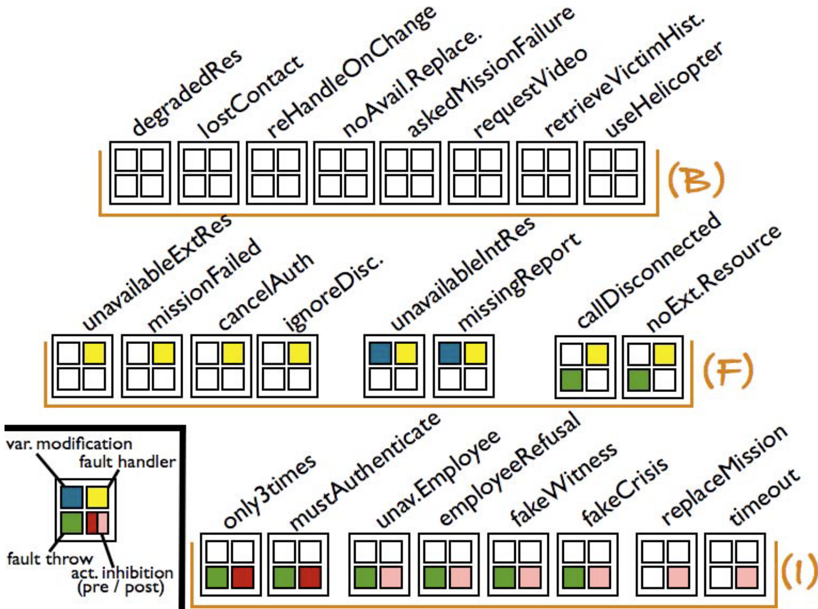


Fig. 3. Fragments dashboard instantiated on the Cccms example

Interpretation. Based on the graphical representation obtained in this view, we have identified 7 different fragment categories, grouped into 3 main families: business extensions ($\mathcal{B}$), fault handling ($\mathcal{F}$) and control–flow inhibition ($\mathcal{I}$).

- Business extensions ($\mathcal{B}$): The *white* fragments only enrich existing process with new additional behavior. They do not modify the initial logic of the business process, and only add new features to enrich it.
- Fault handler ($\mathcal{F}$): *Yellow* boxes represent *fault handling* property. There are several ways to deal with a fault when it occurs in a process: *(i)* doing a re–throw (*green* property) to customize the fault, *(ii)* bypassing the fault (by modifying data to handle the problem, *blue* property) locally and *(iii)* handling the fault by using a business-driven reaction (no other property).

- Control–flow inhibition ($\mathcal{I}$): The *red* property represents the inhibition of an activity and its followers in a business process. The *pink* property is a restriction of the *red* one, since the fragment only inhibits the followers of an activity. Correlated with the fault thrower property (*green* color), we can identify *precondition* (activity inhibition & throw) and *postcondition* (followers inhibition & throw) checker. The *pink–only* fragments represents "dangerous" fragments, which inhibit several activities using a non–standard behavior. A deep understanding of the business–domain is necessary to grasp their behavior properly. The `timeout` fragment is a typical example of this kind of fragments: instead of "simply" throwing an exception when the system detects that a resource takes too much time to reach the crisis location, the Cccms business model asks to stop whatever the system was doing and urgently inform a human coordinator able to decide what to do to counterbalance the situation.

Consequently, this view helps designers to perform a coarse–grained identification of their critical fragments. We have shown this criticality by explaining the *pink–only* fragments in the previous paragraph. Other dangerous color scheme are *green–only* (error throwing in parallel with legacy activities) and *red–only* (control–flow inhibition with a business–driven reaction).

3.3 Composition Dashboard *(Design Phase)*

The previously defined view helps designer to understand the different fragments used in a system. We focus now on the composition visualization defined between fragments and business processes. The composition dashboard visualization represents business processes as rectangles, and fragment using the *4 squares* pattern previously defined. A link between two entities means that they are used together in a composition. The Cccms case study instance of this visualization is depicted in Fig. 4. It represents the 24 fragments and their application on the 11 orchestrations used to realize the use cases.

Interpretation. This visualization helps the designer to understand the depicted system in a holistic way. We identify the following composition categories:

- Orphans ($\mathcal{O}$): Orphans orchestrations are never involved in a composition (*e.g.*, `executeMission` and `execSupObMission`). Their identification helps to detect forgotten composition directives in a complex system. In the Cccms context, they realize very basic use cases which do not define any scenario extensions. As a consequence, there is no fragment associated to these entities.
- Lack of Fault–handlers ($\mathcal{L}_f$): *yellow*–tagged fragments represent *fault handlers*. This visualization allows designers to easily identify a composition which does not involve any fault handler. The lack of fault handling in a process may leads to uncaught fault and jeopardize the behavior of the global system. It can also detect very basic processes which cannot fail.

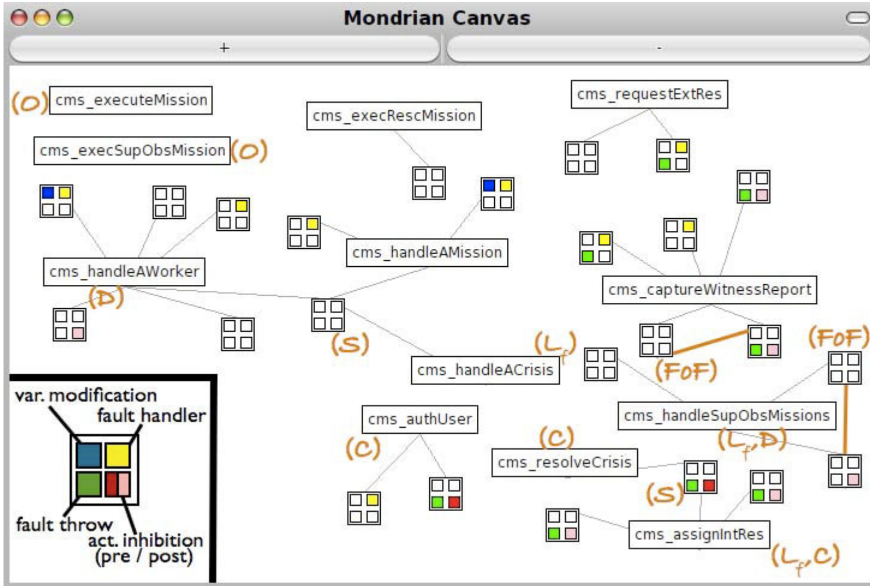


Fig. 4. Composition dashboard instantiated on the CCCMS example

- Conditioned behaviors ($\mathcal{C}$): Several fragments realize precondition (*red* & *green*) and postcondition (*pink* & *green*) checkers. Processes involved in a composition which relies on such checkers must be considered as a good candidate for integration test definition.
- Cross-cutting / Shared concern ($\mathcal{S}$): We can easily identify two shared fragments in this case study. The first one is a precondition checker ("is the user authenticated?"), and the second one is a business driven preoccupation ("re-handle the crisis due to a change of external circumstances"). They clearly represent preoccupations which cross-cuts several scenarios. Their identification helps to identify cross–cutting concerns and can drive system testing and re–engineering.
- Critical / Dangerous behavior ($\mathcal{D}$): Thanks to the *fragments dashboard*, the *pink–only* fragments were identified as *critical* (with other color scheme). Such fragments can inhibit the execution of a process subpart without using usual mechanism (*e.g.* fault throwing) to counter–balance their inhibition. As a consequence, orchestrations enriched using these fragments will require a specific attention from the designer.
- Fragment composed on other fragments ($\mathcal{F}o\mathcal{F}$): We can notice that several fragments are also composed with other fragments. This view lets designer identify such compositions and helps to grasp a semantic link between fragments when discovering an unknown system.

This view lets us identify business–driven processes (using *white–only* fragments) from technical ones. Technical processes deal with precondition and post-condition checking, and do not involve any business driven fragment. On the contrary, high–level business processes rely on business–driven fragments to change the initial control–flow. It produces a dichotomy between *technical* processes (`assignIntRes`, `authUser`, `resolveCrisis`) and *business–driven* processes (the others). Based on this categorization, we can notice two things in the context of the Cccms:

- Categorizing the `resolveCrisis` process as technical seems weird as it corresponds to the main use case of the Cccms. But the scenario realized through this process is clearly technical (*i.e.*, opening a crisis case, asking partners to handle this crisis, closing the case when the crisis is ended) and does not involve any business–driven logic (realized in the other processes, which really *handle* the crisis).
- We can notice the `captureWitnessReport` process, which involves fragments dealing with postcondition checking, fault–handling and business–driven extension in the same composition. As a consequence, this process is both *technical* and *business–driven*. This process is critical in terms of Cccms business–domain: it realizes the retrieval of car crash witness information in a report and drives the trigger of specific missions according to this report. As a consequence, it does not fulfill a single objective and it uses very different fragments to achieve its complete behavior. This process can be then considered as a subsystem in the Cccms context.

Moreover, this visualization supports designers when debugging their compositions. This view identifies in a user–friendly way forgotten composition (through orphans processes) and the lack of fault handler/checker.

3.4 Composition Zoom *(Design Phase)*

When multiple fragments are composed with a process, designers need to restrict their visualization on the system to focus on a given composition. This *zoomed* visualization opens the orchestration box, describing where the fragments are integrated into the initial process. To emphasize the scalability of this visualization, we use the `handleAWorker` process, which is the biggest composition in this case study. The obtained instance is represented in Fig. 5.

Interpretation. This visualization supports a focus on a specific composition and facilitates navigations throughout the orchestration set. Based on this representation, we can easily identify the following concerns in a given composition:

- Isolated activities ($\mathcal{I}$): some activities are not involved in any composition. They represent *technical* activities which do not interfere with the realized scenario. In this particular example, they implement initial message reception, final response sending and other technical activities.

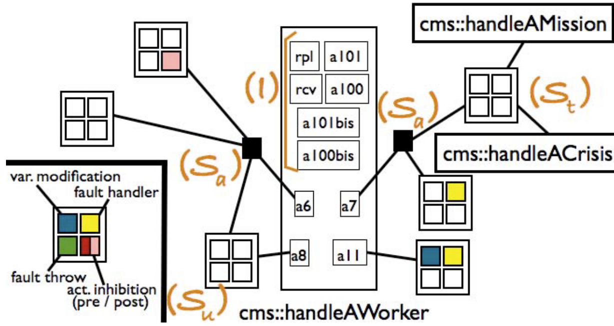


Fig. 5. Composition zoom instantiated on the `handleAWorker` process

- Shared activities ($\mathcal{S}_a$): this visualization lets us clearly identify when several fragments are applied at the same location to a business process (*i.e.* a_6 and a_7 activities). ADORE supports the automatic composition of these fragments into a merged one. However one may need to visualize them simultaneously to understand the merged behavior. In this example, fragments applied to a_6 deal with the non–arrival of a resource at the crisis location. Focusing only on these 3 fragments helps to support the design of the system: are all the non–arrival reasons handled by the process? Are these fragments semantically conflicting?
- Shared Fragments: this view lets the designer identify shared (*i.e.* used several times) fragments for a given composition. Such fragments usually represent cross-cutting concerns. The fact that a fragment is shared with other artifacts gives extra–information when working on its composition:
  - Multiple / Shared targets ($\mathcal{S}_t$): this fragment is related to others processes. This information is a fine–grained version of the "shared concern" one (from *composition dashboard*), linking a process activity to others processes through the use of a common fragment. It can drive the exploration of an unknown system by following such links from a process to another one.
  - Multiple / Shared usage ($\mathcal{S}_u$): this fragment is a factorized enrichment of the initial scenario. As a consequence, it represents a chronic situation which requires special attention when testing and debugging the system. When a fragment is merged several times in the same process, it may introduce redundant activities in the final composition result. This view helps to understand the origin of such redundancy.

3.5 Complexity Dashboard *(Analysis Phase)*

When building a system using a compositional approach, the fragments added into the legacy business process enrich the initial behavior. It is interesting to visualize the differences between the initial and composed processes to identify

composition families. We use several indicators to model business process complexity (inspired by the ones defined by Vanderfesten *et. al*, [16]). In this view, we use the following indicators to represent the system. The complete CCCMS system is represented in Fig. 6.

- *width*: The *width* of a process represents the maximum number of activities concurrently executed in the control–flow.
- *height*: The *height* of a process represents the maximal length of the process.
- *maze*: The *maze* of a process represents the number of different paths available in the process. We map the *maze* indicator to the *color* dimension of the polymetric view.

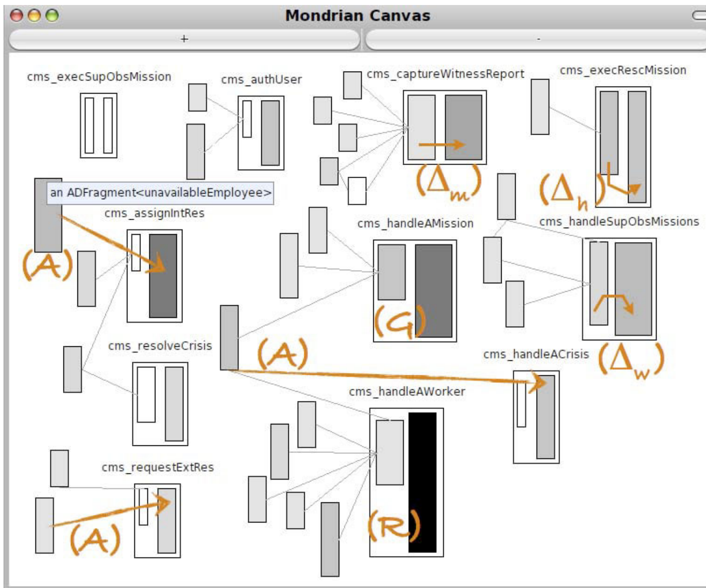


Fig. 6. Complexity dashboard instantiated on the CCCMS example

Interpretation. Using this visualization, designers identify *composition families*, in terms of process complexity evolution (denoted as Δ_i, where i is a business process indicator).

- Δ_w (width expansion): When the composed process is larger than the initial one, it implies that several activities are executed in parallel of the initial behavior. Such an intensive parallelism is *resource-consuming*, and may be deployed on specific high–performance servers. As a consequence, designers are able to identify from this visualization processes requiring a specific attention about performances after composition. In this example, the `handleSupObsMissions` process is a typical case of Δ_w. Fragments used in

this composition introduce several notifications and interactions with other systems (*e.g.,* national crisis management center, internal message bus) done concurrently to the initial scenario.
- Δ_h (height expansion): The augmentation of a process height induces an execution control–flow longer than the initial one. The composition impacts the process execution time and the quality of service is then identified using this visualization. In the `execRescMission` composition, the fragment introduces an interaction with an external system (retrieving victim's medical history before starting the health-care process).
- Δ_m (maze expansion): A dark color identifies a complex process, defining a lot of different paths in its control flow. A contrast change between an initial process and its composed result indicates that the activities defined in this process are more connected between each others. In the Cccms example, the `captureWitnessReport` process is enriched with several small fragments. According to their *impact properties*, these fragments deal with condition checking and fault handling. As a consequence, the composed process contains a lot of new paths (*e.g.* fault bypass) available during the execution of the process. One should pay attention to the possible semantic interactions introduced by such a composition (*e.g* the process handles two faults f and f' but does not define what to do if these two faults happens at the same time).

Based on the previously explained Δ_i expansions, we identify three *critical* situations identifiable in this visualization:

- Global Expansion ($\mathcal{G}$): This phenomenon is identified by an expansion of all the Δ_i indicators. The `handleAMission` process illustrates this global expansion. This strongly indicates that the process designer should particularly focus on this process when designing test, as its apparent initial simplicity hides a very complex process once composed.
- Initial Process Absorption ($\mathcal{A}$): A process is *absorbed* by an extension when the composition output looks like the absorber in terms of complexity. Semantically, it often highlights a requirement granularity problem, where the behavior defined as scenario extensions is more complex than the initial scenario. The `requestExtRes` process illustrates this phenomenon, as it is absorbed by the *degradedRes* fragment. In this case, the system initially requires an external resource, and the extension defines all the actions to be performed when such a resource is not fully available to handle this particular crisis.
- Resonant Composition ($\mathcal{R}$): Resonance is a particular case of the Δ_m expansion. It indicates a lot of interactions between activities in fragments and process, resulting into a labyrinthine process after composition. Designers should handle these processes by taking care of their inherent complexity, focusing on test design and condition checking. The `handleAWorker` process is a typical example of such a resonance.

4 Implementation and Validation

Composition engine implementation. The concrete Adore engine (process representation & composition algorithm) is implemented as a set of logical rules, using the Prolog language. To make Adore interoperable with other tools, we provide an export mechanism, based on Xml. Adore internal representation of orchestrations and business indicators can be exported as Xml documents.

Visualization engine implementation. Our composition dashboards are rendered using Mondrian[3], an agile visualization engine. For the purpose of the experiment, Mondrian operates directly on a metamodel that reifies all the notions introduced in this paper[4].

Validation. The Adore framework was used in five different case studies[5], from a simple *proof of concept* to real-life systems. Business domain handled in these case studies are very diversified (*e.g.*, integer arithmetic, web 2.0 folksonomies, information broadcasting inside academic institution). Visualization techniques presented here were applied with success to these five examples. We present the Cccms example in this paper because of its pertinence to describe the approach (important set of processes leads to a scalability challenge).

5 Discussions and Perspectives

The visualization techniques described in this paper really helps the design of a very large system when using a compositional approach. Considering compositions of processes as first class entities, we provide to designers a framework able to support their design process. Based on these techniques, we identify chronic fragment patterns and sketch a categorization of these entities. Moreover, when analyzing the composed result, we identify critical points where process extensions interact violently with initial behavior. These critical points are easily identifiable using our graphical representation, and may leads to design weaknesses detection. Assuming the fact that the design fits the described requirements, such points can then highlight client requirements weaknesses.

In this paper, we voluntarily focus our visualization work on composition definitions. Our goal is to "understand" a system defined by composition, and support the designer during the design process. We never addressed performances visualization. In an Soa realized using orchestrations of Web Services, partnerships between services and processes is a key point for performance measurement. The invocation of an external partner costs a lot (in terms of data exchanged over the network). Even if Adore proposes a simple visualization of process partners, a Mondrian visualization of the global *choreography* of services [2] helps the designer to easily identify bottlenecks and dangerous patterns in the designed system.

[3] `http://www.moosetechnology.org/tools/mondrian`

[4] `http://www.moosetechnology.org/tools/adore`

[5] Details here: `http://www.adore-design.org/doku/examples/start`

Defining a software using composition techniques often leads to conflicting situations [17,18]. ADORE defines a set of conflict detection rules to identify conflicts in the composed processes. For the same reasons that ADORE raw visualizations do not scale large system representations (too detailed data), designers retrieve from the framework a lot of details concerning the different conflicts detected by the application of detection rules over their models. The *composition zoom* visualization helps designers to identify interaction niche (*e.g.* shared activities), and then tend to reduce the scope of informations handled by the designer. Even if conflict detection mechanisms can be used to automate the detection of conflict, *pragmatic* conflicts[6] will always need an intervention of the designer to be handled properly. Consequently the definition of a MONDRIAN visualization to support conflict resolution will support the global approach and helps designers during the composition process.

6 Related Work

The *Aspect Visualizer* is a visualization-based tool contained in AspectJ Development Tool (AJDT[7]). Source files are represented as rectangles, and lines are colored according to its corresponding advices. This visualization is of a great use when one wants to know which aspects is involved in the definition of a particular class. However, it tells little about the kind of composition mechanism used an essential requirement for understanding aspect-based applications.

Pfeiffer and Gurd address the problematic of aspect visualization [19] (using *treemaps* [20]) to provide a very abstract visualization of aspect oriented programs, based on a hierarchical organization of visualized entities (*e.g.*, package, class). The counter balance of *treemaps* is that it provides little help when relations between elements have their importance, as this is the case in our work. We also propose in this work an intermediate representation, focused on fragment (advice) semantic and independent of any hierarchical relation between entities. Moreover, the use of polymetrics views let us map process metrics to visualized entities.

Fabry *et al.* [21] addressed the issue of visualizing large amount of aspects related hooks (join point shadows) in the base code. AspectMaps enable one to understand a large amount of code bases and permit drilling down to one single join point using a selective structural zooming. AspectMaps differs from composition dashboards essentially on the level of the analysis realized. The AspectMaps tool essentially operates on source code of the base system and its related aspects. Composition dashboards deals essentially with business processes.

All the visualizations presented in this paper are polymetric views. This visualization mechanism has been essentially used to assess software source code [22]. Using polymetric views to assess a software process models has not been considered so far. However, a number of related research efforts are presented below.

---

[6] Which are induced by the business domain (*e.g.*, fault exclusion, overlapped conditions).

[7] http://www.eclipse.org/ajdt

Assessing and visualizing inter software component dependencies and relationships has received a great attention from the software comprehension community. For example, D'Ambros *et al.* [23] visualize how bugs are related to software components. A number of visualizations are providing, ranging from tree maps to complex graph-like structures, on which different layouts are applied.

Byelas and Telea [24] proposed a technique based on "splat texture" that identity *areas* in a software architecture. The idea is to fill a contoured area using this texture. This representation is efficient when dealing with scalability. Their approach is complementary to our, and may be well combined.

7 Conclusions

The composition mechanisms as such defined in the Adore framework help designers to build business process. Visualization techniques can then be used to graphically represent the compositions, and support designers when they assess their systems.

In this paper, we propose three visualizations intended to support designers' understanding of processes compositions. These representations allow designers to easily identify graphical patterns in their composition, and then identify key–points in the resulting composed systems. These key–points are intended to be used in several different ways (*e.g.,* to define unit tests, to catch designers attention on a dangerous situation). We illustrate the scalability of the approach by visualizing the Cccms system. This large system was initially defined to be a comparison referential for Aom techniques. The different visualizations described in this paper are scalable enough to let us represent in an understandable way all the artifacts of this case study.

We focus our work on static visualization of compositions as first–class entities. An interesting perspective of this work is to define new representation dealing with composition conflict detection (*e.g.,* conflict highlighting) or business–process partnerships (*e.g.,* execution bottleneck).

References

1. MacKenzie, M., Laskey, K., McCabe, F., Brown, P., Metz, R.: Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS (February 2006)
2. Peltz, C.: Web Services Orchestration and Choreography. Computer 36(10) (2003)
3. White, S.A.: Business Process Modeling Notation (BPMN). IBM (May 2006)
4. OASIS: WS Business Process Exec. Lang. 2.0. Technical report, OASIS (2007)
5. Pautasso, C., Alonso, G.: Visual composition of web services. In: HCC, pp. 92–99. IEEE Computer Society, Los Alamitos (2003)
6. Douence, R.: A Restricted Definition of AOP. In: European Interactive Workshop on Aspects in Software (EIWAS) (September 2004)
7. Liu, J., Batory, D., Lengauer, C.: Feature Oriented Refactoring of Legacy Applications. In: Int. Conf. on Soft. Engineering (ICSE), Shanghai, China (May 2006)

8. Karastoyanova, D., Leymann, F.: Bpel'n'aspects: Adapting service orchestration logic. In: ICWS 2009: Proceedings of the 2009 IEEE International Conference on Web Services, Washington, DC, USA, pp. 222–229. IEEE Computer Society, Los Alamitos (2009)
9. Charfi, A., Mezini, M.: Ao4bpel: An aspect-oriented extension to bpel. World Wide Web 10(3), 309–344 (2007)
10. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems. In: A Case Study for Aspect-Oriented Modeling. Requirements document for TAOSD special issue, McGuill University & University of Luxembourg (September 2009)
11. Mosser, S., Blay-Fornarino, M., Riveill, M.: Web Services Orchestration Evolution: A Merge Process For Behavioral Evolution. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 35–49. Springer, Heidelberg (2008)
12. Mosser, S., Blay-Fornarino, M., France, R.: Workflow Design using Fragment Composition Crisis Management System Design through ADORE. Transactions on Aspect-Oriented Software Development (TAOSD), 1–34 (2010) (submitted)
13. Sarkar, M., Brown, M.H.: Graphical Fisheye Views of Graphs. In: CHI 1992: Proc. of the SIGCHI Conf. on Human Factors in Computing Sys., pp. 83–91. ACM, New York (1992)
14. Lanza, M., Ducasse, S.: Polymetric views—a lightweight visual approach to reverse engineering. Trans. on Soft. Engineering (TSE) 29(9), 782–795 (2003)
15. Gîrba, T., Lanza, M.: Visualizing and characterizing the evolution of class hierarchies. In: WOOR 2004 (5th ECOOP W. kshp. on OO Reengineering) (2004)
16. Vanderfesten, I., Cardoso, J., Mendling, J., Reijers, H.A., Van Der Aalst, W.M.: Quality Metrics for Business Process Models. BPM and Workflow Handbook, 179–190 (2007)
17. Barais, O., Lawall, J., Le Meur, A.F., Duchien, L.: Safe Integration of New Concerns in a Software Architecture. In: 13th Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS 2006), Potsdam, Germany, March 2006. IEEE, Los Alamitos (2006)
18. Szyperski, C.: Independently Extensible Systems – Software Engineering Potential and Challenges. In: Proceedings of the 19th Australian Computer Science Conference, Melbourne, Australia (1996)
19. Pfeiffer, J.H., Gurd, J.R.: Visualisation-based tool support for the development of aspect-oriented programs. In: AOSD 2006: Proceedings of the 5th International Conference on Aspect-Oriented Software Development, pp. 146–157. ACM, New York (2006)
20. Balzer, M., Deussen, O., Lewerentz, C.: Voronoi treemaps for the visualization of software metrics. In: SoftVis 2005: Proceedings of the 2005 ACM Symposium on Software Visualization, pp. 165–172. ACM Press, New York (2005)
21. Fabry, J., Kellens, A., Ducasse, S.: Aspectmaps: A scalable visualization of join point shadows. Tr/dcc-2010-2, University of Chile (April 2010)
22. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer, Heidelberg (2006)
23. D'Ambros, M., Lanza, M.: Visual software evolution reconstruction. J. Softw. Maint. Evol. 21(3), 217–232 (2009)
24. Byelas, H., Telea, A.C.: Visualization of areas of interest in software architecture diagrams. In: SoftVis 2006: Proceedings of the 2006 ACM Symposium on Software Visualization, pp. 105–114. ACM, New York (2006)

Construction of Asynchronous Communicating Systems: Weak Termination Guaranteed!

Kees M. van Hee, Natalia Sidorova, and Jan Martijn van der Werf

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{k.m.v.hee,n.sidorova,j.m.e.m.v.d.werf}@tue.nl

Abstract. Correctness of asynchronously communicating systems (ACS) is known to be a hard problem, which became even more actual after the introduction of Service Oriented Architectures and Service Oriented Computing. In this paper, we focus on one particular correctness property, namely weak termination: at any moment of the system execution, at least one option to terminate should be available. We present a compositional method for constructing an ACS that guarantees weak termination. The method allows for refinement of single components, refinement of compositions of components and the creation of new components in the system. For two important classes of ACS, weak termination follows directly from their structure. These classes focus on the concurrency over components and on the implementation of protocols and communicating choices.

1 Introduction

Verification of asynchronously communicating systems is known to be a hard problem. In past years, modeling and verification mainly focussed on business processes. With the introduction of paradigms like Service Oriented Architectures (SOA) [1,4,7], the focus shifts more and more to the modeling and verification of inter organizational processes. Different organizations form a virtual organization to deliver a certain service to other organizations. Languages like the Business Process Execution Language (BPEL) [5] and the current draft of the Business Process Modeling Notation 2.0 (BPMN2) [13] are introduced to model the interaction between processes.

In the paradigm of SOA, components deliver *services* to other components. Communication between components is via message sending, and therefore asynchronous. One of the main aspects of SOA is *dynamic coupling* of components: to perform a service, a component may need services of other components, which might be chosen during runtime. This way, during runtime a tree of service instances is formed. We call this a *service tree*. Due to the dynamic coupling, but also for privacy reasons, components only know their direct neighbors. Hence, the whole service tree is not known to any component in the tree, a component only knows to whom it is connected to. This dynamic nature makes the

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 106–121, 2010.
© Springer-Verlag Berlin Heidelberg 2010

verification of a service tree very hard, or even infeasible. In this paper we focus on one correctness property, called weak termination, that can be checked compositionally, by checking pairwise compositions of components.

Weak termination means that at any moment of the system execution, at least one option to terminate exists. Note that weak termination does not require that the system always eventually terminates, we only guarantee that the option to terminate always remains open. Weak termination guarantees that a system cannot deadlock nor can it be trapped in an infinite loop: in each infinite loop there is always an option to exit the loop. As the communication between components is asynchronous, Petri nets [14] are a natural choice for modeling the components and their interactions. In [2], the authors provide a method to verify weak termination of service trees compositionally. Each component should be weakly terminating, and each connected pair of components should satisfy a certain condition. This way, a given service tree can be checked by only verifying each component and each connected pair of components.

In this paper, we show a different approach. Instead of checking an existing service tree, we present a *construction methodology* for a class of service trees which are always weakly terminating. The methodology consists of three rules. The first rule allows for the *refinement* of existing components by refining a single place by a new component. The second rule enables the enrichment of the interaction between components by the *replacement* of pairs of places by coupled components that have the weak termination property. The last rule allows for the *creation* of new components as an offspring of an existing component. On top of this rules, we may apply classical refinement rules, e.g., as defined by Murata [12], Berthelot [8], and the workflow refinement rule [10], to refine the internal structure of the component.

For two base classes of coupled components, weak termination can be decided based on their structure. Both classes occur frequently in the design of components, and are based on two subclasses of Petri nets: marked graphs and state machines. The first class focusses on concurrency over components, whereas the latter focusses on communicating choices over components and the design of interaction protocols.

This paper is structured as follows. In Section 2, we introduce some basic concepts and notations. The component framework is explained in Section 3. We present the construction methodology in Section 4, and in Section 5 we introduce the two base classes of coupled components that are weakly terminating by their structure. Finally, we conclude our paper in Section 6.

2 Preliminaries

Let S be a set. The powerset of S is defined as $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$. With $|S|$ we denote the number of elements in S. The empty set, i.e., the set without any elements is denoted by $\emptyset$. Two sets S and T are disjoint if $S \cap T = \emptyset$. A partition $P \subseteq \mathcal{P}(S)$ is a set such that $\bigcap_{A \in P} A = S$ and $A \cap A' \neq \emptyset \implies A = A'$ for all $A, A' \in P$. We denote the set of all natural numbers as $\mathbb{N} = \{0, 1, 2, \ldots\}$.

A *sequence* σ of length $n \in I\!N$ over S is a function $\sigma : \{1, \ldots, n\} \rightarrow S$. We denote the length of a sequence by $|\sigma| = n$. We denote a sequence of length n by $\sigma = \langle a_1, \ldots, a_n \rangle$ for some $a_1, \ldots, a_n \in S$. If $|\sigma| = 0$, it is the *empty sequence* ϵ. The set of all finite sequences over S is denoted by S^*. *Concatenation* of two firing sequences $\sigma, \upsilon \in S^*$ is a function $\sigma; \upsilon : \{1, \ldots, |\sigma| + |\upsilon|\}$ defined by $(\sigma; \upsilon)(i) = \sigma(i)$ for $1 \leq i \leq |\sigma|$ and $(\sigma; \upsilon)(i) = \upsilon(i - |\sigma|)$ for $|\sigma| < i \leq |\sigma| + |\upsilon|$. The Parikh vector of a sequence σ, denoted by $\overrightarrow{\sigma}$, is a bag representing the number of occurrences of each element in σ. A *bag* m (*multiset*) over S is a function $m : S \rightarrow I\!N$. For $s \in S$, $m(s)$ denotes the number of occurrences of s in m. We denote a bag by square brackets. E.g., in a bag $[a, b^2, c]$, element a occurs once, element b twice, and element c once. All other elements have a multiplicity of 0. We write $I\!N^S$ for the set of all bags over S. The empty bag, i.e., for all elements the multiplicity is 0, is denoted by $\emptyset$. We use $+$ and $-$ for the sum and difference of two bags, and $=, <, >, \leq, \geq$ for the comparison of two bags, which are defined in a standard way. Sets can be seen as a special kind of bag were all elements occur only once.

A Petri net N is a tuple (P, T, F) where P is the set of *places*, T is the set of *transitions*, P and T are disjoint, and $F \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs*. An element of $P \cup T$ is called a *node*. Graphically, we denote places by circles, transitions by squares, and arcs as arrows between places and transitions. The preset ${}^\bullet n$ of a node $n \in P \cup T$ is defined as ${}^\bullet n = \{n' \in P \cup T \mid (n', n) \in F\}$. Its postset $n^\bullet$ is defined as $n^\bullet = \{n' \in P \cup T \mid (n, n') \in F\}$. The *state* of a Petri net, called a *marking* is a bag over the places P of N. A marking is graphically represented by placing *tokens* in each place. A marked Petri net is a pair (N, m_0), where N is a Petri net and m_0 a marking of N. A transition $t \in T$ is *enabled* in (N, m_0), denoted by $(N : m_0 \xrightarrow{t})$ if ${}^\bullet t \leq m_0$. An enabled transition in (N, m_0) can *fire* resulting in a new marking $m' = m_0 - {}^\bullet t + t^\bullet$, denoted by $(N : m_0 \xrightarrow{t} m')$. We lift the notation of transition firing and enabledness to sequences in a standard way. A sequence $\sigma \in T^*$ of length $n \in I\!N$ is a *firing sequence* of (N, m_0) if there exist markings $m_{i-1}, m_i \in I\!N^P$ such that $(N : m_{i-1} \xrightarrow{\sigma(i)} m_i)$ for all $1 \leq i \leq n$, and is denoted by $(N : m_0 \xrightarrow{\sigma} m_n)$. The set of all reachable markings of (N, m_0) is defined as $\mathcal{R}(N, m_0) = \{m \mid \exists \sigma \in T^* : (N : m_0 \xrightarrow{\sigma} m)\}$. The set of all possible firing sequences from m_0 to m is denoted by $\mathcal{L}(N, m_0, m) = \{\sigma \in T^* \mid (N : m_0 \xrightarrow{\sigma} m)\}$. A place is *k-bounded* in (N, m_0) for some $k \in I\!N$ if $m(p) \leq k$ for all $m \in \mathcal{R}(N, m_0)$. A marked Petri net is *k-bounded* if all places are k-bounded. A place or marked Petri net is *safe* if it is 1-bounded. A marking m of N is a *deadlock* if there are no transitions enabled in (N, m). It is a *home marking* of (N, m_0) if $m \in \mathcal{R}(N, m')$ for all $m' \in \mathcal{R}(N, m_0)$.

Two Petri nets $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ are *isomorphic* with respect to some function $\rho : P_1 \cup T_1 \rightarrow P_2 \cup T_2$ if ρ is bijective, $\rho(p) \in P_2$ for all $p \in P_1$, $\rho(t) \in T_2$ for all $t \in T_1$ and $(p, t)\imath F_1$ if and only if $(\rho(p), \rho(t)) \in F_2$.

If for a Petri net $N = (P, T, F)$ we have $|{}^\bullet t| \leq 1$ and $|t^\bullet| \leq 1$ for all $t \in T$, the Petri net is an *S-net*, also called a *state machine*. If $|{}^\bullet p| \leq 1$ and $|p^\bullet| \leq 1$ for all

$p \in P$, it is a *T-net*, also called a *marked graph*. If a net is a state machine, we graphically omit the transitions between places.

A special class of Petri nets are *workflow nets*. A workflow net is a Petri net $N = (P, T, F)$ such that there exist exactly one place $i \in P$ with $^\bullet i = \emptyset$, called the *initial place*, one place $f \in P$ with $f^\bullet = \emptyset$, called the *final place*, and all nodes $n \in P \cup T$ are on a path from i to f. A workflow net is *sound* if $[f]$ is a home marking of $(N, [i])$ and for all transitions $t \in T : \exists m \in \mathcal{R}(N, [i]) : {}^\bullet t \leq m$. A workflow net N is *generalized sound* if $[f^k]$ is a home marking of $(N, [i^k])$ for all $k \in \mathbb{N}$.

3 Asynchronous Communicating Systems

A system consists of components that communicate asynchronously with each other via interfaces, and to each interface at most one component is connected. In this approach, we model a component by a Petri net [14]. As communication is asynchronous, we model the communication via special places, called *interface places* (cf. [11]). An interface place is either an *input place*, i.e., the component receives a message via this place, or an *output place*, i.e., the component sends a message via this place. As a component needs to communicate with other components, the interface places are partitioned into *ports*. Transitions have a sign with respect to a port. For each port, a transition either sends messages (sign !), it receives messages (sign ?), or it is silent (sign τ). Consider the component shown in Figure 1. In this example, component N has three ports, G, H, and J. Port G consists of two input places b and e, and three output places a, c and d. Transition t has sign ! with respect to port G and sign τ with respect to port H.

The internal places together with the transitions form the inner structure of the component, which we call the *skeleton*. The input and output places determine the interfaces to the exterior. A component has one initial and one final marking, in which only internal places are marked, i.e., in the initial and final markings no interface places can be marked. The final marking does not need to be a deadlock.

Definition 1 (Component, skeleton, sign). *A* component *is an 8-tuple* $(P, I, O, T, F, \mathcal{G}, i, f)$ *where* $((P \cup I \cup O, T, F), i)$ *is a marked Petri net; P is a set of* internal places*; I is a set of* input places*, and* $^\bullet I = \emptyset$*; O is a set of* output places*, and* $O^\bullet = \emptyset$*; P, I, O are pairwise disjoint;* $\mathcal{G} \subseteq \mathcal{P}(I \cup O)$ *is a partition of the interface places, called the* ports*. A transition either sends to or receives from a port, i.e.,* $^\bullet G \cap G^\bullet = \emptyset$ *for all* $G \in \mathcal{G}$*.* $i \in \mathbb{N}^P$ *is the* initial marking*; and* $f \in \mathbb{N}^P$ *is the* final marking*. We call the set* $I \cup O$ *the* interface places *of the component. Two components N and M are called* disjoint *if* P_N*,* P_M*,* I_N*,* I_M*,* O_N*,* O_M*,* T_N *and* T_M *are pairwise disjoint.*

The skeleton *of an OPN N is defined as the Petri net* $\mathcal{S}(N) = (P_N, T_N, F)$ *with* $F = F_N \cap ((P_N \times T_N) \cup (T_N \times P_N))$*. The* sign *of a transition with respect to a port* $G \in \mathcal{G}$ *is a function* $\lambda_G : T \to \{!, ?, \tau\}$ *defined by* $\lambda_G(t) =!$ *if* $t^\bullet \cap G \neq \emptyset$*,* $\lambda_G(t) =?$ *if* $^\bullet t \cap G \neq \emptyset$*, and* $\lambda_G(t) = \tau$ *otherwise, for all* $t \in T_N$*.*

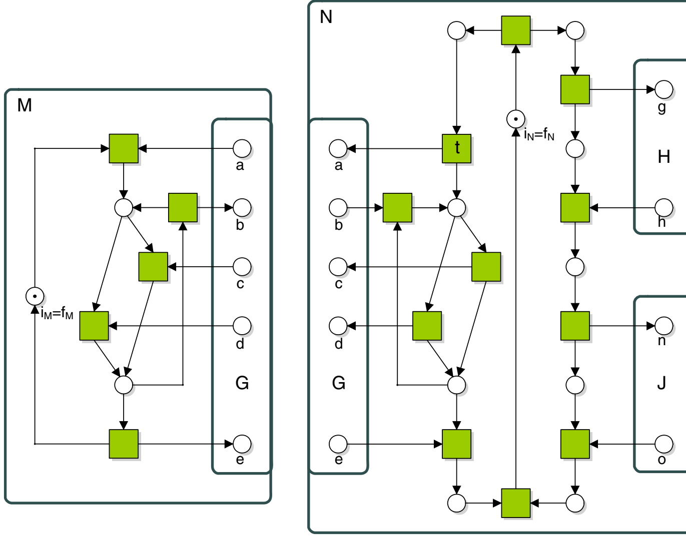


Fig. 1. A component with three ports G, H and J

Even if a component is not connected to any other component, it should work correctly, i.e., it should always be possible to reach the final marking of the component. This property is called *weak termination*.

Definition 2 (Weak termination of a component). *A component N is* weakly terminating *if f_N is a home marking of $(\mathcal{S}(N), i_N)$.*

We do not require the final marking to be a deadlock. Instead, the final marking can be a state from which it is always possible to return to, called an *idle state* in which the component is in rest. For example, if the initial marking and final marking are identical, this is the case.

Components communicate via their ports. Communication is only possible if the input places of the one component are the output places of the other component and vice versa. In our approach we compose components by fusing interface places with the same name.

Definition 3 (Composition). *The components A and B are* composable *if there exists a port $G \in \mathcal{G}_A \cap \mathcal{G}_B$ such that $(I_A \cap O_B) \cup (O_B \cap I_B) = G$ and $P_A \cap P_B = T_A \cap T_B = I_A \cap I_B = O_A \cap O_B = \emptyset$. If A and B are composable, their* composition *is a component $A \oplus_G B = (P, I, O, T, F, \mathcal{G}, i, f)$ defined by: $P = P_A \cup P_B \cup G$, $T = T_A \cup T_B$, $F = F_A \cup F_B$; $I = (I_A \setminus G) \cup (I_B \setminus G)$; $O = (O_A \setminus G) \cup (O_B \setminus G)$; $\mathcal{G} = (\mathcal{G}_A \cup \mathcal{G}_B) \setminus \{G\}$; $i = i_A + i_B$; and $f = f_A + f_B$. If there exists a unique port $G \in \mathcal{G}_A \cap \mathcal{G}_B$ such that $A \oplus_G B$, we write $A \oplus B$.*

Consider again the example of Figure 1. Components N and M share port G, and all input places of port G in N are output places of port G in M and vice

versa. Hence, we can compose the two components via port G. In the resulting net, the interface places of port G become internal places of the composition $N \oplus_G M$.

4 Construction Rules

To guarantee weak termination of a system consisting of asynchronous communicating components is in general very hard due to high degree of concurrency of the components. In [2], a sufficient condition has been presented to pairwise verify weak termination for a tree-structured composition of components. In this section we present an approach that guarantees this condition by construction. The approach consists of three rules to refine a system. The first rule is refinement within a single component. In the second rule, two so called "synchronized places" can be refined by a composition of two components. The last rule involves the creation of a new coupled component in a system.

4.1 Refinement within Components

For Petri nets there already exist many refinement rules, like the rules of Murata [12] and Berthelot [8]. These rules guarantee weak termination: applying them on a weakly terminating component results again in a weakly terminating component. However, these rules are all applied on internal parts of the component and do not extend the ports of a component. In [10], the authors show that a place in a workflow net may be refined by a generalized sound workflow net, while preserving the soundness condition. We redefine this refinement operation on components and refine an internal safe place p of a component N by a workflow component M. A workflow component is a component which has a workflow net as skeleton, the only marked place in the initial marking is the initial place of the skeleton, and the only marked place in the final marking is the final place of the skeleton.

Definition 4 (Workflow component). *A component N is a* workflow component *if $\mathcal{S}(N)$ is a workflow net with initial place i and final place f, and $i_N = [i]$ and $f_N = [f]$.*

When we refine a place p in a component N by a new workflow component M, all transitions in the preset of p are connected to the initial place of M, and all transitions in the postset of p are connected to the final place of M. Place p is then removed from N. The ports of M are added to the ports of N. Consider the example of Figure 2. In this example, place p of component N is refined by component M. In the refined net N', the port G of M is added to the already existing ports of N.

Definition 5 (Place refinement). *Let N be a component and M be a workflow component, such that N and M are disjoint. Let $p \in P_N$ be a place that is safe in $(\mathcal{S}(N), i_N)$. The refined component $N \odot_p M = (P, I, O, T, \mathcal{G}, F, i, f)$ is defined as:*

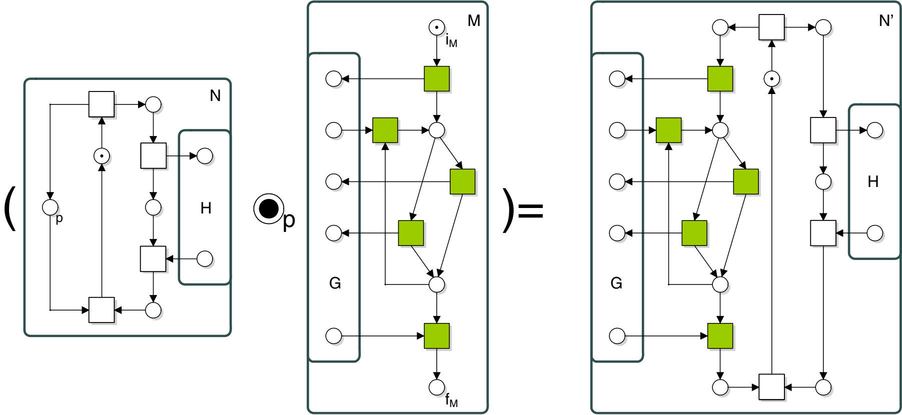


Fig. 2. Refinement of place p in N by component M

$P = (P_N \setminus \{p\}) \cup P_M$; $I = I_N \cup I_M$; $O = O_N \cup O_M$; $T = T_N \cup T_M$; $\mathcal{G} = \mathcal{G}_N \cup \mathcal{G}_M$ $F = F_N \setminus ((^\bullet p \times \{p\}) \cup (\{p\} \times p^\bullet)) \cup F_M \cup (^\bullet p \times \{i_M\}) \cup (\{f_M\} \times p^\bullet)$; $i(s) = i_N(s)$ *and* $f(s) = f_N(s)$ *for all* $s \in P_A \setminus \{p\}$, $i(i_M) = i_N(p)$ *and* $f(f_M) = f_N(p)$.

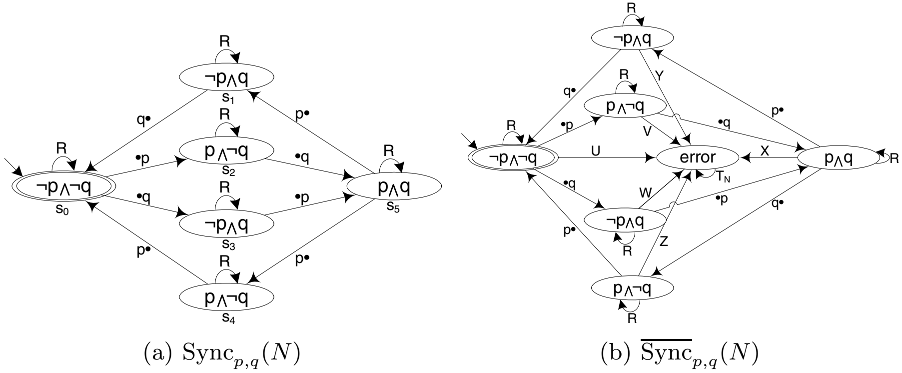
This definition of place refinement propagates the ports of the refining component to the original component. At a first glance, this definition seems to contradict the paradigm of information hiding. However, the definition allows for the refinement of a component by a composed component, as long as this composition remains a workflow component. This way, the ports remain invisible to the environment of the original component.

Since we only allow to refine safe places, we do not need to require the component with which the place will be refined to be generalized sound as in [10], but weak termination as defined in the previous section is sufficient.

Theorem 6 (Refinement of safe places preserves weak termination). *Let* N *be a component and let* M *be a weakly terminating workflow component. Let* $p \in P_N$ *be a safe place in* $(\mathcal{S}(N), i_N)$. *If both* N *and* M *are weakly terminating, then* $N \odot_p M$ *is weakly terminating.*

4.2 Refinement over Components

In system construction, component interaction is often established in several cycles. In each cycle the interaction is refined, until the desired communication protocol is designed. The first rule, i.e., refinement of a single place in a component, does not suffice, as it creates a new port. Thus, it cannot create a more elaborate interaction protocol between components, it can only create new interactions. Moreover, we want to refine the scheme of interacting components. For example, a simple request-response pattern could be refined in a more elaborated negotiation pattern. Therefore, we introduce refinement of special pairs



(a) $\text{Sync}_{p,q}(N)$ (b) $\overline{\text{Sync}}_{p,q}(N)$

Fig. 3. Desired behavior for synchronized places p and q in component N

of places in a composition. Refinement of a single place in a weakly terminating component by a weakly terminating subcomponent results again in a weakly terminating component. Refinement of two places by a composition of two components is in general not weakly terminating. As we refine a pair of places by a composition, we need to be sure that there exists markings in which both places are marked. If such a marking cannot be reached, the interaction we refine the components with cannot be executed properly, and thus the refined composition is not weakly terminating. An intuitive approach would be to apply this refinement only to "synchronizable" places, i.e., two places such that whenever one is marked before the other, it is always possible to keep the place marked until the other is marked as well.

Consider two places p and q that satisfy this intuitive requirement, i.e, if p becomes marked, then it is always possible to keep it marked until q is marked, or vice versa. To describe the desired behavior on places p and q, we use a state machine as shown in Figure 3(a). The initial marking s_0 is also the final marking. In the annotation of places, p means that p contains a token, and $\neg p$ means that p does not contain tokens (similarly for q). The arcs are annotated by sets of transitions: for each element in the set on an arc from s to s' there is a transition with preset $\{s\}$ and postset $\{s'\}$. The set R is defined as $R = T_N \setminus ({}^\bullet p \cup p^\bullet \cup {}^\bullet q \cup q^\bullet)$. Initially, both places p and q are unmarked. If a transition in the preset of p or q fires, we reach a state in which either p or q is marked. Then, if p is marked only transitions in R or ${}^\bullet q$ are able to fire, or, if q is marked, only transition in R or ${}^\bullet p$ are able to fire. If such a transition fires, we reach a marking in which both places p and q are marked. From this state, first both places need to become unmarked, before they can become marked again.

If any accepting firing sequence, i.e., a firing sequence from initial marking to the final marking, in the skeleton of the component is also an accepting firing sequence of $\text{Sync}_{p,q}(N)$, a pair of places is synchronized. Consider the example of Figure 4. In this example, there exist accepting firing sequences of $A \oplus_G B$ that are also accepting firing sequences of $\text{Sync}_{p,q}(A \oplus_G B)$. Consider the accepting

firing sequence $\langle t, u, w, v, x, y, z\rangle$. Then this is not an accepting firing sequence of $\text{Sync}_{p,q}$. However, since transition v does not depend on the input of transition w, we can *swap* these transitions. This way, we can *shuffle* the firing sequence σ to the new firing sequence $\langle t, u, v, w, x, y, z\rangle$. This firing sequence is an accepting firing sequence for both $A \oplus_G B$ and $\text{Sync}_{p,q}$. Hence, if for any accepting firing sequence of $A \oplus_G B$ there exists such a shuffled firing sequence that is an accepting firing sequence of both $A \oplus_G B$ and $\text{Sync}_{p,q}$, places p and q are synchronized. If there would exist an accepting firing sequence that cannot be reshuffled into an accepting firing sequence of both, the interaction we refine with cannot be initiated properly, and thus the refined composition is not weakly terminating anymore.

Definition 7 (Synchronized places). *Let A and B be two component composable with respect to some port $G \in \mathcal{G}_A \cap \mathcal{G}_B$. Two places $p \in P_A$ and $q \in P_B$ are* synchronized*, denoted by $p \rightleftharpoons_N q$ if and only if*

$$\forall \sigma \in \mathcal{L}(N, i, f) : \sigma \in \mathcal{L}(Sync_{p,q}(N))$$

A direct consequence of the definition of $p \rightleftharpoons_N q$ is that the synchronized places p and q are safe in the skeleton of N.

Lemma 8. *Let N be a component, and $p, q \in P_N$ such that $p \rightleftharpoons_N q$. Then p and q are safe in $(\mathcal{S}(N), i_N)$.*

Checking whether two places are synchronized is decidable. The state machine $\overline{\text{Sync}}_{p,q}(N)$ extends $\text{Sync}_{p,q}(N)$ by adding an extra state annotated with *error*. Let $U = p^\bullet \cup q^\bullet$, $V = U \cup {}^\bullet p$, $W = U \cup {}^\bullet q$, $X = {}^\bullet q \cup {}^\bullet p$, $Y = X \cup p^\bullet$, and $Z = X \cup q^\bullet$. Connect the states as shown in Figure 3(b). Then, state *error* is a live lock, which can only be reached if places p and q are not synchronizable. Hence, in the synchronous product of component N and $\overline{\text{Sync}}_{p,q}(N)$, no marking should be reachable in which place *error* is marked, which is a classical coverability problem that is decidable for Petri nets.

Given a composition $N = A \oplus_G B$, the refinement of two synchronized places by a composition of two workflow components $C \oplus_H D$ results in a new composition, where in component A place p is refined by C, and in component B place q is refined by D. The interface places of ports G and H become internal places.

Definition 9 (Refinement of synchronized places). *Let A and B be two components that are composable with respect to port $G \in \mathcal{G}_A \cap \mathcal{G}_B$ and let C and D be two workflow components that are composable with respect to port $H \in \mathcal{G}_C \cap \mathcal{G}_D$ and $A \oplus_G B$ and $C \oplus_H D$ are disjoint. Let $p \in P_A$ and $q \in P_B$ such that $p \rightleftharpoons_{A \oplus_G B} q$. The refined component $(A \oplus_G B)_p \odot_q (C \oplus_H D) = (P, I, O, T, F, \mathcal{G}, i, f)$ is defined by: $P = (P_{A \oplus_G B} \cup P_{C \oplus_H D}) \setminus \{p, q\}$; $T = T_{A \oplus_G B} \cup T_{C \oplus_H D}$; $I = I_{A \oplus_G B} \cup I_{C \oplus_H D}$; $O = O_{A \oplus_G B} \cup O_{C \oplus_H D}$; $F = (P_{A \oplus_G B} \cup P_{C \oplus_H D} \setminus (({}^\bullet p \times \{p\}) \cup (\{p\} \times p^\bullet) \cup ({}^\bullet q \times \{q\}) \cup (\{q\} \times q^\bullet))) \cup ({}^\bullet p \times \{i_C\}) \cup (\{f_C\} \times p^\bullet) \cup ({}^\bullet q \times \{i_D\}) \cup (\{f_D\} \times p^\bullet)$; $i = i_{A \oplus_G B}$; and $f = f_{A \oplus_G B}$.*

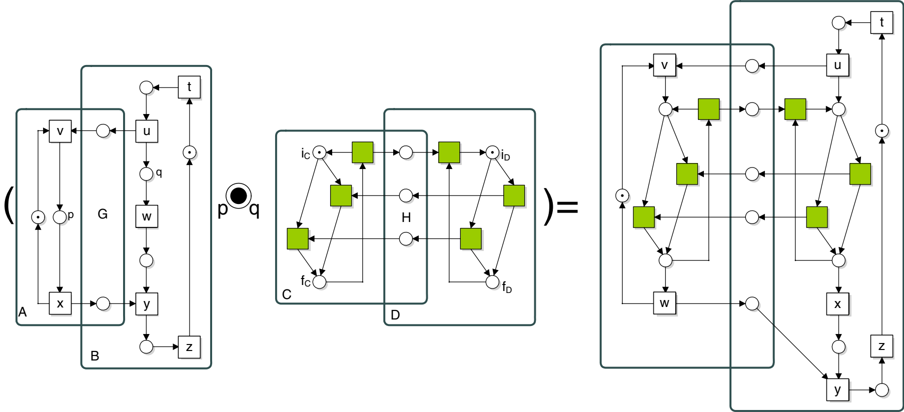


Fig. 4. The refinement $(A \oplus_G B)_p {\odot}_q (C \oplus_H D)$

Consider the example of Figure 4. In this example, place p of component A and place q of component B are synchronized in $A \oplus_G B$. Components C and D are workflow components. Both compositions $A \oplus_G B$ and $C \oplus_H D$ are weakly terminating. The refined component $(A \oplus_G B)_p \odot_q (C \oplus_H D)$ is also weakly terminating.

Theorem 10 (Weak termination for refinement of synchronized places). *Let A and B be two components that are composable with respect to port $G \in \mathcal{G}_A \cap \mathcal{G}_B$ and let C and D be two workflow components that are composable with respect to port $H \in \mathcal{G}_C \cap \mathcal{G}_D$ and $A \oplus_G B$ and $C \oplus_H D$ are disjoint. Let $p \in P_A$ and $q \in P_B$ such that $p \rightleftharpoons_{A \oplus_G B} q$. If $A \oplus_G B$ and $C \oplus_H D$ are weakly terminating, then the refined component $(A \oplus_G B)_p \odot_q (C \oplus_H D)$ is also weakly terminating.*

4.3 Creating New Components

The first two rules only allow to extend existing components. With the third rule, it is possible to connect new components in a system such that the system remains weakly terminating. The rule is based on the principle of outsourcing. Consider Figure 5. For example, if place p has the meaning that when a token resides in it, "an item is produced", and the decision is taken to outsource the production activity, we can add two transitions: a "start producing item" and a "finish producing item". Then the start transition initiates the component producing the item, and the finish transition fires if the item is produced. Creating a new port for these transitions allows the connection of a new component to the existing system. To realize the intended refinement, a place p in a component N is refined by the component M_1 – a component that sends a message over a new port G and then waits in place x until it receives a message on port G.

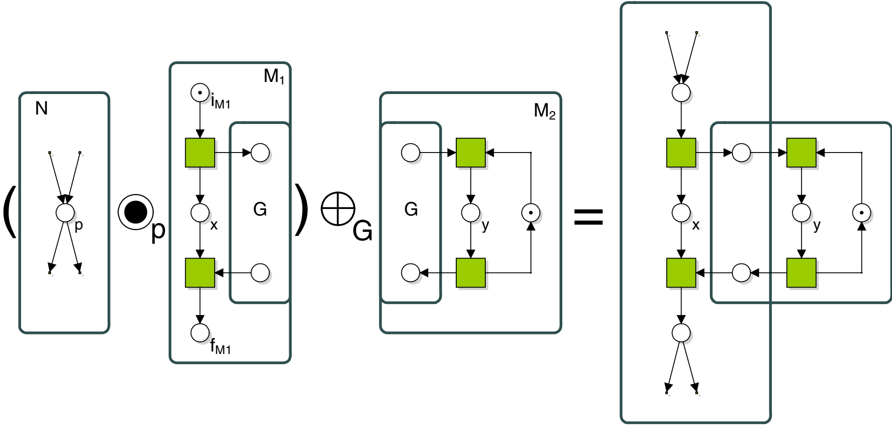


Fig. 5. Extending the composition with a new coupled component

Component M_1 is composed to a component M_2 that waits in an idle state until it receives a message on port G, thus marking place y. M_2 then sends a message, and returns to its idle state. In BPEL, this corresponds to an invoke activity, where a message is sent to execute an operation, and the activity waits for the result of the operation. There are many variants of this rule possible.

If in a weakly terminating component the place we extend is safe, the newly created system is again weakly terminating. Furthermore, the newly added places x and y are synchronized by construction. Hence, we can apply the second rule of the approach on these places to refine the interaction between $N \odot_p M_1$ and M_2.

Theorem 11. *Let N be a component, $M_1 \oplus M_2$ the request-response net as depicted in Figure 5, such that N and M_1 are disjoint and N and M_2 are disjoint. Define $N' = (N \odot_p M_1) \oplus M_2$. Then N' is weakly terminating and $x \rightleftharpoons_{N'} y$.*

5 Basic Classes of Weakly Terminating Compositions

In the previous section, we presented a construction approach to build systems that are weakly terminating by construction. However, for the second rule, i.e., to refine two synchronized places, we need a weakly terminating composition of two workflow components. In this section, we present two basic classes of communicating components for which the communication condition can be decided based on their structure. The first class is based on marked graphs, and introduces concurrency within components and concurrent communication in compositions. The second class is based on state machines, to build more complex communication interactions and protocols.

5.1 Acyclic Marked Graph Components

A special subclass of Petri nets are marked graphs. In a *marked graph*, or T-net, all places have a preset and postset of length at most one. We extend this notion to components: a workflow component is a *T-component* if its skeleton is a marked graph, and all interface places are connected to exactly one transition. In [10], the authors show that if a T-workflow, i.e., a workflow that is also a T-net, is acyclic, it is generalized sound. We can use these results to obtain a similar result for T-components: if a T-component is acyclic, it is safe and weakly terminating.

Lemma 12 (Weak termination and safeness and T-components [10]). *Let N be a T-component such that $\mathcal{S}(N)$ is acyclic. Then it is weakly terminating and safe.*

By definition of a T-component, each interface place in a component has at most one transition connected to it. Hence, if two T-components are composable with respect to some port, in their composition these interface places have one transition in their preset and one transition in their postset, and thus the composition is again a T-component. From Lemma 12, we may directly conclude that if the composition of two T-components is acyclic, the composition is weakly terminating and safe.

Theorem 13 (Weak termination and safeness for compositions of T-components). *Let N and M be two acyclic T-components composable with respect to some port $G \in \mathcal{G}_N \cap \mathcal{G}_M$ such that $N \oplus_G M$ is acyclic. Then $N \oplus_G M$ is safe and weakly terminating.*

As a result, we may refine two synchronized places in a composition by an acyclic composition of T-components. The algorithm presented in the previous section can be used to determine the pairs of synchronized places. If places in a T-component are causal independent, i.e., it is possible that both places are marked in a place, then they are synchronized.

5.2 Isomorphic State Machine Components

A second subclass of Petri nets are state machines. A *state machine* is the dual of a marked graph: each transition has a preset and a postset of length at most one. A workflow component is an *S-component* if its skeleton is a state machine, and each interface place is connected to exactly one transition. From [10], it is easy to conclude that S-components are always weak terminating and safe.

Lemma 14 (Weak termination and safety of S-components). *Let N be an S-component. Then it is weakly terminating and safe.*

Although S-components have a simple structure, their composition is not. Composing S-components introduces concurrency; it is very simple to compose two

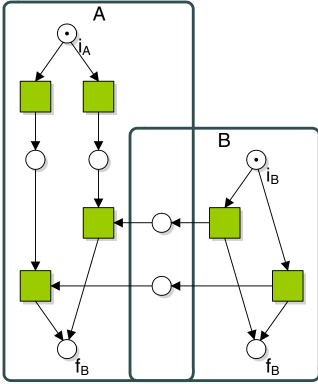


Fig. 6. Classical problem with point where decision is taken. A receiving gives problems.

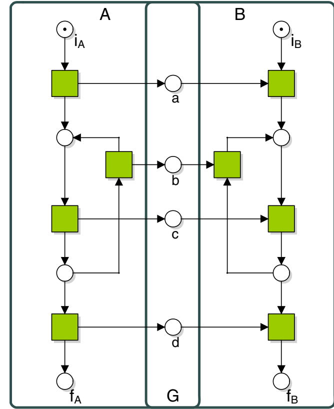


Fig. 7. The direction of communication via b and c matters. If both have the same sign, the composition $A \oplus B$ is not sound.

S-components such that the resulting composition can never reach the final marking, or even has deadlocks. We illustrate this with some examples. In Figure 6 we see a classical example showing that the composition of two sound S-components is not sound anymore. The choice A made is not communicated to component B. Hence, both can make a different choice, thus entering a deadlock. A solution to overcome this problem is to only connect S-components that have isomorphic skeletons, and communicate all choices. However, in Figure 7 we see two isomorphic S-components and although all choices made by A are communicated, we see that $A \oplus B$ is not weakly terminating, since in A the loop may be executed more times than component B executes the loop. Hence, tokens remain in the interface places, and thus the composition is not weakly terminating.

The composition of two T-components resulted in a T-component again. A similar property does not hold for S-components. The communicating transitions violate the state machine property in the composition: in the composition either their preset or their postset contain two places. These examples show that we need strong requirements to allow the composition of two S-components. First, we require the S-components to have isomorphic skeletons, and all transitions communicate, but only to the transition it is isomorphic to. Hence, all transitions either send or receive on the port that is used for the composition. Secondly, we require that if a component makes a choice, i.e., two or more transitions are enabled in a marked place, this choice is communicated, since otherwise the other component does not know in what state the first component is. This means that if two transitions share a place in their preset, they have the same sign. Last, we require that in every loop, there are at least two transitions with

a different sign. As all transitions either send or receive, the last requirement implies the existence of both a sending transition and a receiving transition in each loop. If all the requirements hold, we say that the composition agrees on the isomorphism.

Definition 15 (Composition agrees on isomorphism). *Let A and B be two S-components such that their skeletons are isomorphic with respect to ρ. The composition $N = A \oplus_G B$ for some port $G \in \mathcal{G}_A \cap \mathcal{G}_B$* agrees on ρ *if and only if:*

- *for all transitions $t \in T_A$, $t' \in T_B$, there exists a place $s \in G$ such that $\{(t,s),(s,t')\} \subseteq F_N$ or $\{(t',s),(s,t)\} \subseteq F_N$ if and only if $\rho(t) = t'$;*
- *All transitions in the postset of a place have the same sign, i.e. $\forall p \in P_N, t_1, t_2 \in p^\bullet : \lambda_G(t_1) = \lambda_G(t_2)$;*
- *For all markings $m \in \mathcal{R}(\mathcal{S}(A), i_A)$ and firing sequences $\sigma \in T_A^*$ such that $(\mathcal{S}(A) : m \xrightarrow{\sigma} m)$ there are $i, j \in \{0, \dots |\sigma|\}$ such that $\lambda_G(\sigma(i)) = ! \wedge \lambda_G(\sigma(j)) = ?$.*

Although isomorphism is a strong requirement, in practice it is often used for protocol design between agents. First a state machine is designed that represents the communication between the two agents. In each state of choice, only one agent can make a choice. Then the state machine is copied for both agents, and the communication between the transitions is added. Definition 15 gives a set of rules for asynchronous communication between these transitions such that the composition is always weakly terminating and safe.

Theorem 16. *Let N and M be two composable S-components with respect to some port $G \in \mathcal{G}_N \cap \mathcal{G}_M$ such that their skeletons are isomorphic with respect to some bijective function ρ. If the composition $N \oplus_G M$ agrees on ρ, $N \oplus_G M$ is weakly terminating and safe.*

If the skeletons of two components B and C are isomorphic, and their composition agrees on this isomorphism, the markings reachable in the composition have a special form: each marking consists of a marked place of B, a marked place of C and some marked interface places. As a consequence of the structure of the composition, it is always possible to mark a place p and its isomorphic place $\rho(p)$, without any interface marked.

In Figure 8 an example system is shown that is constructed using the presented approach. The system was constructed by starting with a single marked place. By the standard refinement rules of Murata [12], the component is refined to a simple marked graph with two places in parallel. Using the third rule, for both places a new component is created. Next, the second rule is applied twice to refine the two synchronized places by a composition of two S-components that agree on the isomorphism of their skeletons, and an acyclic composition of T-components. By construction, the system is weakly terminating.

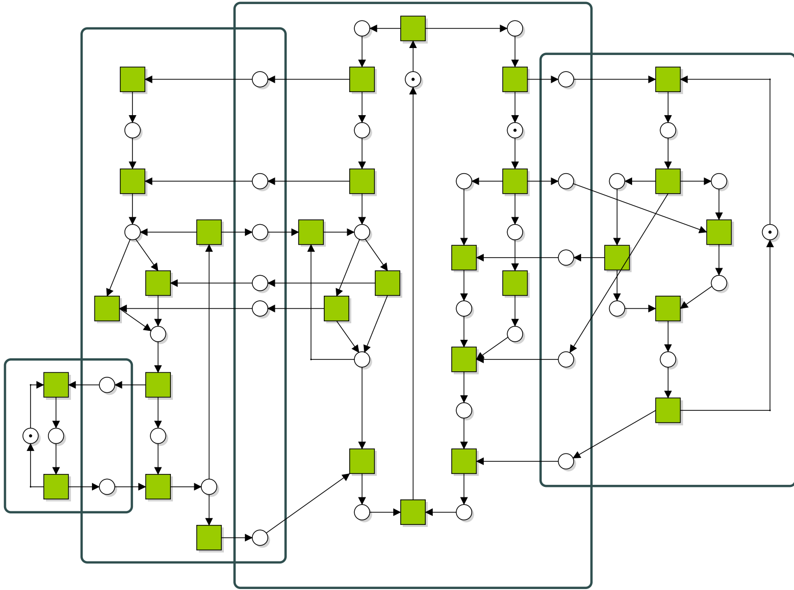


Fig. 8. System constructed using the constuction method

6 Conclusions

In this paper, we presented a construction methodology for asynchronously communicating systems that guarantees weak termination. We model components and their interaction with Petri nets. The construction method consists of three rules. The first rule allows for the refinement of safe places by a weakly terminating workflow component. If the initial component is weakly terminating, the refined component is also weakly terminating. In the second rule, we refine two synchronized places by a weakly terminating composition of two workflow components. A pair of places is synchronized if it is always the case that if one place is marked before the other, it is always possible to keep that place marked until the other is marked as well. If the original composition is weakly terminating, the refined composition is weakly terminating as well. The third rule allows the creation of new coupled components.

We studied two classes of coupled components that are weakly terminating by their structure. The first class is based on marked graphs. If a composition of two workflow T-components is an acyclic marked graph, the composition is weakly terminating. The second class is based on state machines. If the composition of two S-components that are isomorphic on their skeleton agrees on the isomorphism, the composition is weakly terminating.

In [6] the authors give a constructive method preserving the inheritance of behavior, which can be used to guarantee the correctness of interorganizational

processes. Other formalisms, like interface automata [3] use synchronous communication, whereas we focus on asynchronous communication, which is essential for our application domain, since the communication in SOA is asynchronous.

In [9], the authors propose to model choreographies using Interaction Petri nets, which is a special class of Petri nets, where transitions are labeled with the source and target component, and the message type being sent. To check whether the composition is functioning correctly, the whole network of components needs to be checked, whereas in our approach this is guaranteed by construction.

To keep our results at a conceptual level, we present our results on Petri net models. Our method can easily instaciated for industrial languages like BPEL, to facilitate the construction of web services in development environments like Oracle BPEL or IBM Websphere.

References

1. van der Aalst, W.M.P., Beisiegel, M., van Hee, K.M., König, D., Stahl, C.: An SOA-Based Architecture Framework. International Journal of Business Process Integration and Management 2(2), 91–101 (2007)
2. van der Aalst, W.M.P., van Hee, K.M., Massuthe, P., Sidorova, N., van der Werf, J.M.E.M.: Compositional service trees. In: Franceschinis, G., Wolf, K. (eds.) ICATPN 2009. LNCS, vol. 5606, pp. 283–302. Springer, Heidelberg (2009)
3. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes 26(5), 109–120 (2001)
4. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services Concepts, Architectures and Applications. Springer, Heidelberg (2004)
5. Alves, A., Arkin, A., Askary, S., et al.: Web Services Business Process Execution Language Version 2.0 (OASIS Standard). WS-BPEL TC OASIS (2007), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
6. Basten, T., van der Aalst, W.M.P.: Inheritance of Behavior. Journal of Logic and Algebraic Programming 47(2), 47–145 (2001)
7. Bell, M.: Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture. Wiley, Chichester (2008)
8. Berthelot, G.: Transformations and Decompositions of Nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 254, pp. 360–376. Springer, Heidelberg (1987)
9. Decker, G., Weske, M.: Local enforceability in interaction petri nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007)
10. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 337–356. Springer, Heidelberg (2003)
11. Kindler, E.: A compositional partial order semantics for petri net components. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 235–252. Springer, Heidelberg (1997)
12. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77(4), 541–580 (1989)
13. Object Management Group. Business Process Model and Notation (BPMN) Specification 2.0 V0.9 (November 2008)
14. Reisig, W.: Petri Nets: An Introduction. Monographs in Theoretical Computer Science: An EATCS Series, vol. 4. Springer, Berlin (1985)

An Advice for Advice Composition in AspectJ

Fuminobu Takeyama and Shigeru Chiba

Tokyo Institute of Technology, Japan
http://www.csg.is.titech.ac.jp/~{f_takeyama, chiba}

Abstract. Aspect composition often involves advice interference and this is a crucial problem in aspect oriented programming. When multiple advices are woven at the same join point, the advices often interfere with each other. Giving appropriate precedence order is a typical solution of this problem but it cannot resolve all kinds of advice interference. To address this problem, we propose a novel language extension named *Airia*, which provides a new kind of around advice for resolving advice interference. This kind of advice named a *resolver* is invoked only at the join points when given advices conflict with each other. The resolvers can call an extended version of proceed, which takes as an argument precedence order among remaining advices. Furthermore, the resolvers are composable. They can be used to resolve interference among other resolvers and advices.

1 Introduction

In aspect-oriented programming, crosscutting concerns are modularized into aspects. Composing a new aspect by combining existing aspects is not easy. The aspects often conflict with each other and cause undesirable interference. This problem is called *aspect interference*. In particular, resolving interference among advices in the aspects is a serious issue that has been investigated in the research community. This paper addresses this issue called *advice interference*.

A typical solution is to allow programmers to control the precedence order among conflicting advices. For example, AspectJ [10] provides declare precedence for this control. However, for some combinations of advices, there is no correct precedence order with which the composed behavior is acceptable. Such combination needs modifying the bodies of the conflicting advices to explicitly implement the merged behavior. This is not desirable since the programmers have to be aware of the composition when they write an individual advice. The implementation of the composition should be described separately from the conflicting advices.

This paper proposes a novel extension of AspectJ, which is named *Airia*. In this language, a new kind of around advice called *resolvers* are available. A resolver is used to implement the composition of conflicting advices. It is invoked at the join points when the given set of advices conflict with each other. Since a resolver has higher precedence than those conflicting advices, it overrides the implementation of those advices. In the body of the resolver, a proceed call takes

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 122–137, 2010.
© Springer-Verlag Berlin Heidelberg 2010

```
public class Expression extends ASTNode {...}

public class Plus extends Expression {
   private Expression left;
   private Expression right;

   public Plus(Expression left, Expression right) {...}

   public Expression getLeft() {...}
   public Expression getRight() {...}
   // :
}

public class Constant extends Expression {
   Object value;
   public Constant(Object value) {
      this.value = value;
}}

aspect Evaluation {
   public Object Expression.eval() {
      return null;
}}
```

Listing 1. Classes representing the AST

as an argument the precedence order among those conflicting advices. It can thereby control the execution order of the remaining advices. In our language Airia, therefore, declare precedence is not available. Furthermore, a resolver is composable. It can implement the composition of other resolvers and advices. Our language Airia has been implemented by using the AspectBench compiler [4] and JastAdd [8] [1]. In summary, the contribution of this paper is to propose a new language construct named a *resolver*. The advantages are two. (1) It enables separation of the implementation of advice composition while keeping sufficient expressive power. Also, (2) it is composable and thus we can implement composition of several advices in a hierarchical manner.

In the rest of this paper, Section 2 shows a motivating example. Section 3 presents the design of a resolver. Section 4 describes implementation of Airia compiler. Section 5 mentions related work and Section 6 concludes this paper.

2 A Motivating Example

We first show an example of aspect interference that is not resolved in existing approaches.

2.1 A Simple Interpreter

We present a simple interpreter with a binary operator +, which is written in AspectJ. Listing 1 shows classes representing AST (Abstract Syntax Tree) nodes. The Plus class expresses a binary operator +. It has two fields, left and

[1] The source code of Airia is available from: `http://www.csg.is.titech.ac.jp/`

```
aspect IntegerAspect {
    Object around(Plus t): target(t) && execution(Object Plus.eval()) {
        return (Integer)t.getLeft().eval() + (Integer)t.getRight().eval();
}}
```

Listing 2. An aspect for integer values

```
aspect StringAspect {
    Object around(Plus t): target(t) && execution(Object Plus.eval()) {
        return t.getLeft().eval().toString() + t.getRight().eval().toString();
}}
```

Listing 3. An aspect for character strings

right, representing its operands and it extends the Expression class. We declare a method for evaluating an AST in the EvaluationAspect aspect. Since our interpreter currently does not support any data types, this aspect appends an empty eval method to the Expression class by an inter-type declaration.

We then extend the interpreter to support integer values. We do not have to modify the existing classes. We have only to write a new aspect shown in Listing 2. The around advice in the IntegerAspect aspect is invoked when the Plus.eval method is executed; it returns a summation of the two operands. The following code makes an AST representing 1 + 2. When e.eval is executed on this tree, it returns 3.

```
Expression e = new Plus(new Constant(1), new Constant(2));
```

Next, we extend the original interpreter to support character strings. Again, we do not have to modify the existing classes. We implement this extension by the StringAspect in Listing 3. Since the operator + now represents concatenation of character strings, the around advice implements the behavior of the eval method. If we compile the classes and the aspects in Listing 1 and 3, then the resulting interpreter will correctly handle character strings.

The last step is to build an interpreter supporting both integers and character strings. Some readers might expect that we could easily obtain the interpreter if the two aspects IntegerAspect and StringAspect are compiled and woven together. However, these two aspects conflict with each other, *i.e.* multiple advices are woven at the same join point. The resulting behavior of the eval method is different from our naive expectation. This paper deals with this unexpected behavior of the combined advices, which we call *advice interference*. It is one kind of *aspect* interference; other kinds of aspect interference such as [9] are out of the scope of this paper.

If the two advices are combined, the resulting eval method cannot process all acceptable ASTs. In AspectJ, when the eval method is called, the advice with the highest precedence is executed. Suppose that IntegerAspect has the highest.

```
aspect IntegerAspect {
   Object around(Plus t): target(t) && execution(Object Plus.eval()) {
      Object lvalue = t.getLeft().eval();
      Object rvalue = t.getRight().eval();
      if (lvalue instanceof Integer && rvalue instanceof Integer) {
         return (Integer)lvalue + (Integer)rvalue; // not for composition
      } else {
         return proceed(t);
}}}

aspect StringAspect {
   Object around(Plus t): target(t) && execution(Object Plus.eval()) {
      Object lvalue = t.getLeft().eval();
      Object rvalue = t.getRight().eval();
      if (lvalue instanceof String || rvalue instanceof String) {
         return lvalue.toString() + rvalue.toString(); // not for composition
      } else {
         return proceed(t);
}}}
```

Listing 4. A composable version of the conflicting advices

Then the eval method does not process the AST e constructed by the following code:

```
Expression e = new Plus(
    new Constant("Hello "), new Constant("world!"));
```

It will throw ClassCastException since the operands are character strings but the around advice in IntegerAspect assumes that the operands are integer values. Changing the precedence order does not solve this problem. In AspectJ, we can explicitly specify precedence. For example,

```
declare precedence: StringAspect, IntegerAspect;
```

this declaration specifies that StringAspect has higher precedence than IntegerAspect. The eval method now returns an unexpected value when the AST e2 shown below is evaluated:

```
Expression e2 = new Plus(new Constant(1), new Constant(2));
```

The returned value will be a character string "12" although both operands are integer values.

2.2 An Incomplete Solution in AspectJ

A partial solution of the problem above is to make advices composable by *linearization*. Since AspectJ provides proceed calls, we can reimplement advices and connect them by proceed to make a single chain. If proceed is called in an advice body, it invokes the advice with the next highest precedence in the chain. If there is no other advice, the original computation at the join point is executed.

Listing 4 shows the result of the reimplementation to use the linearization. The resulting code can be regarded as an AspectJ version of the chain of responsibility pattern. Now the two around advices call proceed to invoke the next advice when

```
aspect IntegerStringAspect {
   Object resolver plusEvalIntStr(Plus t)
            and(IntegerAspect.plusEvalInt(t), StringAspect.plusEvalStr) {
      Object lvalue = t.getLeft().eval();
      Object rvalue = t.getRight().eval();
      if (lvalue instanceof String || rvalue instanceof String) {
         return [StringAspect.plusEvalStr].proceed(t);
      } else if (lvalue instanceof Integer && rvalue instanceof Integer) {
         return [IntegerAspect.plusEvalInt].proceed(t);
      } else {
         throw new RuntimeException();
}}}
```

Listing 5. The aspect for combining IntegerAspect and StringAspect

they cannot deal with the given operands. Although the combination of some advices requires an explicit declaration of the precedence order for linearization, the two aspects in Listing 4 do not require it; they correctly work under any precedence order. The interpreter containing these two aspects can deal with the AST constructed by this code:

```
Expression e = new Plus(new Constant("Str") + new Constant("1"));
```

However, this solution is not satisfactory from the software engineering viewpoint. Programmers need global reasoning for combining advices; they must be aware of other (maybe unknown yet) advices. Furthermore the implementation of each advice body includes the composition code for linearization, which connects it to other advices. Programmers have to design a composition protocol for the advice chain before implementing each advice body. The protocol design is not easy since the advices must be able to correctly work with and without other advices.

The composition is a crosscutting concern. Note that most statements in Listing 4 are for the composition by the linearization. Only the two return statements marked by a comment implement the behavior of the eval method in the Plus class. The composition code scatters over the two advices.

3 A New Approach for Resolving Interference

To resolve the problem mentioned above, we propose a novel language extension of AspectJ. This new language named *Airia* allows programmers to separately describe how to resolve advice interference. Instead of the conflicting advices themselves, the resolving code is described in a new kind of around advice called a *resolver*. Hence the implementation of each conflicting advice is independent of the other conflicting advices and their composition protocol.

Listing 5 is an example of an aspect including a resolver. It resolves the advice interference presented in the previous section. Details of this resolver are mentioned below. Since the resolving code is separated into this resolver, the conflicting advices do not include the code for composition or resolution of the interference. See Listing 6, which presents the three conflicting advices written

```
aspect Evaluation {
   public Object Expression.eval() {
      return null;
}}

aspect IntegerAspect {
   Object around plusEvalInt(Plus t): target(t) && execution(Object Plus.eval()) {
      return (Integer)t.getLeft().eval() + (Integer)t.getRight().eval();
}}

aspect StringAspect {
   Object around plusEvalStr(Plus t): target(t) && execution(Object Plus.eval()) {
      return t.getLeft().eval().toString() + t.getRight().eval().toString();
}}
```

Listing 6. The aspects written in our language

in our language. They are simpler than the composable version of the aspects shown in Listing 4. They are the same as the original aspects in Listing 2 and 3 except that every advice has a unique name. These advice names are used by the resolver.

A resolver is composable. Programmers can write a resolver that resolves interference among other resolvers and normal advices. Suppose that we write a new aspect EvaluationCacheAspect and its advice causes interference with the advices of IntegerAspect and StringAspect. For these three conflicting advices, we can write a new resolver by reusing the existing resolver of IntegerStringAspect. Since the resolver of IntegerStringAspect deals with the advice interference between IntegerAspect and StringAspect, the new resolver will be declared to deal with the interference between the resolver of IntegerStringAspect and the new advice of EvaluationCacheAspect. The implementation of that new resolver will call proceed to execute the resolver of IntegerStringAspect.

3.1 A Resolver

A resolver is a special around advice, which is declared with a keyword resolver instead of around. The syntax of resolver declaration is the following:

```
RetrunType resolver ResolverName(ArgumentType ArgumentName, ...)
    and|or(ConflictingAdviceName[(BoundArgumentName, ...)], ...)
    [uses HelperAdviceName, ...] { Body }
```

The resolver keyword is followed by a resolver name. A parameter list to the resolver follows the resolver name if any. Unlike normal advices in AspectJ, it does not take a pointcut but it takes an and/or clause, which specifies a list of potentially conflicting advices. The resolver is expected to resolve interference among these advices. Except the resolver keyword, its name, and the and/or clause, a resolver is the same as an around advice. The return type of a resolver is Object if the join points bound to the resolver have different return types. The body of the resolver may include a proceed call.

In Listing 5, a resolver is named plusEvalIntStr and takes an and clause, which lists the names of the around advices in the two aspects IntegerAspect and StringAspect. Note that an advice also has a unique name. See Listing 6. The IntegerAspect aspect has an advice named plusEvalInt and the StringAspect has an advice named plusEvalStr. IntegerAspect.plusEvalInt and StringAspect.plusEvalStr are their fully-qualified names.

The join points when a resolver is executed are specified by an and/or clause. Since the resolver in Listing 5 has an and clause, it is executed at the join points that all the given advices are bound to, that is, when the eval method in the Plus class is executed. Note that those advices of the two aspects IntegerAspect and StringAspect conflict at those join points. A resolver has higher precedence than the advices specified by its and/or clause. Hence, it *overrides* all the conflicting advices at the join points. In our example, when the eval method in the Plus class is called, the body of the resolver is executed first.

The advices given to the and/or clause of a resolver work as pointcuts. Thus, a resolver can take parameters and pass them to those advices. For example, the resolver in Listing 5 takes a parameter t and passes it to the advice in the IntegerAspect. The parameter t is bound to the value that this advice binds its parameter to, that is, the target object of the call to the eval method.

A resolver may have an or clause. This specifies that the resolver is executed at the join points that at least one advice given to the or clause is bound to. For example, the next resolver is executed at the join points that only the two advices A and B are bound to but C is not:

```
Object resolver precedence() or(A, B, C) {
    return [A, B, C].proceed();
}
```

The or clause can be used for specifying precedence order among advices as we do with declare precedence in AspectJ. The resolver shown above specifies that the precedence order is A, B, and C. `[A, B, C].proceed()` executes the three advices in that order (we below mention this proceed call again).

We introduced an or clause for reducing the number of necessary resolvers. If we could not use an or clause, we would have to define many resolvers for all possible combinations of potentially conflicting advices. Suppose that we have three advices A, B, and C. We would have to define resolvers for every combination: A and B, B and C, C and A, and all the three, if they conflict at different join points. Since we expect that those combinations would share the same body, using an or clause would reduce the number of resolvers we must describe.

To be precise, the join points selected by an and/or clause are the intersection/union of the join point *shadow* [14] selected for the advices given to that and/or clause, respectively. Dynamic pointcuts such as cflow and target are ignored. Thus, a resolver may be executed at the join points that the advices in its and/or clause are not bound to.

We adopted this language design since it is extremely difficult to detect conflicts among advices even at runtime. Since an advice in AspectJ can change

the dynamic contexts, after its body is executed, an advice with a lower precedence than that advice may be removed from the set of the executable advices at that point. Suppose that the pointcut of an advice includes if(Expr.flag) and the value of Expr.flag is true. If an advice with higher precedence than that advice sets Expr.flag to false before calling proceed, the advice with if(Expr.flag) will not be executed by the proceed call.

3.2 A Proceed Call with Precedence

Like a normal advice, a resolver can call proceed to invoke another advice with the next highest precedence. The proceed call from a resolver explicitly specifies the precedence order of the advices given to the and/or clause, which will be invoked by the proceed call. Note that unlike AspectJ our language Airia does not provide declare precedence. The precedence order is described between brackets preceding .proceed.

Suppose that there are two advices A and B and they conflict at the join point selected by a pointcut pc(). We assume that there is no other advices. Then a resolver AorB can call proceed twice with different precedence order:

```
void resolver AorB() or(A, B) {
    [A, B].proceed();
    [B, A].proceed();
}
pointcut pc(): ...;
void around A(): pc() {
    proceed();
}
void around B(): pc() {
    proceed();
}
```

When [A, B].proceed() is called, A is invoked. B is invoked by the proceed call in A. The proceed call in B executes the original computation at the join point. On the other hand, when [B, A].proceed() is called, B is invoked. A is the next. Note that [A, B].proceed() does not mean that A and then B. It means that A has higher precedence than B; A or B may not be executed when their pointcuts do not match the current join point.

The proceed call can remove advices from the set of the remaining advices, which will be invoked by later proceed calls. If the advice list between brackets does not include an advice given to the and/or clause, the advice is removed. In Listing 5, both proceed calls remove one advice. The former removes IntegerAspect.plusEvalInt and the other removes StringAspect.plusEvalStr. For example, [IntegerAspect.plusEvalInt].proceed() invokes the plusEvalInt advice in IntegerAspect and then, if it calls proceed again, the original eval method is invoked. The plusEvalStr advice in StringAspect is never invoked.

```
aspect EvaluationCacheAspect {
    Object Expression.cachedValue;
    boolean Expression.isChanged = false;
    void around plusEvalCache(Expression t): execution(Object Plus.eval()) && args(t) {
        if (t.isChanged) {
            cachedValue = proceed(t);
            isChanged = false;
        }
        return cachedValue;
    }
    after changed(): ... {
        isChanged = true;
}}
```

Listing 7. The EvaluationCacheAspect aspect

```
aspect IntegerStringCacheAspect {
    Object resolver evalIntStrCache():
            and(IntegerStringAspect.evalIntStr, EvaluationCacheAspect.plusEvalCache)
        return [EvaluationCacheAspect.plusEvalCache,
                IntegerStringAspect.evalIntStr].proceed();
}}
```

Listing 8. A resolver resolving conflicts between a normal advice and another resolver

3.3 Composability of Resolvers

A resolver, which is a special around advice, may also conflict with other resolvers or normal advices. This conflict can be also resolved by another resolver; a resolver is composable. An advice given to an and/or clause may be a resolver. A proceed call with precedence specifies precedence order among conflicting advices and/or resolvers.

Let us consider a new advice shown in Listing 7. The join point of this advice is the execution of the eval method. Thus, this advice conflicts with the two advices in IntegerAspect and StringAspect shown in Listing 2 and 3. Since the conflict between these two advices has been already resolved by the resolver in IntegerStringAspect, we reuse this resolver to resolve the conflicts among the new advice and these two advices. See Listing 8. This resolver in IntegerStringCacheAspect has an and clause, which lists the new advice in EvaluationCacheAspect and the resolver in IntegerStringAspect. It resolves conflicts between the advice and the resolver.

The behavior of a resolver for another resolver is the same as normal resolvers. When the eval method is called, this resolver in IntegerStringCacheAspect is invoked first since it has higher precedence than the other advices and resolver. When this resolver calls proceed with precedence, the advice with the next highest precedence is executed, which is the advice in EvaluationCacheAspect. After that if the advice calls proceed, the resolver in IntegerStringCacheAspect is executed. Note that this resolver does not explicitly describe how the conflicts between IntegerAspect and StringAspect are resolved. It is encapsulated in the resolver of IntegerStringAspect. The composition of IntegerStringCacheAspect is hierarchical.

Table 1. A summary of precedence relations declared by constructs in Airia

Construct	Precedence relations
Type resolver R() and/or(A, B, C)	R $\prec$ A, R $\prec$ B, R $\prec$ C
[A, B, C].proceed()	A $\prec$ B, B $\prec$ C

Existing resolvers can be overridden when it cannot be reused. To implement a new resolver that changes the precedence order given by another resolver, programmers explicitly remove the other resolver. For example, if a new resolver requires the resolver in `IntegerStringAspect` should have higher precedence than the advice in EvaluationCache, the resolver in `IntegerStringCacheAspect` is removed by the same way that a resolver removes a normal advice. The removed resolver is not executed; the new resolver can define a new precedence order among the advices that were resolved by the removed one.

Unlike declare precedence in AspectJ, a resolver can flexibly modify precedence order among conflicting advices even during runtime by a proceed call with precedence. Thus, declare precedence is not available in our language Airia. The precedence order must be explicitly specified; there is no default precedence order unlike AspectJ.

3.4 A Compile Time Check of Conflict Resolution

Our language Airia requires that all conflicts among advices should be explicitly resolved by resolvers. Our compiler checks this requirement at compile time. If programmers declare inconsistent precedence or forget to specify precedence among advices, then our compiler will report errors.

To enable statically checking whether conflicts are resolved or not, our definition of conflict is conservative like AspectJ. If advices partly share their join point shadow, they conflict. Due to our specification mentioned in Section 3.1, a resolver cannot have dynamic residue. Thus our compiler can statically determine conflicting advices at every join point shadow and the resolvers executed at that shadow.

Our compiler recognizes that a conflict has been resolved if the advice or resolver with the next highest precedence is always determinable. Also the highest resolver executed first at the join point must be uniquely determined. Recall that, in our language Airia, constructs that declare precedence order are a resolver and a proceed call with precedence as summarized in Table 1. The and/or clause declares that the resolver has higher precedence than the advices or resolvers specified in it. Then the proceed call declares precedence order as specified between its brackets. This declarations are effective only in the remaining chain of advices. Here we use a binary relation; X $\prec$ Y represents that X has higher precedence than Y. This relation is transitive, *i.e.*, if X $\prec$ Y and Y $\prec$ Z then X $\prec$ Z. It must be total order; otherwise, it causes an error. Suppose three advices A, B, C and the following resolvers:

```
void resolver R() and(A, C, S) {
    [S, C, A].proceed();
}
void resolver S() and(A, B) {
    [A, B].proceed();
}
```

They conflict at the same join point shadow and no other advices nor resolvers exists. The first executed resolver is R because of $R \prec S$ given by the and clause of R. After S is invoked by the proceed call in R, the proceed call in S invokes C. This is because of $S \prec C$ and $C \prec A$ given by the proceed call in R. Thus this conflict is resolved. On the other hand, if we rewrite R as follows, then the conflict is not resolved:

```
void resolver R() and(C, S) {
    [S, C].proceed();
}
```

The declared relations are only $R \prec S$, $S \prec C$, $S \prec A$, and $A \prec B$; there is no precedence order between A and C.

The precedence order declared by a and/or clause cannot be removed. Even if S is removed by another resolver in the example above, $S \prec A$ and $S \prec B$ are still effective. Without this rule, the check of conflict resolution would be extremely complicated. The following resolvers would be valid:

```
void resolver T() and(U, D) {
    [D].proceed(); //remove U
}
void resolver U() and(T, D) {
    [D].proceed(); //remove T
}
```

They declare $T \prec U$ and $U \prec T$ and thus the precedence order seems to have a cycle. However, if we first pick up T, since T removes U, the result would be only $T \prec U$ and $T \prec D$, which has no cycle. On the other hand, if we first pick up U, since U removes T, the result would be different precedence order including no cycle. We have introduced the rule to avoid this complication and ambiguity.

Some resolvers include multiple proceed calls declaring different precedence order. The next advice invoked at a proceed call is determined dependently on the chain of proceed calls executed so far at the current join point. Our compiler checks conflict resolution along every conservatively possible control path. Please refer to our companion paper [17] for more detail.

3.5 A Helper Advice

A resolver can add a new advice for helping composition. Since a resolver has higher precedence than conflicting advices, the added advice is normally given intermediate precedence among those conflicting advices.

```
aspect TraceLogging {
   before log(): ... {
      Logger.getInstance().debug(thisJoinPointStaticPart.toString());
}}

aspect ArgumentLogging {
   before log(): ... {
      Object[] args = thisJoinPoint.getArgs();
      Logger.getInstance().debug("Arguments: " + Arrays.toString(args));
}}
```

Listing 9. Two aspects for logging

```
aspect LoggingWithSync {
   before lock() {
      Logger.getInstance().lock(); //reentrant lock
   }
   before unlock() {
      Logger.getInstance().unlock();
   }

   void resolver sync()
         and(TraceLogging.log, ArgumentLogging.log) uses lock, unlock {
      [lock, TraceLogging.log, ArgumentLogging.log, unlock].proceed();
}}
```

Listing 10. A resolver for synchronizing two aspects

Suppose that we have two logging aspects shown in Listing 9. The advice in the TraceLogging aspect records executed methods during program execution. The ArgumentLogging aspect records the values of arguments when a method is invoked. If the precedence order specifies that TraceLogging is executed before ArgumentLogging, then a printed method name is followed by argument values. However, if a program is multi-threaded, the two advices must be synchronized. Otherwise, printed log messages will be interleaved as the following:

```
[DEBUG] execution(Object Main.run(String))
[DEBUG] execution(void Test.test())
[DEBUG] Argument: []
[DEBUG] Argument: [--debug]
```

Here, the fourth line shows the value of the argument to the run method.

Listing 10 shows a resolver for synchronizing the two logging advices. This resolver uses two helper advices lock and unlock. Note that this resolver has a uses clause, which specifies the helper advices for that resolver. The pointcut of a helper advice is not explicitly specified; a helper advice is bound to the same join points that the resolver using that helper advice is bound to. The helper advices are included in the precedence order of proceed. In Listing 10, the lock advice is given the highest precedence while the unlock advice is given the lowest precedence among the four advices. Thus, the lock advice acquires a lock, the logging advices print messages, and then the unlock releases before the method logged by the aspects is executed. Without these helper advices, the resolver could not implement synchronization since it had to release a lock between the

```
aspect IntegerStringAspect {
  Object around(Plus t): execution(Object Plus.eval()) && target(t) {
    Object lvalue = t.getLeft().eval();
    Object rvalue = t.getRight().eval();
    if (lvalue instanceof String || rvalue instanceof String) {
      return lvalue.toString() + rvalue.toString();
    } else if (lvalue instanceof Integer && rvalue instanceof Integer) {
      return (Integer)lvalue + (Integer)rvalue;
    } else {
      throw new RuntimeException();
  }}
  declare precedence: IntegerStringAspect, IntegerAspect, StringAspect;
}
```

Listing 11. Another incomplete solution in AspectJ

logging advices and the logged method but the resolver automatically obtains higher precedence than the logging advices.

Multiple resolvers may use the same helper advice. If those resolvers are bound to the same join points, that helper advice is executed only once at every join point (shadow). If a resolver removes another resolver using a helper advice, that helper advice is not removed together. It must be explicitly removed.

3.6 Discussion

A resolver does not take a normal pointcut but an and/or clause — a list of conflicting advices. It can call proceed with precedence. These are unique features of our language Airia. To clarify their benefits, we show another aspect in Listing 11. Like the aspect written in Airia, this aspect does not require us to modify the conflicting aspects in Listing 2 and 3. We wrote this aspect in AspectJ to be similar to the aspect written in Airia shown in Listing 5. The aspect has a normal around advice. We manually translated the and clause of the resolver into a normal pointcut for this around advice. In the body of this around advice, we also manually inlined the body of the conflicting advices since a proceed call with precedence is not available.

This aspect has two drawbacks. First, the pointcut of the advice is fragile. We will have to modify the pointcut of this advice when the pointcuts of the conflicting advices are modified. Second, the body of this advice contains code duplication since we manually inlined the body of the conflicting advices. We will also have to modify the advice body when the bodies of the conflicting advices are modified. The aspect written in Airia does not have these problems.

4 Implementation

We have implemented an Airia compiler by extending an AspectJ compiler named the AspectBench compiler (abc). Its front-end is implemented as an extension of the JastAddJ extensible compiler. The extension to abc is implemented by aspects like the example in Section 2.

The current implementation of our compiler does not support the same level of optimization as abc. Unlike abc, our `proceed` call is implemented only by using closure objects. The size of generated code could be large. A closure is created for each `proceed` call of every path of advice chain because the selection of invoked advice by the `proceed` call depends on which `proceed` calls have been executed at the join point.

5 Related Work

Resolving aspect interference for aspect composition is a classic research topic and hence there have been a number of proposals. For example, Douence *et al.* proposed an approach for detecting and resolving conflicts between aspects on their formal framework, Stateful Aspect [6,7]. Their approach is making a composition operator extensible so that the operator will generate correctly merged behavior when the operands are conflicting with each other. Although this approach is similar to ours, we provide a single composition operator (*i.e.* a resolver) but we do not make the semantics of the operator extensible. We also propose an extension of AspectJ based on our approach.

Most previous approaches are categorized into meta programming. POPART [5] provides a meta-aspect protocol. Advice composition can be dynamically customized by an instance of `MetaAspectManager`. Programmers can define an appropriate `MetaAspectManager` to implement a custom composition policy for resolving conflicts among particular advices. JAsCo [16] also provides a mechanism like this. However, meta programming is often complicated and difficult. Thus, in OARTA [13], the ability for meta programming is restricted. For resolving aspect interference, OARTA allows an advice to modify only the pointcut of another advice. On the other hand, our approach does not need meta programming. A resolver is a special around advice but it is still a base-level language construct.

Context-Aware Composition Rules [11,12] allows programmers to control precedence order among advices for every join point. It also allows removing an existing advice at some join points. However, as we mentioned in Section 2, some kinds of advice interference cannot be resolved by only reordering advices. On the other hand, our language Airia also provides the ability for adding a new advice only at the join points where advices are conflicting with each other.

Reflex [18] is an infrastructure for building an aspect system. It provides an application-programming interface (API) for implementing a new policy for advice composition. Programmers can exploit this API for customizing the aspect system to resolve conflicts among particular advices. On the other hand, our language Airia provides a base-level language construct for resolving conflicts. The users of Airia do not have to consider the implementation of the language.

Aspect refinement and mixin-based aspect inheritance [2,3] enable programmers to incrementally extend the behavior of an existing advice. JastAdd also supports refinement. On the other hand, our language Airia enables extending the behavior of a combination of multiple advices.

The relationship between an around advice and the original computation at the join point is similar to the relationship between a mixin and a class. A `proceed` call corresponds to a `super` call. The former executes the next advice or the original computation while the latter executes the method in the next mixin or class. While advice interference is a problem, interference among mixins given to the same class is also a problem. Traits [15] is a solution for mixin interference. Our approach can be regarded as an application of the idea of traits to aspects. Both approaches allow programmers to define a new advice/method for overriding advices/methods and resolving their conflicts.

6 Conclusion

We presented a language extension of AspectJ. This language named Airia can resolve interference among conflicting advices, which we could not satisfactorily resolve in original AspectJ. Airia enables programmers to separate composition code into an independent resolver, which is a new kind of advice. A resolver is composable. It can resolve interference between another resolver and an advice. We have implemented an Airia compiler by extending the AspectBench compiler using JastAdd.

Our future work includes improving the expressive power of `proceed` calls. There are some proposals such as [1], which are used to detect whether or not advices are commutative, *i.e.* whether or not their combined behavior is independent of their precedence order. In the current design of Airia, programmers have to explicitly specify the precedence order among advices even though they are commutative. This is annoying.

References

1. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: AOSD 2009: Proceedings of the 8th ACM International Conference On Aspect-Oriented Software Development, pp. 39–50. ACM, New York (2009)
2. Apel, S., Leich, T., Saake, G.: Aspect refinement and bounding quantification in incremental designs. In: Asia-Pacific Software Engineering Conference, pp. 796–804 (2005)
3. Apel, S., Leich, T., Saake, G.: Mixin-based aspect inheritance. Technical Report Number 10. Department of Computer Science, University of Magdeburg, Germany (2005)
4. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Abc: an extensible AspectJ compiler. In: AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, pp. 87–98. ACM, New York (2005)
5. Dinkelaker, T., Mezini, M., Bockisch, C.: The art of the meta-aspect protocol. In: AOSD 2009: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, pp. 51–62. ACM, New York (2009)

6. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: GPCE 2002: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, pp. 173–188. Springer, London (2002)
7. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: AOSD 2004: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, pp. 141–150. ACM, New York (2004)
8. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA 2007: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications, pp. 1–18. ACM, New York (2007)
9. Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: A graph-based approach to modeling and detecting composition conflicts related to introductions. In: AOSD 2007: Proceedings of the 6th International Conference on Aspect-Oriented Software Development, pp. 85–95. ACM, New York (2007)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
11. Marot, A., Wuyts, R.: Composability of aspects. In: SPLAT 2008: Proceedings of the 2008 AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies, pp. 1–6. ACM, New York (2008)
12. Marot, A., Wuyts, R.: A DSL to declare aspect execution order. In: DSAL 2008: Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages, pp. 1–5. ACM, New York (2008)
13. Marot, A., Wuyts, R.: Composing aspects with aspects. In: AOSD 2010: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, pp. 157–168. ACM, New York (2010)
14. Masuhara, H., Kiczales, G., Dutchyn, C.: Compilation semantics of aspect-oriented programs. In: Proc. of Foundations of Aspect-Oriented Languages Workshop, AOSD 2002, pp. 17–26 (2002)
15. Schrli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 327–339. Springer, Heidelberg (2003)
16. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development. In: AOSD 2003: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pp. 21–29. ACM, New York (2003)
17. Takeyama, F.: A new kind of advice for advice composition without interference. Master's thesis, Tokyo Institute of Technology, Japan (2010)
18. Tanter, E.: Aspects of composition in the Reflex AOP kernel. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 98–113. Springer, Heidelberg (2006)

The .NET Primitives for Open, Dynamic and Reflective Component Frameworks

Mircea Trofin[1], Nicholas Blumhardt[2], and Clemens Szyperski[1]

[1] Microsoft Corporation
{mirceat,clemens}@microsoft.com
[2] Readify
nicholas.blumhardt@readify.net

Abstract. "Composition Primitives" is a novel component model targeting .NET. The model facilitates composition across component programming frameworks via an adaptation mechanism external to the component. Constructing adapters is relatively inexpensive, because the model is minimal and focused on just one concern: offering enough information to support composition. Although small, the model supports static discovery of the services provided and consumed by a component—in other words, it is reflective. To strengthen the value of its reflection capabilities, it purposely does not rely on the Service Locator pattern and it supports n–order composition scenarios. In this paper, we present our model and support our claims.

1 Introduction

We present a component model that serves as a foundation for creating Open, Dynamic .NET applications built out of Reflective components. These components may have been developed in a domain–specific programming model, or may have been developed in a different, possibly legacy, component framework.

We will use the acronym "ODR" for these kinds of applications. In ODR applications, third–party functionality (components) can be added or removed (the "open" quality), possibly while the application is running (the "dynamic" quality), and there are first–class means to statically determine, for such third–party functionality, what facilities it provides and what its requirements are (the "reflective" quality). Reflection typically needs to be performed without loading any component code to avoid associated performance penalties. Eager loading tends to become prohibitive in applications with a large number of components that do not need to be all loaded upfront—like an integrated development environment or a web browser.

Examples of existing component frameworks targeting the construction of ODR applications include: CORBA Component Model [1], Castle MicroKernel/Windsor [2], Autofac [3], and Fractal [4].

In our model we focus on discovery and basic composition. We believe this is the core concern of any component framework, and, therefore, other aspects commonly covered by component frameworks can be realized separately or built

B. Baudry and E. Wohlstadter (Eds.): SC 2010, LNCS 6144, pp. 138–153, 2010.
© Springer-Verlag Berlin Heidelberg 2010

on top of our model. We designed the Managed Extensibility Framework (MEF), part of .NET 4.0, as one such more comprehensive framework, validating our layering.

In particular, a characteristic property of components is that they form units of versioning [6]. Like other higher-level concerns, the primitives do not address versioning directly. Instead, it is left to component frameworks built on top of the primitives to address such concerns appropriately. While the details are beyond the scope of this paper, MEF, for instance, relies on the underlying versioning semantics in .NET and supports adapters to deal with additional versioning issues.

In contrast to other models targeting ODR applications, we claim our model offers the following novel and differentiating capabilities:

Our model supports creating domain-specific programming models and facilitates composition across component frameworks. A "programming model", in this context, is syntax and semantics defining components[1]. Since .NET is a multi–language platform, the term "syntax" refers to both language–specific, as well as language–independent means of expression. An extreme example of the former would be a language supporting the keyword `component`, with a compiler targeting the .NET Common Instruction Language (CIL) specification [5]. An example of the latter is the use of generally–supported .NET concepts, like custom attributes, types, or properties, to define what a component is, regardless of language. For example: any type annotated with the custom attribute `Component` is a component.

Choosing a component framework is an architectural decision [6]. It is hard to move away from such decisions: As applications evolve, it becomes important to enable interoperability with components written for other component frameworks—perhaps more recent ones. It is always possible to enable this—one can always write custom adapters and wrap components on a case-by-case basis. This tends to be expensive. We offer a solution for some of the most repetitive problems, such as discovery, without using the classical solution of a Service Locator.

The Service Locator pattern, also known as the Lookup pattern [7], is a widely used mechanism for late binding in open systems. It consists of a naming service, where service providers register under a name (typically a string). Consumers of services are bootstrapped to the naming service and use such a name (obtained through bootstrapping or as a parameter) to imperatively find and utilize a service.

Similar to the designers of other frameworks for ODR applications [1,3,2], we see the Service Locator pattern as hindering reflectivity and opted for a solution pertaining to the alternative pattern, namely Inversion-of-Control [8].

Supporting domain-specific programming models, or other component frameworks, is easy and inexpensive, from an engineering perspective, because the model is small and focused just on core concerns. As cost is one of

[1] This definition makes the term "programming model" synonymous to "component model". We intentionally use the term "programming model" to indicate layering with respect to the Composition Primitives, i.e. a component model adapted to the Primitives.

the primary concerns of any engineering team, ensuring the model imposes a small cost is inherently important.

Our model supports n-order composition without imposing requirements on the component author. Complex applications take complex dependencies—such as dependencies on providers of services (other components) rather than just the services themselves. We refer to these as n-order dependencies, and we will define them in more detail later in the paper. It will be explained that it is important to facilitate discovery of these kinds of dependencies without forcing component developers into modeling for each such concern—i.e. a solution and guidance is necessary in the framework, rather than be left to component and application authors to decide upon.

The remainder of this paper is organized as follows: We provide an outline of our model to offer support for our claims in the next section. A broader comparison and contrast with related work in the area of component-oriented software as well as other technologies follows. We conclude with an outline of current applications of our model in commercial and in open-source projects that validate the applicability of our solution.

Other literature in the area of component frameworks uses mathematical formalism to introduce and prove properties of the respective component framework [4]; we believe that, for our purposes in this paper, using a well-established programming language (and its semantics) is sufficient. Given that our model targets the .NET platform, we use C$^{\#}$. For compactness and enhanced readability, we removed unnecessary (for this paper) annotations like visibility keywords ("public") or, in some places, type casts. As such, we assume basic familiarity with C$^{\#}$and .NET, but provide an appendix with an overview of less familiar features that our model relies on, such as lambda expressions, delay–compiled expressions, and the functional model of sets (features commonly known as "Language Integrated Query", or LINQ, which were introduced to the scientific community as $C\omega$ [9] and have been part of C$^{\#}$since .NET 3.5).

2 The Primitives Model

The Primitives model (see figure 1) consists of: Services, Service Definitions, Instances, Components, and Dependencies[2].

Somewhat different from other component frameworks, ours is a management model, meaning that instances of the Primitives model are separate from operational program instances, used only for composition, and can be discarded after composition occurred, thus imposing no runtime overhead once composition or re-composition completes.

[2] This ontology happens to translate into the .NET Managed Extensibility Framework (MEF) model, which builds on the Primitives, using different names: Export, ExportDefinition, ComposablePart, ComposablePartDefinition, and ImportDefinition. The paper uses the Service, Instance, Component terminology. Refer to `http://mef.codeplex.com` for more information on MEF, including the open-source implementation and samples.

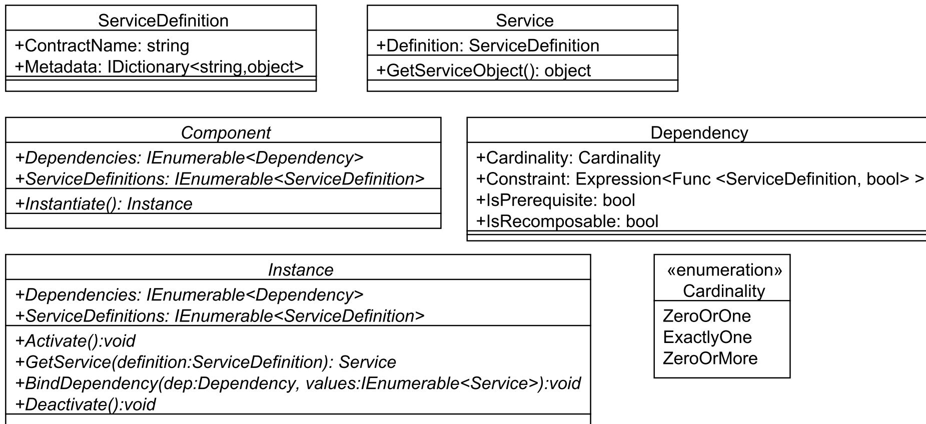


Fig. 1. Overview of the Primitives Model

A Service Definition is a semi–structured data object. It describes a kind of functionality that is offered for utilization, or consumption. It is the kind of information that one would use to decide whether a Service is interesting, without requiring the loading of code, which is a feature desirable in ODR applications.

A Service Definition is modeled as an atomic and opaque contract name (a string) and a string–keyed dictionary of objects, referred to as "metadata". The contract name identifies a document in the general sense, i.e., information, referred to as "contract", that describes the functionality offered for consumption. This description may be parameterized, in which case the contract must also describe how to encode such parameters using the metadata part of the Service Definition.

Let us consider an extensible stack-based calculator application—a perhaps contrived example, but suitable for illustrative purposes. The calculator can be extended with new operators. The contract for a particular operator may use, for the contract name, the string `Operator`, which implies the requirement that metadata contain the key `Symbol` and an associate string-typed value, such as "+".

A Service is an association between a mechanism (`GetServiceObject`) for obtaining an object satisfying a Service Definition, and the Service Definition. This is the basic building block for the rest of the model and it represents functionality ready-to-use by a consumer. Consumers may expect that the object obtained via `GetServiceObject` respect whatever prescriptions the contract requires—typically, that it implements some interface.

A Component represents a unit of reusable code. The code may be used if its dependencies are satisfied. Since dependencies may be satisfied with different values, an Instance (discussed further below) represents a particular such satisfaction which can be used independent from others.

A component advertises the set of Dependencies that all its Instances will need to have satisfied (the `Dependencies` property), and the set of statically–known Service Definitions all these Instances will offer (the `ServiceDefinitions` property).

Components produce Instances via the `Instantiate` method. Separate calls to this method must always result in different Instances being produced. Such Instances may end up offering more Services than statically advertized by a Component.

To enable composition, a given component's dependencies need to be analysed and candidate components must be determined that could satisfy these dependencies. Candidates whose own dependencies cannot be satisfied in the given composition context need to rejected. Component analysis and selection/rejection may occur without loading any code—based solely on the information provided by its `ServiceDefinitions` and `Dependencies` properties. Code loading can be deferred to the point a first request for utilizing a Service offered by an Instance is made. Compared to systems that load component code early, composition analyses and resulting selection/rejection decisions that are performed without loading component-specific code can have significant performance advantages, as validated by our use of MEF in Visual Studio 2010.

An Instance represents a set of Services and a set of Dependencies. Over time, more or less Services may be available, depending, for example, on the satisfaction of some optional Dependencies, or other runtime conditions. Mandatory Dependencies ("prerequisites") need to be satisfied before an Instance can be asked for Services.

To satisfy a Dependency, an external agent calls `BindDependency` and provides a set of Services. The agent may use `GetService` to retrieve a particular Service that is offered by the Instance.

Once all dependencies are satisfied, the Instance is ready to be used, meaning any of its offered Services may be utilized. In particular, this is the time an implementation may decide to load the actual component code and perform the actual satisfaction of dependencies.

Dynamic composition or recomposition is supported by allowing Instance dependencies be rebound on a live Instance.

Finally, a Dependency is defined through a `Constraint`. This is an Expression object applicable on a Service Definition and producing a boolean (a filter, essentially). Expressions are typed, delay–compiled functions. In our context, they are assumed to be pure. For more information, refer to Appendix A.

The key feature of using .NET expressions for the Constraint is that we can describe arbitrarily complex boolean expressions, with terms that may be arbitrary, type–safe navigations in the structure of the metadata of a Service Definition. We will discuss how this is the basis for supporting complex composition scenarios (section 2.5), as well as how this simplifies the composition process (section 2.2), when compared to known alternatives used in related work (section 3).

Other concepts exposed on a Dependency, but not explored in this paper, are: cardinality of the Dependency (`ZeroOrOne`, `ExactlyOne`, or `ZeroOrMore`); whether it is prerequisite (a generalization of the notion of constructor parameter); and whether if it can be rebound (support for dynamic scenarios).

An example of a Constraint would be the expression, using C# lambda notation:

```
(sd)=>sd.ContractName=="Operator" &&
        sd.Metadata.ContainsKey("Symbol") &&
        sd.Metadata["Symbol"]=="+"
```

This Constraint expresses a dependency on services respecting the "Operator" contract, and, in particular, on those that have a metadata key called `Symbol`.

The calculator in our example could expose itself as a component with two dependencies: one on a stack, with cardinality `ExactlyOne`, and one on a collection of Operators, with cardinality `ZeroOrMore`. The constraints of these dependencies would be as follows:

```
//stack
(sd)=>sd.ContractName=="Stack";

//any operator
(sd)=>sd.ContractName=="Operator" &&
        sd.Metadata.ContainsKey("Symbol")
```

In the case of a dependency on operators, the only statically known data is that the contract name needs to have a particular value, and that the metadata must contain the "Symbol" key—our calculator will use this for user interaction purposes, to identify each operator. In the case of a dynamic dependency (one that is satisfied based on dynamic system behavior), the Constraint should still be expressed in terms of statically–known properties of Service Definitions, and the Cardinality should always be `ZeroOrMore`. With that, it is possible to reason about composition of even dynamic dependencies—and still before loading component code.

As a final note about the Primitives, it is not necessary that Services be produced by Instances, nor that Instances be produced by Components. For example, a Service representing external functionality (e.g. a web service) may simply be instantiated by the application and subsequently considered for composition just like Services produced by Components. An implementation provided by a host that must be shared as a singleton by all hosted components may be represented as an Instance. This is also the reason that both Services and Instances carry their descriptions (i.e. Service Definitions and Dependencies)—in order to allow for analysis in the absence of a Component object.

2.1 Simple Example

We use the calculator example to illustrate the Primitives model. To avoid noise, in this example, the programming model used *is* the Primitives, however, in practice this is atypical - the Primitives are meant to be implemented by adapters to programming models, while the components would be implemented in such programming models.

Figure 2 shows the implementation of Component, and Figure 3 for the corresponding Instance for a generic operator. The example assumed a few constructors for some of the Primitives, which we have previously excluded from the model for brevity. In Figure 2, line 10, a new Service Definition is constructed with the contract name "Operator" and the metadata being a dictionary with only one key value pair, the key being "Symbol" and the value the variable `symbol`. Line 12 assumes a constructor for Dependency that constructs a constraint on a contract name ("Stack" in this case), and assumes an `ExactlyOne`

```
1  class OperatorComponent:Component{
2      internal Func<double,double,double> Op{get;set;}
3      private ServiceDefinition _offeredSvc;
4      private Dependency _dep;
5
6      OperatorComponent  (string symbol, Func<double,double,double> f){
7          Op = f;
8          _offeredSvc = new ServiceDefinition(
9                  "Operator", {"Symbol",symbol}));
10         _dep = new Dependency("Stack");
11     }
12     override IEnumerable<ServiceDefinition> ServiceDefinitions{
13             get { yield _offeredSvc; }
14     }
15     override IEnumerable<Dependency> Dependencies{
16         get { yield _dep; }
17     }
18     public override Instance Instantiate(){
19         return new OperatorInstance(this);
20     }
21 }
```

Fig. 2. Example Operator Component

```
1  class OperatorInstance : Instance
2  {
3    private OperatorComponennt _theComp;
4    private Stack<double> _stack;
5    private Service _theService;
6
7    OperatorInstance(OperatorComponent comp){
8      _theComp = comp;
9    }
10   override IEnumerable<ServiceDefinition> ServiceDefinitions{
11     get { return _theComp.ServiceDefinitions; }
12   }
13   override IEnumerable<Dependencies> Dependencies{
14     get { return _theComp.Dependencies; }
15   }
16   override void BindDependency (Dependency dep, IEnumerable<Service> values){
17       var stackExp = values.First();
18       _stack = (Stack<double>)(stackExp.GetServiceObject());
19   }
20   override Service GetService(ServiceDefinition svcDef){
21     if (svcDef.Name == "Operator"){
22       if (_theService == null){
23         Func<double> operator = () => _theComp.Op(Stack.Pop(), Stack.Pop());
24         _theService =  new Service(svcDef, () => operator);
25       }
26       return _theService;
27     }
28     return null;
29   }
30 }
```

Fig. 3. Example Operator Instance

cardinality. In Figure 3, line 24, we construct a Service object based on a given service definition, and where `GetServiceObject` delegates to the provided function (in this case, a function returning the `operator` function).

In this example, one would obtain the component for the "+" operator as follows:

```
Component plus=new OperatorComponent("+",(x,y)=>x + y)
```

As it can be seen, the "Operator" contract stipulates that the service object have the type `Func<double,double,double>`, and the "Stack" contract requires a `Stack<double>` object. This illustrates the fact that the Primitives do not impose any limitations over the kinds of types that may constitute valid operational interfaces to services. Concretely, `Func<>` is a sealed type, while `Stack<>` may be inherited from (both are part of the core .NET Framework).

Because the Service Definitions used are pure data, one can reason over a space of such Service Definitions using the Constraint of a Dependency and, without loading any component code, make determinations over feasibility of composition.

2.2 Composition in the Primitives Model

Composition in the context of the primitives consists of creating Instances out of Components and resolving their Dependencies with Services obtained out of other Instances. The composition may be controlled by an agent, generally referred to as "composition engine", external to the components involved. The composition engine is expected to satisfy a dependency `d` with a Service `s` for which the following expression evaluates to true (where `Compile` is a standard method on .NET expression objects—see Appendix A):

```
d.Constraint.Compile()(s.Definition)
```

Our goal for domain independent composition is supported by the fact that the engine need not understand the constraint in order to determine whether a service may be used to satisfy dependencies. Parsing the Constraint is possible (since it is an Expression) and may be useful for optimizations, such as indexing. It is also useful if the engine is capable of recognizing and treating specially particular kinds of contracts.

A characteristic of the Primitives design is that it has no built–in notion of identity. In particular, this allows for defining new Components from existing Components through an equivalent of the notion of partial application found in functional languages. A custom implementation of Component can be constructed that, based on an existing Component and a set of pairs (`Dependency`, `IEnumerable<Service>`), presents itself as exposing the same Service Definitions as the original component, and the same Dependencies, except for those provided in the set of pairs. Calls to Instantiate lead to the creation of instances of the original component, where the Dependencies provided in the set are hidden and pre–satisfied by the values they were associated with.

2.3 Supporting Other Component Frameworks and Domain–Specific Programming Models

We implemented an extensible chatting application[3]. Components for this application can be developed using the attributed programming model that is provided as reference implementation with .NET Framework 4.0, or as "plain old CLR" (POCO) types typically composed by Autofac[3]. In turn, these components may be composed using either the composition engine that is part of .NET Framework 4.0, or with the Autofac container. The former was designed upfront to be based on our model, while the latter was adapted to support our model.

The ChatClient.ManagedExtensibilityComponents project comprises MEF components, while the ChatClient.AutofacComponents project comprises POCO types for Autofac.

[3] `http://nblumhardt.com/archives/composition-primitives/`

The ChatClient.ManagedExtensibilityHost project uses the MEF CompositionContainer and PocoAdapter types to host both MEF and POCO components.

The ChatClient.AutofacHost project hosts the same components, but uses the Autofac.Integration.CompositionPrimitives adapters to host both MEF and POCO components in the same (Autofac) container.

2.4 Cost of Constructing Adapters

The adapter that allows Autofac components to be composed by the .NET composition engine operating on our model can be seen in the PocoAdapter project of the chatting application. The adapter consists of an implementation of the Component and Instance concepts. This adapter totals 125 lines of code, supporting our claim of low-cost adapter construction.

2.5 Modeling Higher–Order Dependencies

In our model, a dependency is taken on a Service, via a condition over the description of that Service. Typically, that Service represents a value with meaning in the application domain, and no meaning in the Primitives model. It is possible, however, that we represent a Component, for example, as a Service: after all, it provides the service of generating Instances. In this case, the value represented by the Service has a meaning in the Primitives model, which is understood and utilized by its consumer. We call this a higher–order dependency.

Scenarios where higher–order dependencies are required are apparent in complex composite applications, where some components act as generic containers for other components, managing their life–cycle and controlling access to them. An example would be an Integrated Development Environment (IDE) that can be extended with custom designers for Graphical User Interfaces (GUIs). Such designers may be usable as independent applications (i.e. they can be used as a component), however, internally, such designers may be themselves extended, and may use other components (like an editor, a canvas, etc). Hence, designers are higher-order components that also compose (and expose services of) other components.

To support such kinds of dependencies, one option would be to manufacture contracts that make the high–order dependency implicit, for example, a dependency on a component that produces an "Operator" would be encoded as a dependency on a Service respecting the contract "Operator Component". The problem with this approach is that it does not scale—the set of such contracts is a power set of the set of "simple" service contracts.

The options we discuss in what follows express higher–order dependencies through a description of structure, and require composers to understand a few contracts, one for each kind of higher–order dependency. Our goal for programming model–independent composition is still supported, however, components exposing such higher-order dependencies end up being less reusable than "simple" components. Still, any composition engine would be able to assess at least that it is unable to compose them.

In this paper, we will only illustrate support for dependencies on components.

Dependencies on Components. Consider a scenario where we want to package the wiring of a given stack with a private instance of a calculator user interface component and a private instance of each of the operator components available in some set of components. Then, we want to treat this package as a component.

Next, suppose we want to create a new Component, where each Instance would internally instantiate two Calculator aggregates and connect them to a shared instance of a Stack. The dependency of this new Component would be solely on other components: a Stack Component, a set of Operator Components, and an Aggregate Component (as defined above). It can be observed that this mechanism allows for recursive definitions of "Components".

To model such dependencies, we introduce a contract named "Component" which requires that the describing ServiceDefinition have a metadata property called "ServiceDefinitions", which should be a collection of ServiceDefinitions. Said metadata property maps to the corresponding property of a Component. A Service exposing the "Component" contract produces a Component when `GetServiceObject` is called. All that is required is that a composition engine understand the contract name "Component" and map its set of available Components into a set of Services correspondingly.

A dependency on the addition operator component we used in our examples so far would be expressed through a constraint as follows (where `Any` is the .NET LINQ operator for existential quantification over an enumerable; also, some type casts removed for brevity):

```
(sd)=>sd.ContractName=="Component" &&
        sd.Metadata.ContainsKey("ServiceDefinitions") &&
        sd.Metadata["ServiceDefinitions"] is
                IEnumerable<ServiceDefinition> &&
        sd.Metadata["ServiceDefinitions"].Any(
                s=>s.ContractName=="Operator" &&
                    s.Metadata.ContainsKey("Symbol") &&
                    s.Metadata["Language"]=="+")
```

This illustrates the rationale for using expressions for describing constraints in dependencies: they are sufficiently powerful to describe the conditions the Service Definition needs to exhibit, yet, since they represent delay–compiled code, a composer need not interpret their contents. The composer simply needs to search the expression tree for the equality test between the Service Definition contract name and the string `Component`, then compile the whole expression (via the `Compile` method expression objects expose in .NET) and evaluate the resulting function over the Service Description representation of currently available components.

3 Related Work

The area of component–based programming [6] features a large number of component models. This Section will only focus and contrast a relevant sample of those that can be used to build ODR applications. For instance, models that focus on static ahead-of-time composition, such as Koala [10], are not discussed.

The two traditional areas supported by ODR component models are extensible enterprise and rich-client applications. Representative for the enterprise area

are the Enterprise JavaBeans (EJB) framework [11], the CORBA Component Model (CCM) [1], and more recently, Dependency Injection containers. Rich-client applications use frameworks such as COM [12]. Besides these, there are several frameworks that developed out of research, such as Fractal [4]. In the following, we contrast these frameworks and our model.

A component in EJB is a bundle consisting of a lifecycle manager ("home", optional as of EJB 3.x), component code, and a deployment descriptor (additional EJB–specific information). Offered services are modeled as Java interfaces, and composition is driven by component code, by using the Java Naming and Directory Interface (JNDI) [13]—a Service Locator mechanism. Only component homes can be found and bound via JNDI, component instances are subsequently obtained via the component home.

EJB features a hybrid between Service Locator–based mechanisms and explicit dependency declaration. The home of a component "A" is registered at deployment time with a server–wide name (e.g. "Server–A"). Another component, "B", wishing to use "A", uses a relative name (e.g. "B–A") when using JNDI programatically to lookup "A". Then, the relative name and the expected home interface are listed, by the component developer of B, in the deployment descriptor. Finally, at deployment time, the application composer associates the relative name to the server–wide name.

This mechanism may be considered a way to expose component dependencies, however, nothing stops a component from attempting (and succeeding at) a lookup using a "guessed" server–wide name. As such, knowledge of an EJB component's dependencies is generally incomplete, when such knowledge is based solely on information supported by the component model. Given the introspectable nature of Java bytecode, it is possible to construct tools to extract complete inter–component dependency information (e.g. [14]), however, this is something the component model supports accidentally, and the mechanisms required may not be suitable for all applications, since they require time–consuming bytecode parsing, and since some EJB containers generate merged container/component code at deployment time.

CCM is an extension of CORBA, and aims at allowing the creation and composition of components developed on a variety of operating system platforms and using different languages. In CCM, a component features "ports", describing: (i) implemented IDL interfaces ("facets"), (ii) required implementations of IDL interfaces ("receptacles"), (iii) produced and consumed events ("sources" and "sinks"), and (iv) "attributes", which are configurable properties. Except for attributes, which are intended to be primitive types (e.g. integers, strings), the ports are expressed in terms of IDL interfaces. For example, a receptacle is expressed in terms of the interface the component wishes to consume, as well as an optional indication on whether this is rather a collection of such interfaces. Just like in our model, and in contrast to EJB, a CCM component does not rely on a Service Locator to be composed—composition is externally managed.

Components themselves are described using IDL. In contrast to our approach, IDL describes the operational interface of the consumed service, not the service

itself. For example, all calculator operations in our example would implement the same IDL interface. However, the operation that they implement would not be part of that interface, while it would be an interesting criterion for expressing dependencies. Our model provides the notions of Contract, as separate from operational interface, and Metadata for this purpose.

The closest to our Service Definition concept is the concept of WSDL [15] documents in the area of Web Services. WSDL, however, is heavily Web–oriented. Conceptually, both utilize semi–structured data to describe a service. In the same area, the Component Object Model framework (COM) [12] uses the Windows registry as basis for component discovery. The information that may be stored in the registry is similar conceptually to the notion of Service Definitions. Similarly, the enterprise-oriented COM+ uses a registration database to store component metadata.

The Vienna Framework [16] has similar inter-component framework composability goals to our effort. It takes the approach of "wrapping" components, thus imposing a constant runtime penalty. A second effect of this approach is a larger solution space, with a model trying to address the different operational interface features that are currently popular - e.g. methods, properties, events. More importantly, it does not have a model for dependencies. The metadata concept in Vienna revolves around operational contracts, while our concept of metadata models a broader kind of contracts - similar, as noted, to the Web Services notion of contracts.

A number of Dependency Injection frameworks describe components in terms of offered and required interfaces. Examples include the Castle Microkernel [2] and Autofac [3]. Both target the enterprise space, and both use a conventions–based approach to comprehending a plain .NET class as a component. For example, constructor parameters are treated as dependencies, and their type is used for matching with the types offered by other components. In turn, the interfaces implemented by a type are considered as offered services. There are policies for matching requirements expressed on collections, or inheritance–based matching. While both these frameworks use plain .NET types to model components, since the translation between a type and the internal notion of a component differs, interoperability is hard to achieve.

Outside of the enterprise space, the Fractal component model [4] was designed explicitly to permit language–specific implementations, albeit without the interoperation goal that CCM has. It relies on Service Locators to drive binding to other components, through a binding controller, which a component may expose. Binding controllers expose offered interfaces, but do not expose requirements, which is one contrasting difference to our model. Fractal models explicitly components that aggregate other components, through the concept of a content controller, which, if implemented by a component, it allows for other components to be added or removed from it.

The capability offered via content controllers is simlar to the capability, in the Primitives model, to model dependencies on components (see Section 2.5). The main difference is that the Primitives do not need a separate concept to achieve

the same result, in fact, the concept used (the Dependency model) is reusable for a variety of other kinds of dependencies, as was illustrated in section 2.

OSGi [17], the component model foundation of systems like Eclipse, addresses features similar to our model. Despite its layered architecture, and unlike our Primitives model, OSGi has a large surface area with complex protocols around aspects like deactivation and reactivation. Primitives relegate such protocols to the space of normal contracts and expects that, in the context of specific systems like MEF, such contracts are published and well-known. There is no expectation that such protocols are necessarily used across all systems build on top of Primitives. As a result, it is much easier to build implementations or adapters to and from our model. As a further point of distinction: OSGi uses LDAP query expressions to describe constraints. While LDAP is a broadly accepted directory access standard, it was not designed for the specific needs of a higher-order component model. Our model draws on a general-purpose delay-compiled and type-safe expression model (a standard part of the .NET framework) that we apply over a general metadata design, to support higher-order composition directly.

The mechanism used to express dependencies is somewhat reminiscent of the way design–by–contract languages Eiffel [18], ESC/Java [19], or SpecSharp [20] specify pre– or post–conditions by using a rich expression language describing Boolean constraints.

The concept of decoupling the concern of modeling components from the concern of composing them is found in the area of Architecture Description Languages (ADL). An ADL is a formal language that describes how components are instantiated and connected. Typically, ADLs are used for architectural validation. Interesting examples, in the context of this paper, are the Darwin ADL and ArchJava, a Java extension that can express architectural constraints.

Darwin[21,22] introduced the notion of hierarchical composition—which is similar to the kind of second order composition the Primitives support (Section 2.5).

ArchJava [23] is an ADL extension to the Java language, mainly aimed at validating communication integrity. It offers keywords for defining components with ports (`component`, `port`). A port specifies method signatures that it provides and that it requires (the `provides` and `requires` keywords) . A `connect` keyword can be used to compose components, by connecting `provides` and `requires` port elements of various components. Like Darwin, ArchJava also supports hierarchical composition through the concept of “composite components”. Unlike our model, and similar to CCM, ArchJava does not specify a notion of contract separate from operational interface contracts.

4 Conclusion

We have presented a component model for building reflective components for open and dynamic applications in .NET. Using a publicly–available sample chat application, we showed how the model supports arbitrary programming models and domain-specific component frameworks. Using this example, we showed how the development effort required to adapt such programming models or component frameworks to our model is relatively small.

A key feature of our model is the utilization of the .NET delay–compiled code model, or expressions, for describing dependencies. This offers support for higher–order dependencies, without significant complexity on the side of composition engines, and without requiring that component authors be aware of such potential consumption scenarios.

The model has current practical applications. It forms the foundation for the Managed Extensibility Framework (MEF), which is part of the .NET Framework 4.0[4]. In turn, MEF is used by Visual Studion 2010 as both an internal component model and as an external third-party extensibility mechanism. Other implementations of the primitive model have been contributed by third-parties[5], for example, an IronRuby[6] programming model[7].

References

1. Object Management Group. Corba Component Model (2002), http://www.omg.org
2. The Castle Project. Microkernel, http://www.castleproject.org
3. Nicholas Blumhardt. Autofac, http://www.autofac.org
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: An open component model and its support in java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)
5. ECMA International. Standard ECMA-335 - Common Language Infrastructure (CLI), 4 edn. (June 2006)
6. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Component Software. Addison-Wesley/ACM Press (2002)
7. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture. In: Patterns for Resource Management, vol. 3. Wiley, Chichester (2004)
8. Johnson, R.E., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming 1(2), 22–35 (1988)
9. Black, A.P. (ed.): ECOOP 2005. LNCS, vol. 3586. Springer, Heidelberg (2005)
10. van Ommering, R.: Software reuse in product populations. IEEE Transactions on Software Engineering 31(7), 537–550 (2005)
11. Sun Microsystems. Java 2^{TM} Platform Enterprise Edition Specification, v1.4 (November 2003), http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf
12. Box, D.: Essential COM. Addison-Wesley, Reading (1998)
13. Lee, R., Seligman, S.: The Jndi API Tutorial and Reference: Building Directory-Enabled Java Applications. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
14. Trofin, M., Murphy, J.: Static verification of component composition in contextual composition frameworks. International Journal on Software Tools for Technology Transfer 10(3), 247–261 (2008)

[4] For reference, the open source code is available at http://mef.codeplex.com

[5] Available at http://www.codeplex.com/MEFContrib

[6] IronRuby is an open source implementation of Ruby for .NET. More information is available at http://ironruby.net/

[7] Used in http://www.mahtweets.com/ Twitter client

15. World Wide Web Consortium (W3C). Web services description language (WSDL) v. 2.0, http://www.w3.org/TR/wsdl20
16. Oberleitner, J., Gschwind, T., Jazayeri, M.: The vienna component framework enabling composition across component models. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, pp. 25–35. IEEE Computer Society, Los Alamitos (2003)
17. OSGi Alliance. Osgi service platform – release 4, version 4.2 (2009)
18. Meyer, B.: Eiffel: The Language. Prentice-Hall, Englewood Cliffs (1992)
19. Flanagan, C., Leino, K., Lillibridge, M., Nelson, C., Saxe, J., Stata, R.: Extended Static Checking for Java. In: Proceedings of Programming Language Design and Implementation (2002)
20. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
21. Ng, K., Kramer, J.: Automated support for distributed software design. In: CASE 1995: Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, Washington, DC, USA, p. 381. IEEE Computer Society, Los Alamitos (1995)
22. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989. Springer, Heidelberg (1995)
23. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pp. 187–197. ACM, New York (2002)

A .NET Framework Features

The model relies on two features in the .NET framework, which, for the purpose of clarity, are briefly described in what follows. These features are: type–checked delay–compiled code, also known as “Lambda Expressions” (in short, “expressions” in this paper), and a monadic functional model for collections, commonly referred to as “LINQ” (Language–INtegrated Query).

An expression is an object of type `Expression<Func <...>>`, and may be constructed in two ways: either through explicit construction of an abstract syntax tree, by using typed nodes, or through language facilities, in languages that support that. For example, C#provides syntax and compile–time verification for constructing such objects. The following line of code constructs the object that represents the division computation of two integers, returning a double:

```
Expression<Func<int, int, double>>
        divisionExpression = (x,y)=>x/y;
```

Expressions can be used two ways. One way is to compile them, which results in a typed function (“delegate” in .NET nomenclature), which can then be applied:

```
Func<int, int, double> division =
    divisionExpression.Compile();

double result = division(1,2);
```

Alternatively, expressions can be passed to interpreters. LINQ–to–SQL, for example, translates an expression to SQL statement, which is then evaluated by a relational database server.

The second feature we mentioned is LINQ, which allows for the definition of lazily–enumerated collections in a functional style. For example, given a collection of integers `integers`, the following defines the subset of even numbers.

```
IEnumerable<int> evens =
    integers.Where(i=>i%2==0);
```

Besides the `Where` operator, Linq defines a large number of further standard operators over IEnumerables: functions from enumerable to enumerable. In combination, these can be used to express a full range of queries. Some languages, such as C#, provide syntactic sugar for a large subset of these operators. For example, the following form yields the same even numbers:

```
IEnumerable<int> evens =
    from i in integers where i%2 == 0 select i;
```

The rich support of expressions and enumerator operators, C#language sugar, and dynamic compilation, in combination, create a potent foundation for the constraint system of the Composition Primitives presented in this article.

Author Index

Acher, Mathieu 17
Appeltauer, Malte 50

Bergel, Alexandre 90
Binder, Walter 82
Blay–Fornarino, Mireille 90
Blumhardt, Nicholas 138
Brottier, Erwan 1

Chiba, Shigeru 122
Collet, Philippe 17
Costanza, Pascal 66

De Meuter, Wolfgang 66
D'Hondt, Theo 66

France, Robert 17

González, Sebastián 66

Haupt, Michael 50
Hirschfeld, Robert 50

Jazayeri, Mehdi 82

Kawauchi, Kazunori 50

Lahire, Philippe 17
Le Traon, Yves 1

Malenfant, Jacques 34
Masuhara, Hidehiko 50
Mens, Kim 66
Mosincat, Adina 82
Mosser, Sebastien 90

Nicolas, Bertrand 1

Rogovchenko, Olena 34

Sidorova, Natalia 106
Szyperski, Clemens 138

Takeyama, Fuminobu 122
Trofin, Mircea 138

Vallejos, Jorge 66
van der Werf, Jan Martijn 106
van Hee, Kees M. 106