Journal Subline

LNCS 4242

Transactions on Aspect-Oriented Software Development II

Awais Rashid · Mehmet Aksit
Editors-in-Chief

Springer

Awais Rashid   Mehmet Aksit (Eds.)

Transactions on Aspect-Oriented Software Development II

Springer

Volume Editors

Awais Rashid
Lancaster University
Computing Department
Lancaster LA1 4WA, UK
E-mail: awais@comp.lancs.ac.uk

Mehmet Aksit
University of Twente
Department of Computer Science
Enschede, The Netherlands
E-mail: aksit@ewi.utwente.nl

Library of Congress Control Number: 2006921902

CR Subject Classification (1998): D.2, D.3, I.6, H.4, K.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-48890-1 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-48890-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11922827 06/3142 5 4 3 2 1 0

Editorial

Welcome to the second volume of Transactions on Aspect-Oriented Software Development. The successful launch of the journal and publication of its first volume in March 2006 has been followed by a steady stream of submissions as well as several special issue proposals on topics ranging from Early Aspects and aspect interactions through to aspect-oriented programming and middleware technologies. This volume comprises two regular papers and six papers constituting a special section on AOP Systems, Software and Middleware.

The first article "On Horizontal Specification Architectures and Their Aspect-Oriented Implementations" by Aaltonen, Katara, Kurki-Suonio and Mikkonen proposes the use of horizontal architectures to improve alignment between system requirements and aspect-oriented implementations realizing those requirements. The authors' approach builds on their earlier work on the DisCO method which utilizes the notion of *superpositions* as a basis of aspect composition. The second article "A Framework for Policy Driven Auto-Adaptive Systems Using Dynamic Framed Aspects" by Greenwood and Blair synthesizes techniques such as event–condition–action rules and parameterization with dynamic AOP to develop reusable aspects that can be woven at run time. The goal of the authors' work is to support auto-adaptive behaviour using such reusable aspects. They present performance measurements as well as metrics-based evaluation of their approach in this context.

The remaining six papers focus on various topics in AOP systems, software and middleware. The guest editors, Yvonne Coady, Hans-Arno Jacobsen and Mario Südholt, provide an introduction to these in their editorial.

We wish to thank the editorial board for their continued guidance, commitment and input on the policies of the journal, the choice of special issues as well as associate-editorship of submitted articles. We also thank the guest editors, Yvonne Coady, Hans-Arno Jacobsen and Mario Südholt, for putting together the special section on AOP systems, software and middleware. Thanks are also due to the reviewers who volunteered time from their busy schedules to help realize this volume. Most importantly, we wish to thank the authors who have submitted papers to the journal so far, for their contributions maintain the high quality of Transactions on AOSD.

The journal is committed to publishing work of the highest standard on all facets of aspect-oriented software development techniques in the context of all phases of the software life cycle, from requirements and design to implementation, maintenance and evolution. The call for papers is open indefinitely and potential authors can submit papers at any time to: taosd-submission@comp.lancs.ac.uk. Detailed submission instructions are available at: http://www.springer.com/sgw/cda/frontpage/0,,3-164-2-109318-0,00.html. Two more special issues on current important topics, "Early Aspects" and "Aspects, Dependencies and Interactions", are in preparation. Calls for such special issues are publicized on relevant Internet mailing lists, Web sites as well as conferences such as the Aspect-Oriented

Software Development conference. We look forward to further high-quality submissions from prospective authors and their publication in future volumes of Transactions on AOSD.

Awais Rashid and Mehmet Aksit
Co-editors-in-chief

Organization

Editorial Board

List of Reviewers

Alessandro Garcia
Chris Gill
Andy Gokhale
Aniruddha Gokhale
Jeff Gray
Phil Greenwood
Stephan Hanenberg
William Harrison
Stephan Herrmann
Kabir Khan
Gregor Kiczales
Olaf Spinczyk
Stefan Tai
Eric Tanter
Wim Vanderperren
BartVerheecke
J. Vitek
Jon Whittle
Xiaoqing Wu
Egon Wuchner
Charles Zhang

Printing: Mercedes-Druck, Berlin
Binding: Stein+Lehmann, Berlin

Table of Contents

On Horizontal Specification Architectures and Their Aspect-Oriented Implementations

Timo Aaltonen, Mika Katara, Reino Kurki-Suonio, and Tommi Mikkonen

Institute of Software Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere, Finland
firstname.lastname@tut.fi

Abstract. In order to provide better alignment between conceptual requirements and aspect-oriented implementations, specification methods should enable the encapsulation of *behavioral abstractions* of systems. In this paper we argue that *horizontal architectures*, consisting of such behavioral abstractions, can provide better separation of concerns than conventional architectures, while supporting incremental development for more common units of modularity such as classes. We base our arguments on our experiences with the DisCo method, where behavioral abstractions are composed using the *superposition* principle, a technique closely associated with aspect orientation. Moreover, we demonstrate how the alignment between an abstract, horizontally architected specification (or model) and its aspect-oriented implementation can be achieved. Mappings are discussed that implement symmetric DisCo specifications both in Hyper/J, which enables symmetric separation of concerns, and in AspectJ that uses asymmetric structuring.

1 Introduction

Postobject programming (POP) mechanisms, like those developed in aspect-oriented programming [15, 30], provide means to modularize crosscutting concerns, which are in some sense orthogonal to conventional modularity. The background of this paper is in the observation that the same objective has been pursued also at the level of formal specifications of reactive systems, and that the results of this research are relevant for the theoretical understanding of POP-related architectures and of the associated specification and design methods.

Unlike conventional software modules, units of modularity that are suited for a structured description of the intended logical meaning of a system can be understood as aspects in the sense of aspect-oriented programming. We call such units horizontal in contrast to conventional vertical units of modularity, such as classes and processes. While the vertical dimension remains dominant because of the available implementation techniques, the horizontal dimension can provide better separation of concerns than the vertical one improving, for example, traceability of requirements and conceptual understanding of the features of the system.

A somewhat similar dualism exists in aspect-oriented specification and programming languages [18]. Asymmetric approches, such as AspectJ [48], separate between a conventionally structured base implementation and aspects that cut across the units of the base implementation, such as classes. In contrast, no such separation exists in symmetric approaches, such as Hyper/J [39], that consider systems to be composed of aspects, or slices, that potentially cut across each other.

Unfortunately, horizontal units of modularity in specification do not always align with conventional implementation techniques that rely on vertical units of architecture. However, a better alignment can be achieved with aspect-oriented implementation techniques supporting crosscutting concerns. Based on earlier work [1, 26], this paper contributes in summarizing our experiences with the DisCo method regarding horizontal architecting and the alignment between DisCo specifications and their implementations. Moreover, a mapping is discussed that implements DisCo specifications in Hyper/J and AspectJ achieving alignment between a symmetric specification and its symmetric and asymmetric implementations. Provided with the mapping, we believe that the higher-level methodology could serve well in guiding the design of aspect-oriented programs.

The rest of the paper is structured as follows. First, in Sect. 2, the idea of structuring specifications using horizontal units that capture behavioral rather than structural abstractions of the system is presented. Next, Sect. 3 discusses aligning the units of modularity at specification and implementation levels. In Sect. 4 we introduce the DisCo method, which utilizes such components as primary units of modularity. Section 5 then goes on to discuss the implementation of a DisCo specification using aspect-oriented implementation techniques. As concrete vehicles of implementation, we use Hyper/J and AspectJ. Sections 6 and 7 provide an example on preserving DisCo structures in aspect-oriented implementations. Section 8 discusses related work, and finally, Sect. 9 draws some conclusions.

2 Two Dimensions of Software Architecture

Describing architecture means construction of an abstract *model* that exhibits certain kinds of intended properties. In the following we consider *operational* models, which formalize executions as state sequences, as illustrated in Fig. 1, where all variables in the model have unique values in each state s_i. In algorithmic models, these state sequences are finite, whereas in reactive models they are nonterminating, in general.

2.1 Vertical Units

The *algorithmic meaning* of software, as formalized by Dijkstra in terms of predicate transformers [12], has the desirable property that it can be composed in a natural manner from the meanings of the components in a conventional architecture. To see what this means in terms of executions in operational models,

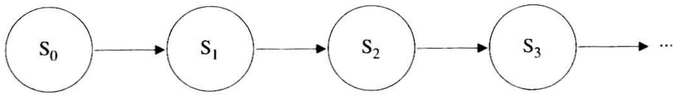

Fig. 1. Execution as a state sequence

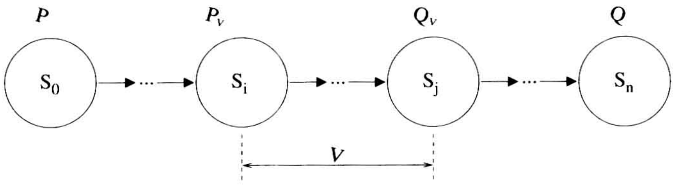

Fig. 2. A vertical slice V in an execution

consider state sequences that implement a required predicate transformation. Independently of the design principles applied, a conventional architecture imposes a "vertical" slicing on these sequences, so that each unit is responsible for certain *subsequences* of states. This is illustrated in Fig. 2, where the satisfaction of the precondition–postcondition pair (P, Q) for the whole sequence relies on the assumption that a subsequence V, generated by an architectural unit, satisfies its precondition–postcondition pair (P_V, Q_V).

More generally, an architecture that consists of conventional units imposes a nested structure of such vertical slices on each state sequence. In the generation of these sequences, the two basic operations on architectural units can be characterized as *sequential composition* and *invocation*. The former concatenates state sequences generated by component units; the latter embeds in longer sequences some state sequences that are generated by a component unit. In both cases, the resulting state sequences have *subsequences* for which the components are responsible.

In current software engineering approaches, this view has been adopted as the basis for designing behaviors of object-oriented systems, leading the focus to interface operations that are to be invoked, and to the associated local precondition–postcondition pairs. The architectural dimension represented by this kind of modularity will be called *vertical* in the following.

2.2 Horizontal Units

In contrast to precondition–postcondition pairs, the meaning of a system can also be defined by how the values of its variables behave in state sequences. In order to have modularity that is natural for such a *reactive meaning*, each component must generate state sequences, but the associated set of variables is then a subset of all variables. For each variable, the generation of its values is thus assigned to some component. An architecture of reactive units therefore imposes

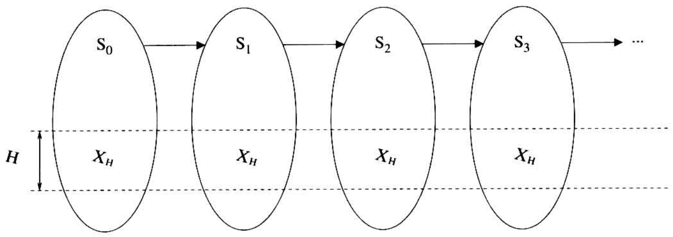

Fig. 3. A horizontal slice H in an execution

a "horizontal" slicing of state sequences, so that each unit H is responsible for some subset X_H of variables in the state sequences generated by the system, as illustrated in Fig. 3.

The two basic operations on horizontal units can now be characterized as *parallel composition* and *superposition*. In terms of state sequences, the former merges ones generated by the component units; the latter uses some state sequences that are generated by a component unit, embedding them in sequences that involve a larger set of variables. In both cases, the resulting state sequences have *projections* (on subsets of variables) for which the components are responsible.

In parallel composition of two units with variables X and Y, the units may also have a common subcomponent with variables $Z \subseteq X \cap Y$. In this case their composition has the property that projecting the resulting state sequences on Z yields state sequences generated by the common subunit. This is, of course, essential for understanding such a subcomponent as a component within the resulting *horizontal architecture.*

Since the variables that are associated with a horizontal unit may contain variables for which different vertical modules would be responsible, the properties of a horizontal unit emphasize collaboration between vertical units and relationships between their internal states. The two dimensions of architecture are in some sense dual to each other. On the one hand, from the viewpoint of vertical architecture, the behaviors generated by horizontal units represent crosscutting concerns. From the horizontal viewpoint, on the other hand, vertical units emerge incrementally when the horizontal units are constructed and put together.

Superposition is typically used for refining specifications in a stepwise manner. Even independent refinement steps are then taken in some order, which results in irrelevant dependencies between the associated horizontal units. If such dependencies are to be avoided, different branches of the specification should be used whenever possible, i.e., superposing units on a common subcomponent. Such branches typically address different concerns of the same system that are composed using parallel composition at a later stage of the specification process. It should be noted, however, that parallel composition is provided for purposes of convenience, the same behavior could be specified by using only superposition.

In that case, different branches of the specification are seen to define a partial order in which superposition steps should be applied when composing the units (see discussion in [25]).

2.3 Architecting Horizontal Abstractions

To illustrate the nature of horizontal units, consider a simple modeling example of an idealized doctors' office, where ill patients are healed by doctors.[1] The natural vertical units in such a model would model patients, doctors, and receptionists. Horizontal units, on the other hand, would model their cooperation as specific projections of the total system, and the whole specification could be built incrementally from these.

The specification process can start with a trivial model of the simple aspect that people get ill, and ill patients eventually get well. The "illness bits" of the patients are the only variables that are needed in this horizontal unit. Next, this unit can be embedded in a larger model where a patient gets well only when healed by a doctor. This extended model has events where a doctor starts inspecting a patient, and participation of a doctor is added to the events where a patient gets well. Finally, a further superposition step can add the aspect that also receptionists are needed in the model, to organize patients to meet doctors, and to make sure that they pay their bills. This aspect is truly cross-cutting in the sense that it affects all the vertical units, i.e., patients, doctors, and receptionists.

Each unit in this kind of a horizontal architecture is an *abstraction of the meaning* of the total system. The first horizontal unit in this example is an abstraction where all other behavioral properties have been abstracted away except those that concern the "illness bits" of patients. In terms of temporal logic of actions (TLA) [35], (the meaning of) the total system always implies (the meaning of) each horizontal unit in it. As for variables, each component in the horizontal structure focuses on some variables that will become embedded in the vertical components in an eventual implementation. This can be related with the observation of Parnas, Clements, and Weiss in [41], where such embedded "secrets" are considered more important than the interfaces associated with them in early phases of design.

This gives a formal basis for specifying a reactive system—i.e., for expressing its intended meaning—incrementally in terms of operational abstractions that can be formally reasoned about. Since it is unrealistic to formulate any complex specification in one piece, this is a major advantage for using horizontal architectures in the specification process. A classical example of using horizontal slices is the separation of correctness and termination detection in a distributed computation, with nodes forming the associated vertical structure [13]. This is also the earliest known use of superposition in the literature—its close relationship with aspect orientation was first reported in [28].

[1] This is an outline of a simplified version of an example that was used to illustrate the ideas of DisCo in [31].