Lecture Notes in Computer Science 4640

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

Awais Rashid Mehmet Aksit (Eds.)

Transactions on Aspect-Oriented Software Development IV

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: 2007939971

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 1861-3027
ISBN-10 3-540-77041-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-77041-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12197705 06/3180 5 4 3 2 1 0

Editorial

Volume IV of Transactions on Aspect-Oriented Software Development continues the special issue on Early Aspects from volume III. The special issue was guest edited by João Araújo and Elisa Baniassad and handled by one of the co-editors-in-chief, Mehmet Aksit. The papers in volume III discussed topics pertaining to analysis, visualisation, conflict identification and composition of Early Aspects. The papers in this volume focus on mapping of Early Aspects across the software lifecycle. Complementing this focus on aspect mapping is a special section on Aspects and Software Evolution guest edited by Walter Cazzola, Shigeru Chiba and Gunter Saake—the co-editor-in-chief handling this issue was Awais Rashid.

We wish to thank the guest editors for their commitment and effort in producing such a high quality volume. We also 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. Thanks are also due to the reviewers who volunteered time amidst 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.

There are two other special issues on the horizon. One focuses on *aspects, dependencies and interactions* (guest editors: Ruzanna Chitchyan, Johan Fabry, Shmuel Katz and Arend Rensink) for which the call for papers has closed and the papers are currently in the review phase. There is an open call for papers for a special issue on *aspects and model-driven engineering* (guest editors: Jean-Marc Jezequel and Robert France). The call will close on 15 November 2007. These special issues coupled with the regular submissions to the journal mean that we have a number of exciting papers to look forward to in future volumes of Transactions on AOSD.

There are also important changes afoot at the journal. At the last editorial board meeting Don Batory and Dave Thomas volunteered to step down from the editorial board. Their input and guidance were invaluable in the start-up period of the journal. Don was also a very active and conscientious associate editor. We thank them both for their contributions. At the same time, we welcome Bill Harrison, Oege de Moor and Shriram Krishnamurthi to the editorial board and look forward to working with them.

Another major change involves the co-editors-in-chief. As per the journal policy, one of the founding co-editors-in-chief, Mehmet Aksit, is stepping down after the first two years of the journal. So this is the last volume Mehmet will be co-editing in this role. Needless to say, Mehmet has been instrumental in the successful launch of the journal and its operations to date and the editorial board is most grateful for his efforts and contributions. We do not lose Mehmet although as he will remain on the editorial board and continue to guide us.

At the same time, it is with great pleasure we welcome Harold Ossher who will be taking over from Mehmet as co-editor-in-chief. Harold's name needs no introduction in the AOSD and software engineering communities. His work on subject-oriented

programming laid the early foundations of AOSD and, subsequently, his work on multi-dimensional separation of concerns has been fundamental in influencing how we perceive the notion of aspects. The journal will continue to flourish under his guidance and leadership and we feel that the future for both the journal and the AOSD community at large is very bright.

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

Organization

Editorial Board

List of Reviewers

Jonathan Aldrich
João Araújo
Don Batory
Klaas van den Berg
Lodewijk Bergmans
Jean Bézivin
Gordon Blair
Johan Brichau
Shigeru Chiba
Ruzanna Chitchyan
Paul Clements
Yvonne Coady
Arie van Deursen
Erik Ernst
Robert Filman
Lidia Fuentes
Alessandro Garcia
Sudipto Ghosh
Jeff Gray

Michael Haupt
Bill Harrison
Robert Hirschfeld
Jean-Marc Jézéquel
Gregor Kiczales
Günter Kniesel
Thomas Ledoux
Cristina Lopes
Roberto Lopez-Herrejon
David Lorenz
Hidehiko Masuhara
Ana Moreira
Klaus Ostermann
Martin Robillard
Americo Sampaio
Christa Schwannninger
Dominik Stein
Mario Südholt
Eric Tanter
Gabriele Tänzer
Peri Tarr
Bedir Tekinerdogan
Emiliano Tramontana
Ian Welch
Jon Whittle

Table of Contents

Focus: Early Aspects – Mapping Across the Lifecycle

Focus: Aspects and Software Evolution

Guest Editors' Introduction: Early Aspects — Mapping Across the Lifecycle

João Araújo[1] and Elisa Baniassad[2]

[1] Universidade Nova de Lisboa, Portugal
ja@di.fct.unl.pt
[2] Chinese University of Hong Kong, China
elisa@cse.cuhk.edu.hk

Early Aspects are aspects found in the early life-cycle phases of software development, including requirements elicitation and analysis, domain analysis and architecture design activities. Aspects at these stages crosscut the modular units appropriate for their lifecycle activity; traditional requirements documentation, domain knowledge capture and architectural artifacts do not afford separate description of early aspects. As such, early aspects necessitate new modularizations to be effectively captured and maintained. Without new tools and techniques, early aspects remain tangled and scattered in life-cycle artifacts, and may lead to development, maintenance and evolution difficulties.

Overview of the Articles and the Evaluation Process: This special issue consists of eight articles, selected out of ten submissions. Each were evaluated by three reviewers and revised at least twice over a period of 7 months.

The Early Aspects special issue covers three main areas of research, and is split over two volumes of the journal. The papers in vol. III focused on Analysis and Visualization, and Conflicts and Composition. This volume contains papers on mapping early aspects throughout the life-cycle.

Mapping

The relationship between aspects between life-cycle phases is of primary interest to the Early Aspects community. In this work, researchers attempt to draw a correspondence between concerns in one lifecycle phase, to those found in another. These approaches may involve link recovery, in which existing artifacts are examined and the links between them derived, link formation, in which aspects in each phase are captured in such a way that promotes traceability between them, or link exploitation, in which traceability links are made explicit, and then exploited for other purposes. Here we present two papers related to mapping between aspects at life-cycle phases.

COMPASS: Composition-Centric Mapping of Aspectual Requirements to Architecture by *Ruzanna Chitchyan, Mónica Pinto, Awais Rashid and Lidia Fuentes*

This paper presents COMPASS, an approach that offers a systematic means to derive an aspect-oriented architecture from a given aspect-oriented requirements specification. COMPASS provides an aspect-oriented requirements description

A. Rashid and M. Aksit (Eds.): Transactions on AOSD IV, LNCS 4640, pp. 1–2, 2007.

language (RDL) that enriches the informal natural language requirements with additional compositional information. COMPASS also offers an aspect-oriented architecture description language (AO-ADL) that uses components and connectors as the basic structural elements with aspects treated as specific types of components.

Aspects at the Right Time by *Pablo Sánchez, Lidia Fuentes, Andrew Jackson and Siobhán Clarke*

This paper describes an aspect mapping from requirements (specified in Theme/Doc) to architecture (specified in CAM) to design (specified in Theme/UML). The mapping includes heuristics to guide to the right time to specify the right aspect properties. Moreover, it allows aspect decisions captured at each stage to be refined at later stages as appropriate. Also, they provide a means to record decisions that capture the alternatives considered and the decision justification, crucial for managing aspect evolution at the right time.

COMPASS: Composition-Centric Mapping of Aspectual Requirements to Architecture

Ruzanna Chitchyan[1], Mónica Pinto[2], Awais Rashid[1], and Lidia Fuentes[2]

[1] Computing Department, Lancaster University, Lancaster LA1 4WA, UK
{rouza, marash}@comp.lancs.ac.uk
[2] Dept. Lenguajes y Ciencias de la Computación, University of Málaga, Málaga, Spain
{pinto, lff}@lcc.uma.es

Abstract. Currently there are several approaches available for aspect-oriented requirements engineering and architecture design. However, the relationship between aspectual requirements and architectural aspects is poorly understood. This is because aspect-oriented requirements engineering approaches normally extend existing requirements engineering techniques. Although this provides backward compatibility, the composition semantics of the aspect-oriented extension are limited by those of the approaches being extended. Consequently, there is limited or no knowledge about how requirements-level aspects and their compositions map on to architecture-level aspects and architectural composition. In this paper, we present COMPASS, an approach that offers a systematic means to derive an aspect-oriented architecture from a given aspect-oriented requirements specification. COMPASS is centred on an aspect-oriented requirements description language (RDL) that enriches the usual informal natural language requirements with additional compositional information derived from the semantics of the natural language descriptions themselves. COMPASS also offers an aspect-oriented architecture description language (AO-ADL) that uses components and connectors as the basic structural elements (similar to traditional ADLs) with aspects treated as specific types of components. Lastly, COMPASS provides a set of concrete mapping guidelines, derived from a detailed case study, based on mapping patterns of compositions and dependencies in the RDL to patterns of compositions and dependencies in the AO-ADL. The mapping patterns are supported via a structural mapping of the RDL and AO-ADL meta-models.

Keywords: aspect–oriented software development, early aspects, requirements engineering, architecture design, requirements to architecture mapping, requirements composition, architecture composition.

1 Introduction

As aspect-oriented software development (AOSD) grows in popularity, more and more requirements [9, 45, 54, 69] and architecture [9, 67] level approaches emerge. They all aim to improve modular representation and analysis of crosscutting concerns at the requirements- or architecture-level, but no single approach covers both activities: starting from requirements and resulting in an architecture specification for

A. Rashid and M. Aksit (Eds.): Transactions on AOSD IV, LNCS 4640, pp. 3 – 53, 2007.

the given requirements. Though some approaches, e.g., [45, 54], provide initial insights into architectural choices, no concrete mapping guidelines for deriving the architecture are provided. Our approach, COMPASS, is based on a composition-centric perspective for such requirements-to-architecture mapping. That is, it focuses on the compositional information and dependencies of concerns at the requirements-level and utilises these as a basis for a systematic transition from an aspect-oriented requirements specification to an aspect-oriented architecture. Compositions are the embodiments of aspectual interactions in requirements. The mapping facilitated by COMPASS allows a developer to utilise requirement compositions to pinpoint the likely aspectual relationships in architecture that originate from the requirements.[1]

Such a composition-centric approach requires rich composition semantics at the requirements-level. However, the majority of current aspect-oriented requirements engineering (AORE) techniques have been developed as extensions to other contemporary requirements engineering (RE) approaches. For instance, the AORE with ARCADE approach [54] extends a viewpoint-based requirements engineering model called PREView [64] with the notion of aspects. Similarly, the aspect-oriented use case approach [31] extends the traditional use case model with aspectual use cases. Although this provides backward compatibility in terms of software processes and development practices, it also restricts these AO approaches to the same dominant decomposition as the extended RE approach, turning everything that does not fit quite well with the base[2] approach into aspects. The semantics of such concerns put into this "aspect-bin" are often under-investigated; they frequently do not receive adequate representation and reasoning support either. Though some of these concerns may very well align with the given notations (often adopted from the base approach, or new dedicated "add-ons") and classification, others may be forced into such adapted frameworks. For instance, in case of aspectual use cases [31] the *extend* and *insert* use cases are re-classified as "aspectual" and an additional set of *infrastructure* use cases is introduced for the representation of non-functional concerns. Although the *extend* and *insert* use cases fit very well into the traditional use case (i.e., functionality-related) semantics and representation, the *infrastructure* use cases are forced to "functionalise" the qualitative semantics of non-functional concerns. As such the expressive and compositional power of the aspect-oriented approach is limited by that of the base approach.

The provision of richer composition semantics at the requirements-level is the first aim of COMPASS. The COMPASS Requirements Description Language (RDL) partitions requirements into concerns like most RE techniques but with two main differences. First, it takes a symmetric approach to such partitioning, i.e., aspects and base concerns are treated uniformly using the same abstraction, a *concern*. [46, 66]. Second, it enriches the usual informal natural language requirements with additional compositional information derived from the semantics of the natural language descriptions themselves. This compositional information is utilised for semantics-based composition of requirements-level concerns. It also provides core insights into

[1] It must be noted that other aspects, motivated by the solution domain, may also arise in architecture. Such solution domain aspects are not targeted by this approach. Our compositions pinpoint the aspects arising from the problem domain, i.e., the requirements.

[2] "Base approach" here is the approach being extended with Aspects.

the intentionality of a requirement hence facilitating a clearer mapping to relevant architectural elements. The natural language requirements' annotation with the RDL is fully automated via our Wmatrix [58] natural language processor. Tool support is also available for crosscutting concern identification [56, 57].

A composition-centric approach also requires clearer architectural composition semantics for aspects. Presently in many cases, aspect-oriented architecture design approaches adopt concepts introduced by aspect-oriented programming (AOP) languages, without questioning how appropriate these may be at the architecture level. Some of the examples of such programming language driven features are: introductions; asymmetric representation — i.e., use of different artefacts for base functionality and aspectual behaviour; and the lack of separation of compositional information (i.e., the pointcuts) from the aspect behaviour (i.e., the advice). Although such features provide a closer alignment between architecture and a given AOP language, they do not always help to capture the fundamental nature of software architecture descriptions, unnecessarily complicating architecture comprehensibility and evolution. For instance, AOP `introductions` are implementation-specific mechanisms thought to extend the interface and behaviour of a class when only the binary code is available. This is not appropriate at the architecture level, where instead the interface of a component should be extended by transforming the component into a composite component with multiple interfaces. Also, pointcuts specify composition of architectural components, be it aspectual ones, and, therefore, ought to be part of the connector semantics rather than be included within the aspect specification.

The provision of suitable abstraction and composition mechanisms at the architecture-level is the second aim of COMPASS. We propose an aspect-oriented ADL (AO-ADL) based on a symmetric decomposition model — it uses components and connectors as the basic structural elements (similar to traditional ADLs) with aspects treated as specific types of components. Connectors are enriched with additional composition semantics to cope with the crosscutting effect of aspectual components.

Having enriched requirements and architecture models with suitable aspect composition semantics, COMPASS provides a set of concrete mapping guidelines, derived from a detailed case study, based on mapping patterns of compositions and dependencies in the RDL to patterns of compositions and dependencies in the AO-ADL. The mapping patterns are supported via a structural mapping of the RDL and AO-ADL meta-models.

The mapping guidelines in COMPASS are a significant contribution not only to improving transparency of transition from aspectual requirements to architecture but also to the general issue of relating requirements to architecture in a systematic manner. The third goal of COMPASS is to establish clear links between the requirements-level aspects and their compositions with architecture elements and transition from the requirements-level to architecture. The architecture derived from COMPASS mappings acts as a starting point for refinement and elaboration into an architectural solution. We leave the topics of architecture refinement and elaboration out of this paper for a separate discussion, and focus on the actual mappings themselves.

The rest of the paper is structured as follows. Section 2 presents how COMPASS fits within the software development activities. Section 3 discusses our RDL and its composition semantics. Section 4 discusses the abstraction and composition mechanisms in our AO-ADL. Section 5 presents the mapping patterns based on the compositional information and dependencies as well as the structural mapping between the RDL and AO-ADL meta-models. The discussion in Sects. 3,4 and 5 is based on a concrete case study of an online auction system that has also been used as a basis for eliciting the mapping guidelines. Section 6 discusses and demonstrates application of guidelines, as well as presents some difficult issues that the COMPASS approach faces. Section 7 discusses related work and Sect. 8 concludes the paper. The summary of the mapping guidelines is presented in Appendix A.

2 COMPASS Within the Software Development Process

In order to explain the relation of COMPASS to the general software development activities, we have highlighted the COMPASS activities in Fig. 1. As shown, COMPASS is concerned with the link between the RE and Architectural activities, which is represented as the oval containing "Requirements to Architecture Mapping" activity along with its adjacent arrows.

Figure 1 explicitly mentions a number of RE activities that come before the COMPASS activities, as well as a number of Architecture Design activities that come after. Though none of these pre and post COMPASS activities are the focus of this paper, for the sake of completeness and clarity, we provide a brief overview of some related issues in this section.

2.1 Requirements Engineering

For use of COMPASS we do not prescribe any particular RE technique, the only specified condition being that the used technique will contain natural language requirements specification. The decision to use natural language specification was motivated mainly by the fact that the majority of the requirements are still specified in natural language text [12]. Clearly, it is hardly possible to establish a direct mapping from raw natural language text to architecture design. Thus, a number of tasks dedicated to concern identification and structuring requirements into a specification document need to be undertaken. The specification document will then form input for the COMPASS mapping.

Thus, any kind of structured text-based specification (such as viewpoints or use cases) can form COMPASS input.[3] Our own work on producing such structured natural language requirements for AORE from initial natural language (NL) text is presented in [56, 57]. In brief, our approach for structuring is based on use of tool-supported corpus-based natural language processing methodology. We initially apply an NL processor (called WMATRIX [6, 58]), which helps to identify the main topics of interest in the given natural language document by comparing the given document

[3] Input for COMPASS can be produced by any other RE approach that uses natural language. It is not necessary to use specifically our approach that is discussed further in this section.

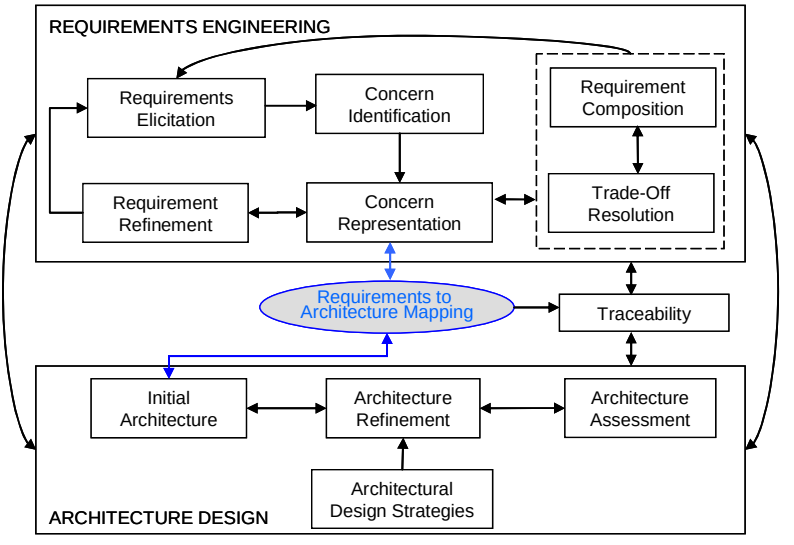


Fig. 1. COMPASS within the Software Development activities

against the British National Corpus [4], and pinpointing statistically significant words in the text, thus assisting in the Requirements Elicitation Task (note, here we imply elicitation from the input documents).

This information and the related annotation is used by EA-Miner [56, 57]— a requirements structuring and early aspects identification tool to help structure the text from the input documents into a specific requirements model (e.g., viewpoints, use cases, aspectual concerns). EA-Miner does not completely automate this structuring process, but assists the requirements engineer in realising it by providing the significant topics of the text for candidate concerns (e.g., selected nouns could become viewpoints in a viewpoints based model), as well as eliminating repetition, grouping related requirements for a concern, visualising the relations between various concerns, as well as producing the eventual structured requirements document to be used for COMPASS. Thus, EA-Miner addresses the topics of Concern Identification, Representation, and (partially) Refinement (as the stakeholders may be asked to clarify some issues when structuring documents, and the requirements engineer's domain knowledge can also be used). Besides, EA-Miner assists with initial crosscutting relationship identification by identifying the items that belong to dedicated non-functional requirements (NFR) lexicons (e.g., security) in text linking the given NFR to the requirement containing that item. For the functionality-related crosscutting identification a metric-based estimation is used. The requirements analyst then chooses which of these suggested crosscutting relationships merit further attention and defines compositions for them. Also, any relevant broad relationships between requirements may be specified as compositions, if deemed necessary by the requirements engineer.

In addition, a tool (called MRAT [68]) was also developed to support requirements composition and some trade-off analysis using the extended Wmatrix annotations,

thus addressing the respective Composition and (partially) Trade-off Resolution tasks in Fig. 1.

2.2 Architecture Design

The COMPASS mapping and guidelines (discussed later in this paper) are applied to the above discussed structured requirements to result in an *Initial AO Architecture* outline. This initial architecture outline is by no means to be considered the definitive architecture for the system under development. On the contrary, it is only the beginning of architecture design and will undergo a number of iterations based on *Architecture Assessment*, as well as horizontal and vertical *Architecture Refinement* before coming close to the definitive architecture design. In fact, as shown by the double directional arrows between the mapping and initial architecture derivation activities in Fig. 1, this is an iterative process. This is in line with Nuseibeh's Twin Peaks model [46] where architecture derivation and requirements engineering activities are carried out in parallel. During the *Refinement* process, several *Architectural Design Strategies* may help a software architect to choose the best architecture that fulfils not only the quality attributes demanded by requirements but also the evolvability and reusability properties of the architecture itself. Non-aspect-oriented and aspect-oriented architectural patterns and styles, as well as the software architect's domain expertise can be used to evolve the initial architecture [1, 19, 59]. In addition, some requirements can not be completely realised at the architecture level. Thus, their realisation should be postponed to detailed design and even to implementation. Such requirements need to be recorded, along with other decisions taken by the software architect and passed to the designers and implementers [13].

Having briefly placed COMPASS into the framework of the general development activities in this section, we henceforward focus only on the relevant elements of COMPASS itself.

3 Requirements Description Language (RDL)

Our RDL, detailed in [15], is based on the observation that the natural language used for expressing requirements already reveals semantics that can be used as a basis for composition (and subsequent analysis).[4]. Consequently, rather than searching for *external* ways of discovering composition semantics (e.g., by observing dependencies through use of requirements), we look *into* the requirements themselves to make their *intrinsic* semantics and relationships more explicit. For this, we utilise the vast body of work on natural language processing (NLP) [22, 40, 55] that has studied the meaning of words and their link to the roles these words play within natural language statements.

Several prominent works in linguistics [22, 40, 55] have shown that there is a clear link between the meaning of the words and their grammatical behaviour. Dixon [22] suggests that the varying grammatical behaviour of verbs is the result of the

[4] Using RDL composition features, we can detect conflicts and inconsistencies between requirements, and take early corrective actions. However, the analysis of compositions for conflict identification and resolution is not in the scope of this paper.

differences in their meaning. Thus, using this principle, he groups verbs into several semantic categories. In our RDL, we use the semantic categories of Dixon as a basis for identifying the types of relationships between concerns and, eventually, deriving composition operators. A detailed discussion of Dixon's classification and its adaptation for use within the RDL is presented in [14, 15, 17].

Another area that the RDL draws upon is natural language grammar [52] that defines a set of grammatical functions (i.e., subject, verb phrase and object) which reflect the main actors and activities in a given sentence. We consider the elements that realise these grammatical functions to be very important in conveying requirement semantics.

Thus, the RDL[5] is a language designed to capture the meaning of natural language requirements via the elements that realise certain grammatical functions (i.e., subject, verb phrase and object) and relate to certain semantic categories (e.g., verb classes). This language allows description of requirements interrelationships (which are utilised in compositions) based on their semantics [15]. The language is represented via an XML-schema. The XML annotations of requirements with the RDL elements are supported via the Wmatrix semantic tagger. Composition specification and manipulation is supported via MRAT [15, 68] — an associated requirements specification and composition tool.

3.1 The RDL Meta-model

The meta-model of the RDL is shown in Fig. 2. As mentioned earlier and shown in Fig. 2, the RDL is based on a symmetric model of concerns, i.e., all concerns (whether aspects or otherwise) are represented using the same abstraction. This is natural since there is no conceptual difference between these concerns or their requirements; they only differ in the way they interact with each other. Consequently,

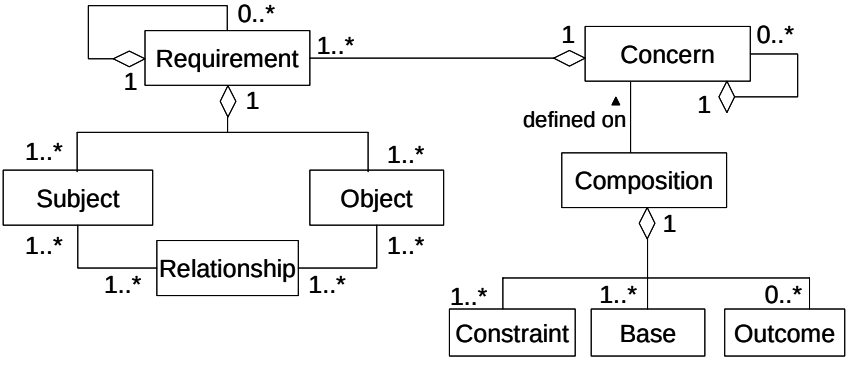


Fig. 2. RDL Meta-model

[5] Please note the distinction between language design and implementation [21]. Here, the RDL design is based on the natural language grammar and semantics, while its implementation is supported via XML schema and NLP annotation tools. Here, we focus on the design perspective.

we provide compositions to support first-class treatment of the mutual influences of different types of concerns (e.g., functional and non-functional concerns [44, 45]), in contrast to having the analysis driven mainly by a single concern type (e.g., by the qualitative non-functional properties).

A **Concern** is a module for encapsulating requirements pertaining to a specific matter of interest (e.g., selling and account management). A concern can be simple (containing only requirements), or composite (containing other concerns as well as requirements). Each concern is identified by its name. Figure 3 illustrates a partial description of the `Auction Management` concern. Note that we will use the example description in Fig. 3 for discussing other elements of the RDL meta-model as well.

A **Requirement** is a description of a service the stakeholders expect of the system, the system behaviour, and constraints and standards that it should meet [63]. One or more requirement elements are encapsulated within a concern (as shown in Fig. 3 with <Requirement> </Requirement> tags) and each requirement is identified by a unique identifier within the concern scope. Similar to concerns, requirements too can have sub-requirements.

3.2 Semantic Elements of a Requirement

The requirements specified using the RDL are annotated natural language sentences. Each requirement may contain one or several *clauses*[6] [52]. Each clause contains sub-elements for *subject*, *relationship* and optionally for *object(s)*.

Each clause has one subject and its related relationship(s), though these elements can be complex. For instance, in the clause "`Buyers and sellers can bid, buy, sell and negotiate over the auction system`" the subject is "`buyers and sellers`" and the relationship includes the "`bid, buy, sell, negotiate`" verbs. Nevertheless, in the following discussion we assume a simple clause structure, which can always be obtained from a complex one. In the above example the simple clause can be obtained by separating the enumerated subjects and relationships into one-to-one clauses, e.g., "`Buyers can bid over the auction system`"; "`Buyers can buy over the auction system`", etc. In the following discussion we further simplify assuming that a requirement consists of one sentence[7] and each sentence contains a clause with one subject and one relationship.

A **subject** is the entity that undertakes actions described within the *requirement* clause. Subject in our RDL corresponds to the *grammatical subject* in the clause. In order to support composition specifications involving a subject denoted with different words representing the same semantics, a set of synonymous definitions must be provided. These synonyms could be provided either through a standard synonym dictionary or per project through project specific dictionaries. For instance, in Fig. 3 the subject that undertakes the actions in the requirement with id 1 is `customer`. However, any reference to a customer (generally) will also be relevant to the `user`,

[6] As per standard definition, a simple sentence contains a single clause, while a composite one may have more than one clause [52].

[7] This simplification is acceptable as further sentences of a requirement, if needed, can be identified as sub-elements of the complete requirement. See, for example, requirements 1.1–5 in Fig. 3 which decompose a complex clause into simpler ones.

`buyer`, `bidder`, and `seller` subjects within the auction system. In cases where projects have project-specific ontology this richer information is invaluable for further analysis.

An **object** is the entity which is being affected by the actions undertaken by the subject of the *requirement* sentence, or in respect with which the actions are undertaken. Object in our RDL corresponds to the *grammatical object* in the clause. Usage and properties of an Object are very similar to those of the Subject. However, in a requirement there could be several objects associated with (affected by) a single subject. For instance, in requirement 1 of Fig. 3, the object affected (i.e., initiated) by customer is the `auction`. In this case the `auction` is the *direct object*, i.e., it follows the verb with no prepositions before it. On the other hand, in sub-requirement with id 1.1 `goods` is an indirect object, as it has the *of* preposition before it. There could be only one direct object in a clause and more than one indirect objects. The issue as to which indirect objects are relevant for RDL annotation in a particular clause depends on the type of verb (also called *relationship* in the RDL) of the clause.

```
<Concern name="Auction Management">
    <Requirement id="1"> A
        <Subject> customer </Subject> that wishes to
        <Relationship type="Move" semantics="Transfer_Posession">sell</Relationship>
        <Relationship type="Move" semantics="Set_in_Motion">initiates</Relationship> an
        <Object> auction </Object> by
        <Relationship type="Communicate" semantics="Inform"> informing </Relationship> the
        <Object> system </Object> :
        <Requirement id="1.1"> The
            <Subject> customer </Subject>
            <Relationship type="Communicate" semantics="Inform">informs</Relationship> of the
             <Object> goods </Object> to auction.</Requirement>
        <Requirement id="1.2"> The
            <Subject> customer </Subject>
            <Relationship type="Communicate" semantics="Inform">informs</Relationship> of the
             <Object> minimum bid price </Object> of goods to auction.</Requirement>
        <Requirement id="1.3"> The
            <Subject> customer </Subject>
            <Relationship type="Communicate" semantics="Inform"> informs </Relationship> of
            <Object> reserve price </Object> for the goods to auction. </Requirement>
        <Requirement id="1.4"> The
            <Subject> customer </Subject>
            <Relationship type="Communicate" semantics="Inform">informs</Relationship> of the
            <Object> start period </Object> of the auction. </Requirement>
        <Requirement id="1.5"> The
            <Subject> customer </Subject>
            <Relationship type="Communicate" semantics="Inform">informs</Relationship> of
            <Object> duration </Object> of the auction, e.g., 30 days. </Requirement>
    </Requirement>

    <Requirement id="2">The
            <Subject> seller </Subject> has the right to
            <Relationship type="Move" semantics="Set_in_Motion"> cancel</Relationship>  the
            <Object> auction </Object> as long as the auction's
            <Subject> start date </Subject> has not been
            <Relationship type="Move" semantics="Set_in_Motion"> passed</Relationship>, i.e., the
            <Subject> auction</Subject> has not already
            <Relationship type="Move" semantics="Set_in_Motion"> started </Relationship>.
    </Requirement>
</Concern>
```

Fig. 3. Example of RDL Use

Relationship depicts the action performed (state expressed) by the *subject* on or with regards to its *object(s).* Relationships can be expressed by any of the verbs or verb phrases[8] in natural language. Using Dixon's verb categories [22], we classify relationships into a set of types (the *type* attribute of the `<Relationship>` element in Fig. 3) and their more specific sub-types (the *semantics* attribute of `<Relationship>`). For instance, the `inform` verb in Fig. 3 is annotated as being of the *Communicate* type with *Inform* semantics. The various relationship categories derived from Dixon's verb classification are detailed in [14, 15].

It must be noted, that we do not suggest that ALL semantics of a requirement are reduced to subject-relationship-object constructs (SRO). Indeed, elsewhere we are looking at such element of requirements as degree of importance (i.e., which requirements are more urgent compared to others) or quality satisfaction, among others. However, we suggest that SROs are the main elements with respect to which the rest of the requirement semantics are formulated. Thus, SROs are the elements which participate in relations with other requirements, and are qualified, constrained or otherwise defined by both single requirement semantics, and the inter-requirements dependencies. Such semantics and dependencies can be defined in the RDL compositions, as discussed below and detailed in [15].

3.3 Semantics-Based Composition of Concerns

A composition rule in the RDL comprises three elements: Constraint, Base and Outcome. Each of these elements has an operator attribute and a semantic query expression. The query expression is enclosed within the element tags (e.g., `<Constraint>` query expression `</Constraint>`) and can select whole concerns or individual requirements from within concerns. A concern is selected if the *concern* keyword is used in the query, otherwise requirements are selected.

These query expressions are called *semantic* queries, since they select concerns/requirements on the basis of the semantics of (parts of) these concerns or requirements. For instance, the query expression (`subject="seller" and relationship="end" and object="auction"`) on lines 2 and 3 in Fig. 4 specifies that requirements that either fully or partially relate to the seller ending an auction should be selected. The queries can use all kinds of annotations provided by the RDL, including the SRO, verb types and semantics (e.g., `relationship.type= "Modify"`), concern names, or even requirement ids (though the last option reverts back to syntactic matching and is discouraged). It should be noted that a requirement may have several sentences, but if one clause of one of its sentences matches the specified semantics, the requirement will be relevant for this query. Benefits of the semantic queries are twofold: first, we avoid syntactic matching in the composition specifications, thus avoiding unintended element matching. Instead, compositions are specified based on the semantics of the requirements. Second, it ensures that requirements compositions are semantically justified, rather than arbitrarily provided by a requirements analyst.

[8] We underline that Relationships can be expressed not only by verbs (e.g., enrol), but also verb phrases made up with an auxiliary verb and a non-verb element such, as adjective, adverb, and noun (e.g., "is higher", "is full").

Constraint element (lines 2 and 3, Fig. 4) specifies what constraint/restriction will be placed on some requirements and what action must be taken in imposing these constraints. The restriction that this element imposes is defined in its query expression. In addition to supporting RDL queries, the Constraint query may define a "free text" statement, i.e., a constraint that is not expressed in terms of any RDL elements. For instance, it could simply define that the set of requirements selected by the Base query will be included into the next release planning by stating "Include requirements into next release".

```
1 <Composition name="CancelAuctionComposition">
2  <Constraint operator="begin/end">subject="seller" and relationship="end" and
3     object="auction"</Constraint>
4     <Base operator="ifNot">subject="auction" and relationship="begin"</Base>
5     <Outcome operator="ensure"/>
7 </Composition>
```

Fig. 4. An example composition in the RDL. The composition rule states that all requirements matching the semantics that the *seller end*s an *auction* can be satisfied only if the requirement that an *auction* has *begun* has *not* already been satisfied. In other words, a seller can only end an auction if it hasn't already started. Essentially this composition expresses an XOR dependency between a set of requirements described in the Constraint and Base. The "ensure" outcome operator reflects that as a post-condition to this composition we should check that the stated dependencies have been realised.

Base element (e.g., line 4, Fig. 4) reveals the set of requirements that are affected by the elements selected in the Constraint element's query. The operator in Base element (e.g., `operator="ifNot"`, line 4, Fig. 4) depicts the temporal or conditional dependency between the requirements selected by the Base element query and those of the Constraint query.

The composition operators used in Base elements reflect the ordering or conditional dependencies of requirements. Expressions such as "first… then", "once", "if" and alike are used to express such semantics. Therefore, the base operators fall into three categories: (1) Temporal operators, e.g., *before*, *after* and *meets*; (2) Concurrent operators, e.g., *overlap*, *during*, *starts*, *finishes* and *concurrent*; (3) Conditional operators, e.g., *if* and *if not*. We describe the meaning of these operators in more detail when discussing the mapping in Sect. 5.3.

It is worth noting that since the RDL is based on a symmetric model, it is possible to choose any set of concerns (using semantic queries) as Constraint and any other set as Base. The same requirement may be selected by a Constraint query in one composition, and by a Base query in another. We do not discuss the actual composition process and subsequent analysis in this paper (details available in [15], using ideas from [45]). Also, in the remainder of this paper, for brevity, we refer to "the set of concerns/requirements selected by the Constraint element's semantic query" as constraint, crosscutting or aspectual requirements. Similarly "the set of concerns/requirements selected by the Base element's semantic query" may be referred as base or non-aspectual requirements.

Outcome element defines how imposition of constraint requirements upon the base set of requirements should be treated. For instance, the outcome element may specify

a set of requirements that must be satisfied as post-conditions upon application of the Constraint. Unlike for Base and Constraint elements, the semantic query of the Outcome element can be empty, if no additional requirements/concerns are affected due to the Base and Constraint element interactions.

In summary, the composition example in the above figure communicates the following dependency: a Constraint operator has the role of the Causer setting in motion some Thing possibly at some Location. This role must be filled in by the sub-elements of the Constraint and Base elements to satisfy the Constraint operator semantics. Here, the Causer role is filled in by the seller, the Thing role by auction and the auction element participates in two `Set_in_Motion` relationships: end and begin respectively within the Constraint and Base. The ifNot conditional operator in Base ensures that only one of these relationships holds at any given time for the given set of seller and auction elements in subject/object roles, as defined within the composition.

It is worth reiterating that it is a characteristic of our approach that there is no asymmetry between concerns, as there is no semantic difference between aspectual and non-aspectual concerns. The issue of crosscutting arises only due to the way concerns interact. The aspectual (i.e., broadly scoped) relationships are shown by the compositions. In compositions the Outcome element may define the additional results of such relationships. In some cases where there are no additional effects of the crosscutting relationship defined (i.e., the Outcome element has an empty query), the symmetric view of concerns can be observed even in the composition.

3.4 RDL Dedicated Annotation and Analysis Support

In the previous section, we have outlined our requirements description language which utilises the semantics of the natural language (i.e., Relationship types, synonym or ontology-based querying) and the syntactic functions (i.e., subject, verb phrase and object) of the natural language elements to define queries and compositions. However, in order to use the RDL elements, we first need to introduce them into the requirements specification. We have already outlined the tools used for the RE process in Sect. 2. Here, we only summarise the additional tool extension required specifically for RDL annotation application.

As mentioned before, the automation for the RDL annotations is provided via an extension of Wmatrix. Wmatrix already provided part-of-speech and semantic annotations along with a wealth of other details, such as word frequency profiling and keyword analysis. In order to support text annotation with the RDL, Wmatrix was extended with such functionality as tagging the subject, verb and object elements, and RDL verb classes. For the subject, verb, and object element identification the existing rules for grammatical relationships from the SketchEngine [5] system used in the dictionary publishing industry were adopted. For the RDL verb classes a new tag set was defined and this was mapped to the Wmatrix classification. In some cases the groups of RDL and Wmatrix native verbs cases had direct correspondence, while in other cases a word by word mapping was required.

The MRAT [15, 68] Eclipse plug-in was specifically developed to support RDL composition definition and analysis using the results of Wmatrix annotations. MRAT provides both editing support for the RDL-annotated documents, and a number of

XML-based, or graphical (e.g., tree-based) views for analysis. For instance, a view for selecting the requirements from the input documents that will be affected by a given semantic query, or a view for observing temporal conflicts between specified compositions.

Having outlined the elements of RDL and its composition, we turn to presentation of the AO-ADL to present the model onto which the RDL elements are mapped.

4 Architecture Description Language (AO-ADL)

Several aspect-oriented software architecture specification approaches have been proposed for modelling software architectures [16]. Analyzing these approaches, such as AOGA [37, 38], AspectualACME [26] TranSAT [10], the PCS Framework [34], PRISMA [47], Fractal [48] and the DAOP-ADL language [49], we have found that even though they can be seen as complementary approaches, covering most important AOSD concepts, they provide a very heterogeneous set of first-order artefacts for the purpose. Moreover, the representation of AOSD concepts in some of these approaches is highly implementation language-dependent.

The main architectural elements of AO-ADL [36, 50, 51] are components and connectors. Note that these are the same as that of traditional ADLs [7, 27]. We on purpose avoid the definition of new architectural elements, instead integrating typical aspect-oriented concepts, such as join points, pointcuts, aspects and weaving, within the definition of components and connectors. (Please note that we still use terms “aspectual” and “non-aspectual” for clarity in the rest of the paper, though no such distinction is necessary for use of our AO-ADL).

As such, like the RDL, the AO-ADL has a symmetric concern model with both “base” and “aspectual” components treated as components. This is because the distinction between crosscutting and non-crosscutting concerns is merely a syntactic sugar and the main difference among these entities is in the role they play in a particular composition binding. Therefore, instead of inventing a new structural element, we redefine the connector and extend its semantics with aspectual bindings. When no crosscutting behaviour is specified, the software architect can use the same component bindings as in traditional ADLs, so component reuse is enforced.

The other key design principles of our AO-ADL can be summarized as follows:

1. The pointcuts are not specified jointly with the aspect advice. Instead, they are specified separately from components and inside the connectors. This separation increases the possibility of reusing components in different contexts and allows the representation of both components and aspects using the same architectural entity.
2. There are no “introduction–like” constructs in our AO-ADL. Instead, in order to adapt the interfaces of components to reuse them in other contexts, we make use of traditional architectural concepts of composite components. At the architecture level the purpose for extending the interface of a component is to let this component play a new role, specified as a set of methods. Composite components are the natural artefacts for this kind of evolution at the architectural level. The original component is then transformed into a compound component with a new interface, in addition to the previous one.

Since many ADLs already exist, we could have extended one of these languages with aspect-oriented characteristics [3, 20, 27]. Indeed, this was one of the paths we selected in developing the previous version of AO-ADL (called DAOP-ADL). DAOP-xADL is an implementation of DAOP-ADL that was built by extending xADL. As a result, we observed that xADL simply provided an empty template that was to be filled with the new characteristics [24] of DAOP-ADL. We also had to extend the tool for graphical representation of the xADL architecture. Thus, the benefits of extension-based approach are that the XML schema for the new ADL is already implemented and the effort for developing tools is reduced. However, there is a significant drawback in that the XML of the extended ADL becomes much more complex, less readable, and awkward. An even bigger problem of extending a language is obscuring the semantics of the elements of the new language. For instance, ACME defines the property concept to extend a component or connector with new characteristics. If we were to extend ACME for AO-ADL, all AO-ADL distinctive features would be ACME properties. This means that the semantics of any of these new characteristics would be hidden under a unique concept, the property. In conclusion, we deduced that for AO-ADL it was not worth extending an existing ADL, since AO-ADL required a number of innovative characteristics the semantics of which we wanted to keep clear.

Like RDL, AO-ADL is also an XML-based language[9] hence facilitating potential automation of the mapping (discussed in Sect. 5) through XML transformations [2] in the same spirit as model-driven development [60, 61].

4.1 The AO-ADL Meta-model

The meta-model of AO-ADL is shown in Fig. 5, where we can observe that the main architectural elements in AO-ADL are components and connectors.

A **Component** is the locus of computation and state [42]. As discussed above, we have chosen a symmetric approach for representing both components and aspects, without distinction, with a single architectural element. This element provides the following characteristics for components and aspects:

1. They are identified by a required unique *name* in a particular architectural design.
2. They are described by means of their ports. AO-ADL has two kinds of ports: *provided* and *required interfaces*. The definition of these interfaces determines the different *roles* the components can play in a particular architectural design. Thus, the role name has to be seen as an identifier that helps to distinguish the different interfaces of components. For example, in an auction system a customer plays the role of both buyers and sellers, and therefore, the `Customer` component has two provided interfaces, with role names *buyer* and *seller*.
3. They can both expose their *public state*.

[9] In order to avoid the direct manipulation of XML documents we are currently developing an Eclipse plug-in to generate and manipulate AO-ADL software architectures. Concretely, we are adapting the tools already available for DAOP-ADL. See http://caosd.lcc.uma.es/CAM-DAOP/tools.htm for more details.

4. They can optionally expose their *semantics*, which is a high-level model of a component's behaviour [42]. This behaviour is expressed in terms of the elements (interfaces, operations, state attributes) that are exposed in the specification of the components. For instance, it may reflect the ordering of the reception of operations in provided interfaces and the sending of operations through required interfaces.
5. They can interact with the rest of the components in the application, and consequently, can have different kinds of dependencies with the rest of the components in the application.

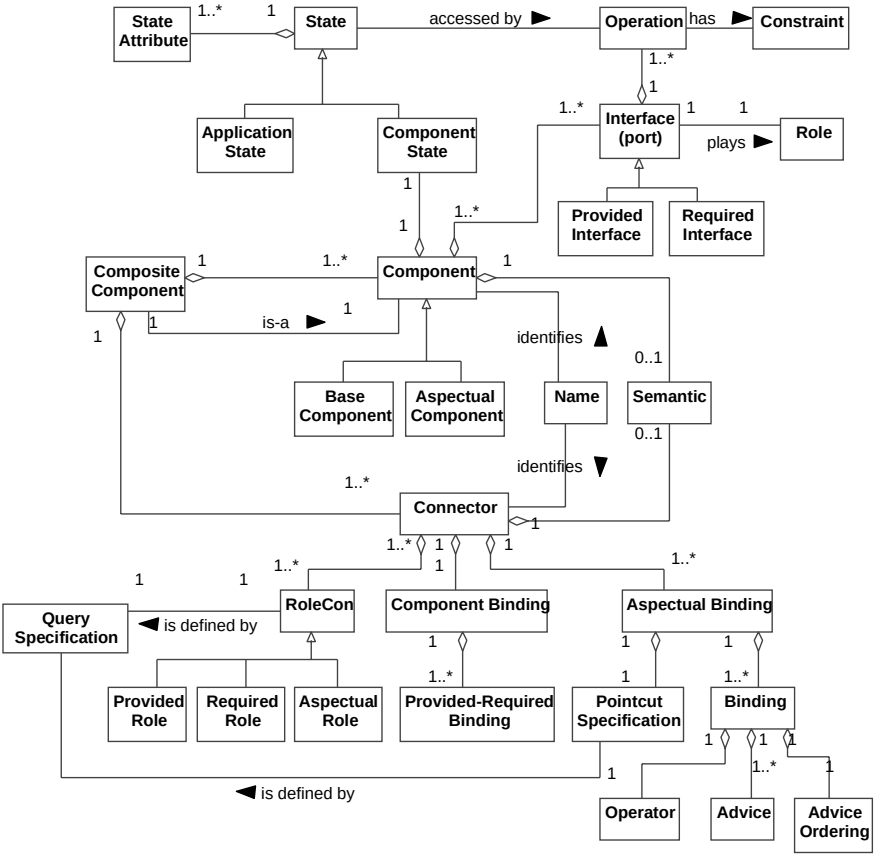


Fig. 5. AO-ADL Meta-model

Note that, as shown in Fig. 5, in spite of using the same architectural entity, we offer the possibility of optionally identifying base components and aspectual components. This is useful in those cases where the software architect desires to explicitly express this role of the component during its specification.

The concept of **component state** is used here with the same meaning as in traditional Interface Description Languages (IDLs) of distributed systems. Thus, it

represents the component's public state — e.g., the information that should be made persistent to be able to later restore the state of a component.

The concept of **application state** in AO-ADL identifies and models separately the data dependencies among both base and aspectual components. In the particular case of dependencies between aspectual components, the concept of application state allows to model, at the architectural level, certain types of interactions and dependencies among aspects. For example, consider the dependency between an authentication aspect and an access control aspect in the auction system. The first one obtains the name of the user during the authentication process. Later, the second one will need the identification of the user to check if s/he is allowed to do some actions in the application. The description of this information is external to any of the components described as part of the architecture of the application. Particular components only reference the attributes of the application state to indicate if they would generate this information (*outputState*) or if they would need this information (*inputState*).

A **Composite Component** in AO-ADL can be used to encapsulate sub-architectures, or to define different levels of abstraction of a component. This kind of component is internally composed by a set of components and connectors, which specify the interactions and relationships among them. Additional characteristics of traditional ADLs, like for instance ACME properties [7] could be easily included in AO-ADL, if needed, by using the XML extension mechanisms.

4.2 Composition of Concerns in AO-ADL

The **connector** is the architectural element for composition in AO-ADL. Connectors are the building block used to model interactions among components and rules that govern those interactions [42]. There are two main differences between the representation of connectors in traditional ADLs, and in AO-ADL. The first difference is the distinction between component bindings (interactions among base components) and aspect bindings (interactions between aspectual and base components). This means that components referenced in the aspect bindings are playing the role of an aspectual component in the specified interaction. The second difference is in specification of the roles of connectors. In AO-ADL the roles of connectors are defined specifying a query that identifies the component(s) that may be connected to these roles. Thus, instead of associating a particular interface to a role, a role may be described using a query describing "any component playing a particular role", or "any component containing a set of particular operations in one of its provided interfaces".

Additionally, and similarly to a component, a connector can also expose its **semantics**. In this case, the semantics will expose a high-level model of the interaction among components.

Figure 6 shows the `CustomerAuctionSecurity` connector for the auction system description in our AO-ADL. It describes the interaction between the `Customer` component (specified in the provided-role clause of the connector in lines 2–7) and the `Auction` component (specified in the required-role clause of the connector in lines 7–11). Since the interactions among these components have to be secure, a `Security` component is also connected to this connector (specified in the aspectual-role clause of

the connector in lines 12–16). This component behaves as an aspectual component and, therefore, the information to bind `Security` with `Customer` and `Auction` differs from the information to bind `Customer` and `Auction`, as described below.

4.2.1 Base Component Composition

The specification of the component bindings inside the connector describes the connections among base components. In Fig. 6, lines 17–22, the *componentBinding* clause in the connector describes the interaction between the `Customer` and the `Auction` components by connecting the corresponding provided and required roles previously described in the connector. Note that we do not discuss the kind of interactions or communication mechanisms among components that may be considered in AO-ADL, i.e., the type of connector (message-oriented infrastructures, pipes or filters.). Interested readers are referred to [43, 62] for such a discussion.

4.2.2 Aspectual Component Composition

The specification of the aspect bindings describes the connections between aspectual components and base components. In Fig. 6 the `Security` component affects the interaction among the `Customer` and the `Auction` components (lines 23–33).

Concretely, in an aspectual binding the following information is provided:

1. *Pointcut Specification*. The description of the pointcuts identifying the join points that will be intercepted. These pointcuts are to be defined in terms of: (1) a provided role of the connector; (2) a required role of the connector, or (3) a composition binding previously described in the same connector. In the example of Fig. 6, the join points captured by the pointcut specification are all the operations in the interaction between the `Customer` and the `Auction` component (specified referring to the binding name `Sell` in lines 25–27). Other possibilities would be to intercept the reception of a message in one of the provided interfaces of the `Auction` component [case (2) above], or the sending of a message through one of the required interfaces of the `Customer` component [case (1) above]. This can be expressed by omitting the other party in the communication from the pointcut specification.
2. *Binding*. A different binding (lines 28–31) is defined for each kind of advice (before, after, concurrent-with) to be included in the definition of the aspect bindings. For instance, the set of aspectual components being composed "before" a join point may be different from the set of aspectual components composed "after" a join point.
    2.1. *Operator*. The kind of advice is specified in the operator. AO-ADL operators are *before*, *after* and *concurrent-with*. They take the form: X before Y, X after Y and X concurrent-with Y, where X is an advice and Y is a join point. In Fig. 6, line 28, the operator *before** indicates that the `Security` component is injected always before the interaction among the `Customer` and the `Auction` component.
    2.2. *Constraint*. A constraint restricting the injection of aspects under certain circumstances. Constraints can be expressed as:
        - [pre-condition] X, the [pre-condition] must be satisfied before X
        - X [post-condition], the [post-condition] must be satisfied after X
        - [invariant] X, the [invariant] must be satisfied during X

In Fig. 6, line 28, the "`is_not_registered`" pre-condition is specified to avoid the execution of the corresponding `Security` advice once the user has already been logged on to the system.

2.3. *Advice*. For each kind of advice, the list of aspectual components to be composed is specified. This specifies the name of the aspectual component, and the name of an operation in one of its provided interfaces. The behaviour specified by this operation is the behaviour to be composed at the matched join points. In Fig. 6, the `log` method of the provided interface with role name `log-on` of the `Security` component is specified as the advice to be executed.

2.4. *Advice Ordering*. Several aspects being composed at the same join point need to be appropriately ordered. This information is provided for each aspect binding, since the ordering among the same set of aspects may be different depending on the context in which they are composed. The operators before, after and concurrent previously described are also used to indicate the order of execution of advice.

```
<connector name="CustomerAuctionSecurity">
    <provided-role name="CustomerSell">
        <role-specification>
            component-name="Customer" and interface-role="CustomerSell"
        </role-specification>
    </provided-role>
    <required-role name="AuctionSell">
        <role-specification>
            component-name="Auction" and interface-role="AuctionSell"
        </role-specification>
    </required-role>
    <aspectual-role name="Security">
        <role-specification>
            component-name="Security" and interface-role="Log-on"
        </role-specification>
    </aspectual-role>
    <componentBindings>
        <binding name="Sell">
            provided-role-name="CustomerSell" and
            required-role-name="AuctionSell"
        </binding>
    </componentBindings>
    <aspectBindings>
        <aspectual-binding name="security">
            <pointcut-specification>
                binding-name = "Sell" and operation-name="*"
            </pointcut-specification>
            <binding operator="before*" pre-condition="is_not_registered">
                <aspectual-component aspectual-role-name="Security"
                                     advice-label="log"/>
            </binding>
        </aspectual-binding>
    </aspectBindings>
</connector>
```

Fig. 6. Example of a Connector in AO-ADL

Since aspectual components encapsulate crosscutting concerns, this means that the usual situation is one in which the same aspectual component affects different connections between "base" components. For instance, in our case study `Concurrency` is an aspectual component that crosscuts all the interactions of any other component with the `AuctionSystem` component. Therefore, to avoid the

scattering of the same aspectual binding information through different connectors a connector specifying each identified aspectual composition is defined by making use of *wildcards* and *binary operators*. For the `Concurrency` component, the new connector would specify the following: "For all the source components that communicate with the `AuctionSystem` component, the `Concurrency` component is attached before and after the communication takes place". The "before" advice is the entry protocol to the critical section, while the "after" advice is the exit protocol. Notice that this is a simplification of modelling concurrency at the architecture level, only for the purpose of explaining the use of our queries. More information about the relationship of aspects and concurrency is discussed in [41].

This feature of AO-ADL enriches the expressiveness of connectors, but they still can be used with their initial meaning in traditional ADLs. This means that more conservative architectures may still be described using the traditional approach, where the roles of connectors are described by the list of operations associated to that role.

4.3 Definition of Connector Templates

Another feature of AO-ADL is the definition of *connector templates*. The idea behind connector templates is that the definition of the connections between aspectual and

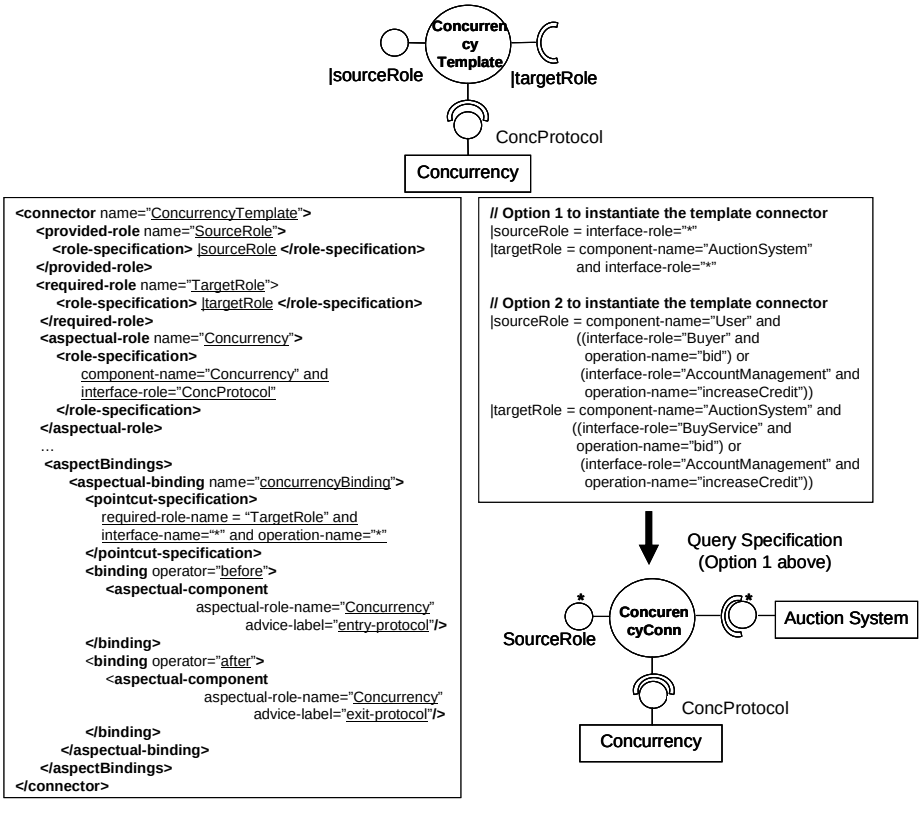


Fig. 7. Architectural connector template

base components from scratch is not a very practical solution since it requires a lot of effort and, potentially, it is an important locus of errors. An alternative option would be that the software architect has an available set of aspect-oriented architectural patterns or templates for well-known crosscutting concerns. Then, these templates could be instantiated in particular software architectures.

An example of a connector template is shown in Fig. 7. The template is interpreted as follows: "The behaviour of the `Concurrency` aspectual component is the same independently of the particular components connected to the provided and required role of this connector (notice the use of `|sourceRole` and `|targetRole` formal parameters). Always, the "entry-protocol" advice of the `Concurrency` aspectual component is executed "before" receiving a message in the target component, and always the "exit-protocol" advice of the `Concurrency` aspectual component is executed "after" receiving and executing a message in the target component.

Then, the `|sourceRole` and `|targetRole` formal parameters need to be substituted by particular component names or queries in order to instantiate the pattern, as shown in Fig. 7. While option 1 allows the description of the same query-based connector described at the end of Sect. 4.2 (see connector-diagram in lower right side of Fig. 7), option 2 defines a more concrete connector where particular components, interfaces and operations of the auction system application are specified.

We present the mappings to derive the AO-ADL architecture from the RDL specification next.

5 Deriving an AO-ADL Architecture from the RDL Specification

COMPASS aims to reduce the gap between RE and architecture derivation activities by providing a concrete mapping of the requirements specified in our RDL to the corresponding elements of the AO-ADL. However, the establishment of correspondence of the elements between the two languages is a non-trivial task as the granularity of conceptual elements differs significantly: multiple requirements and concerns from the RDL specification could potentially be realized by one architectural component, and, at the same time, a requirement of a single concern may motivate an entire component by itself. Therefore, our mappings have been derived and generalised from the concrete case study of mapping the RDL specification of the Auction system requirements to an AO-ADL description of its architecture. We will summarise the general mapping guidelines in Appendix A and discuss their application examples in Sect. 6.

5.1 Mapping RDL Concerns to AO-ADL

Concerns from RDL can be candidate simple components, composite components, interfaces or parts of components in AO-ADL. The specific corresponding element is defined by the semantics of the concern — i.e., whether a concern represents a specific stakeholder viewpoint, a system feature, a usage scenario, etc. For instance, viewpoint concerns normally map onto aspectual or non-aspectual components, while feature concerns normally map onto interfaces. One way of establishing the type of the concern is by analysing its name and its participation in the RDL composition. A summary of such analysis is presented in Table 1. It should be noted that this is only a very general guideline and exceptions to this are possible.

The information in Table 1 indicates that an RDL concern identified by a concrete noun or a noun + verb, that participates in RDL compositions as a base element — i.e., is not a crosscutting concern at the requirements level — would usually be mapped to a component in the architecture. Similarly, an RDL concern that is identified by an abstract noun, a noun + verb or a verb, and which participates in RDL compositions as a constraint element — i.e., is a crosscutting concern at the requirement levels — would usually be mapped to an aspectual component in the architecture. Finally, an RDL concern that is identified by an abstract noun, a noun + verb or a verb, and which participates in RDL compositions as a base element would usually be mapped to an interface in the architecture. This interface will then be part of the provided or required service of some of the components in the architecture.

Table 1. Mapping of concern by its name and composition participation

Concern	**Name**				**Participation in Composition**	
	Concrete noun (e.g., User)	Abstract noun (e.g., Coordination)	Noun+Verb (e.g., Information-Retrieval)	Verb (e.g., Sell)	Mainly <Const-raint>	Mainly <Base>
Component	☑		☑			☑
Aspectual Component		☑	☑	☑	☑	
Interface		☑	☑	☑		☑

5.2 Mapping Subject, Object and Relationship Semantics

In the RDL, the pattern SRO is clearly identified for each requirement in each concern. This pattern carries the main semantic load of a requirement. From our case study we have identified a number of possible mapping alternatives for the SRO elements to AO-ADL. Some of these alternatives can be used as independent patterns while others are only variants of the independent SRO patterns.

In the simplest "classic" case, a requirement will have all elements of the pattern present and each of these elements will map to a separate element of the AO-ADL. Thus, since Subject and Object in RDL are normally semantically significant nouns or noun phrases, they are likely to correspond to real world/solution space objects[10] and Relationships (i.e., verb phrases) are likely to correspond to operations over the related objects.

This case is illustrated in Figs 8 and 9 for a logging on requirement of the `Security` concern, which states that "User has to log on to the auction system for each session".

Figure 8a presents the RDL description of this requirement. When we study this requirement, we can observe that both user (Subject) and auction system (Object) must form separate components in the architecture, and that the "`log on`" Relationship

[10] Concrete nouns normally denote concrete objects in the solution space. On the other hand, abstract nouns denote concepts and alike, which often also may be realised in objects for architecture design purposes. Alternatively, they may be considered "features" and hence realised through interfaces at the architecture-level (cf. Sect. 5.1).

reflects an operation which the `AuctionSystem` component provides, and `User` component requires. Thus, the Relationship is mapped to an operation of the required interface of the component denoting the RDL Subject element, and the provided interface of the component denoting the RDL Object element. This relationship between the component interfaces is captured in the component binding section inside the AO-ADL connector shown in Fig. 8b, which describes the interaction between the `User` and the `AuctionSystem` components.

(a) RDL SRO Pattern

```
<Concern name="Security">
  <Requirement id="1">
    <Subject>Users</Subject>
    have to
    <Relationship type="Move"
    semantics="Group">log on
    </Relationship> to the
    <Object>auction system
    </Object>.
  </Requirement>
</Concern>
```

(b) ADL Mapping

```
<component name="User">
  <required-interface role="Log-on">
      <operation name="log on"/>
  </required-interface>
</component>
<component name="AuctionSystem">
  <provided-interface role="Log-on">
      <operation name="log on"/>
  </provided-interface>
</component>
<connector name="UserAuctionSystem">
  <provided-role name="UserRole">
    <role-specification>
        component-name="User" and interface-role="Log-on"
    </role-specification>
  </provided-role>
  <required-role name="AuctionRole">
    <role-specification>
      component-name="AuctionSystem" and
      interface-role="Log-on"
    </role-specification>
  </required-role>
  <componentBindings>
      <binding name="UserAuctionSystemB">
        provided-role-name="UserRole" and
        required-role-name="AuctionRole"
      </binding>
  </componentBindings>
</connector>
```

Fig. 8. RDL and AO-ADL descriptions of the Log on requirement

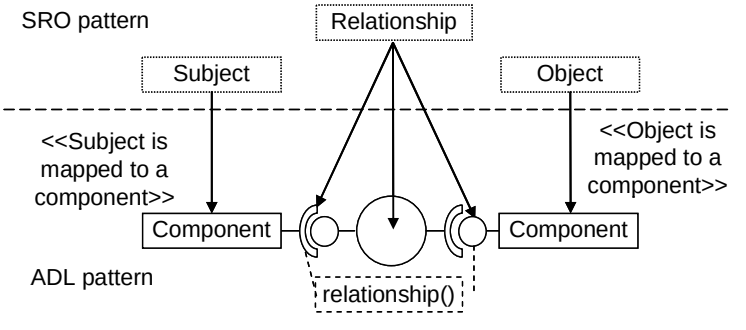


Fig. 9. SRO Mapping to Component – Relationship – Component

A visual representation of this mapping between the SRO patterns in the RDL to AO-ADL is shown in Fig. 9 — note that there is a corresponding AO-ADL pattern, component-operation-component, for the SRO pattern.

The "classical" SRO case demonstrates the mapping of an RDL statement where all three SRO elements are present and are mapped to individual AO-ADL elements. However, often the Object may be absent in a requirement clause, or alternative mappings of the RDL elements to AO-ADL may be justified by their semantics. These alternative mappings are summarized in Tables 2, 3 and 4.

Table 2 summarizes the Subject-Relationship patterns that can be used both independently, and as "part-patterns" where the Object is mapped in the same way as in the "classical" case.

Table 2. Subject-Relationship Patterns

SRO Pattern	ADL Mapping	Explanatory Comments
Case 1: Object is absent. E.g., "Buyers are bidding in parallel", "Auction is activated".	*Subject → Component* *Relationship → operation in the required interface of subject's component* Binding cannot be specified, as the provider of the operation is not evident. Additionally, this pattern can be used as indicators of possible component states (e.g., activated/not activated).	Object is often absent with intransitive verbs [45] (e.g., "Ice melted"). Object can also be missed out intentionally (implied) or by mistake.
Case 2: Subject is abstract concept. E.g., "Bidding is closed to new buyers".	*Subject → interface* *Relationship → operation of the subject's interface* Additionally, this pattern can be used as indicator of possible component states (e.g., closed/not closed).	This situation is feasible when a subject is an abstract "concept" (e.g., "Validation confirms user input").
Case 3: Subject is not tangible object. E.g., "The auction end date passed".	*Subject → attribute of some component* *Relationship → operation over that attribute*	In cases when a subject in the SRO pattern is not an independent tangible object but is referring to a specific property or characteristic (e.g., start date) of another object (e.g., auction). It is worth noting, that this case could be difficult to establish if the attribute is not easily identifiable as belonging to a specific object.

Table 3 shows the Object-Relationship patterns. These are unlikely to have independent uses, but will generally assume presence of a Subject which is mapped in the same way as in the "classic" case.

Table 3. Object-Relationship Patterns (OR)

SRO Pattern	ADL Mapping	Explanatory Comments
Case 4: Subject is implied. E.g., "Require increase of account credit".	*Relationship → operation* *Object → component which provides the relationship's mapped operation*	This case is mostly a sub-pattern of the general SRO, where subject is implied (e.g., "Require increase of account credit" may imply the system as a subject); however, it may also occur along side other above mentioned sub-patterns. It is advised to review such requirement statements.
Case 5: Subject is unspecified. E.g., "Someone initiates the sale".	*Relationship → Operation* *Object → Interface*	This case is similar to case 3. However, in this case an unspecified subject is used. An example of a requirement conforming to this pattern is the requirement "someone initiates the sale", where an unspecified subject initiates (relationship) the sale (object).
Case 6: Object is not tangible object. E.g., "Customer informs of the start date of the auction".	*Subject → Component* *Relationship → operation* *Object → State Attribute* Here the "start period" (which is synonymous to the *start date*) is the object in the RDL description, and is an attribute of the "auction" component. The "informs" Relationship depicts the state of that attribute (has it been sent to the auction?). Thus, the RDL "start date" object will map to the state attribute of auction component in the AO-ADL, with "inform" relationship mapping to an operation over that attribute.	This case is very similar to case 4, however in this case it is the object in the SRO pattern that is not an independent tangible object but is referring a specific property or characteristic of another object. Similar to case 4, in AO-ADL this object will correspond to an attribute of some component.

Finally, Table 4 summarizes the mapping in cases where subject and object are realized by the same entity.

The patterns in Tables 2, 3 and 4, along with the classical SRO to AO-ADL mapping, can be used to map RDL requirements to core and aspectual components, interfaces or state attributes in the architecture design. This "1 to n" mapping defined for the RDL subjects and objects implies different possible mapping links between the RDL and AO-ADL meta-models. We present this meta-model mapping in Sect. 5.4.

Table 4. Same Subject and Object Pattern

SRO Pattern	ADL Mapping	Explanatory Comments
Case 7: Use of reflexive verbs. E.g., "The buyers are bidding against each other".	*Subject → component* *Object → same component as Subject* If any of the sub-patterns occurs with the subject and object referring to the same entity, so will their architectural mappings. So, for example, if the subject is mapped to an interface (as in case 3), so will the object (case 6).	This occurs when a reflexive verb is used, or when the subject and the object explicitly refer to the same entity. In this situation both the subject and the object of RDL description are mapped to the same architectural entity (e.g., component or interface) in the AO-ADL since they represent the same concept in the RDL.

5.3 Mapping RDL Compositions to AO-ADL Bindings

The compositions of the RDL provide information about broad interactions (i.e., aspectual relationships) between various concerns and concern elements. In AO-ADL such information is contained in the aspect bindings, which are described as part of the connectors. Thus, there exists a very close link (often one-to-one mapping) between RDL compositions and AO-ADL aspect bindings.

An RDL composition element contains base, constraints and outcome sub-elements. Each of these corresponds to specific sub-elements in the description of aspect bindings in AO-ADL.

5.3.1 Constraints

In the RDL, for each composition, the constraint elements specify the crosscutting concerns affecting the concerns selected by Base query. In AO-ADL this information is mapped to the "aspectual component" elements of the aspect bindings. RDL uses semantic queries to identify the crosscutting concerns in terms of SRO, types, concerns, or requirements, The AO-ADL queries (with wild cards) are used in the definition of the connector roles. Then, AO-ADL specifies the "aspectual components" in terms of the aspectual roles, the role or interface that describes their behaviour and the particular operation (i.e., advice) to be composed at the matched join points. An example of this is shown in mapping of the RDL composition in Fig. 10 to the AO-ADL aspect binding in Fig. 11.

```
<Composition name="LoginComposition">
  <Constraint operator="affiliate">concern="Security"</Constraint>
  <Base operator="before">all concerns where subject="user" and
        (relationship="buy" or relationship="sell")</Base>
  <Outcome operator="ensure"/>
</Composition>
```

Fig. 10. Example RDL composition specification for part of the Auction System

In the RDL, as described in Sect. 3, constraints are described by a query and an operator. In Fig. 10 the query matches the `Security` concern as a whole, referenced via the concern name (`concern="Security"`). (Please note that other means are also available, for instance SRO patterns which match requirements of interest, verb types and semantics, element ids that can be used if syntactic match is required and specific element selection can also be used.) In AO-ADL this information is mapped to: (1) the specification of an "aspectual-role" to which the `Security` concern is connected (cf. lines 8–12 in Fig. 11), and (2) the specification of `Security` as an aspectual component in the <binding> sub-section of the aspect binding part (cf. lines 19–20 in Fig. 11). Notice that from the information in Fig. 10 only the concern name (e.g., Security) can be known. The references to the "`login`" operation in Fig. 11, through which the `Security` component is connected to the connector, are obtained by mapping relevant SRO patterns in concern requirements. Finally, consider that the information generated from the mapping process produces concrete connectors, since only this concrete information can be generated during the mapping process. A later refinement process would be needed (not covered in this paper) in order to group several connectors, describing the same crosscutting behaviours, in one that broadly-scoped query based on wild card use.

```
 1 <connector name="…">
 2  <required-role name="User">
 3    <role-specification>
 4          component- name="User" and
 5          (operation-name="buy" or operation-name="sell")
 6    </role-specification>
 7  </required-role>
 8  <aspectual-role name="Security">
 9     <role-specification>
10         component-name="Security" and operation-name="login"
11    </role-specification>
12  </aspectual-role>
13  <aspectBindings>
14    <aspectual-binding name = "LoginComposition">
15         <pointcut-specification>
16            required-role-name="User"
17         </pointcut-specification>
18         <binding operator="before*">
19           <aspectual-component  aspectual-role-name="Security"
20                                 criticality="critical" advice-name="login"/>
21         </binding>
22    </aspectual-binding>
23  </aspectBindings>
24 </connector>
```

Fig. 11. Example of AO-ADL mapping of the composition specification in Fig. 10

The operator from the RDL constraint element does not have an explicit representation in the AO-ADL binding, yet the semantics of this operator are useful for the mapping as they guide the architect with the knowledge on the kind of roles needed for the given constraint-base interaction. For instance, the `affiliate` operator in Fig. 10 requires the Member (a party that is to join some group or collection) and Group (the collection to be joined with) roles, and optionally a Causer role (the party that

causes the Member to join the Group). This operator indicates that for the present composition the constraint and base elements must satisfy the semantics of being a member of a certain group. Indeed, the intent of the `Security` composition, in mapping the `Security` aspectual component (the Causer) is to cause the instances of the `User` component (the Member) to join a certain "authenticated" Group. It is the fulfilment of this intent that the semantics of the constraint operations and their related roles help to verify.

5.3.2 Base

In the RDL, the Base element's query describes the concerns/requirements that are affected in a given composition. The base query is specified in terms of the RDL's join points. In AO-ADL this information is mapped to the "pointcut-specification"

Table 5. Base operator mapping: Temporal operators

RDL Base Operators	**Description**	**AO-ADL binding Operators**
X meets Y/ Y met by X	There is no temporal interval between the ending of X and the starting of Y: XXXYYY E.g., The trace concern meets the buying/selling of goods	Before/after Every time X occurs, Y also occurs before/after X. E.g., A `trace()` operation of the Trace component is invoked always just before invoking the `buy()/sell()` operations of a `User` component
X before Y/ Y after X	There is a temporal interval when X is completed but Y has not started yet: XXX YYY E.g., The enroll concern occurs always "before" any other concern i.e., the user can buy/sell only after enrollment Sometimes this is not mapped to an aspect binding, but to a constraint (pre-condition, post-condition or invariant) associated to an operation in the interface of a particular component. E.g., An auction is activated "before" the user can buy or sell	Before*/After* This is a variant of the previous before/after operators, where X occurs before/after Y only if it has not occurred before. E.g., Before the `User` component invokes a `buy/sell` operation: (1) if the user is already enrolled the operation is invoked. (2) if the user is not still enrolled, first the user is enrolled and then the buy/sell operation is invoked. Note that the before* operator is interpreted as follows: *if [X has occurred]* → *Y* *else-if [X has not occurred]* → *X;Y* *end-if*

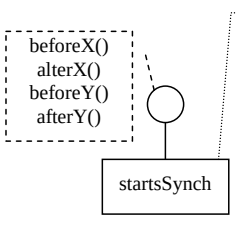
Table 6. Base operator mapping : Concurrent operators

RDL Base Operators	**Description**	**AO-ADL binding Operators**
X overlaps Y/ Y overlapped by X	X has commenced before Y; Y commences while X is in progress; X completes while Y is in progress: XXX YYY E.g., The "Run an auction" requirement overlaps the "Reserve highest bid" (whereby the amount of the bid placed is "reserved away" from the bidders credit) since the reservation is applicable only after the auction is started and the 1st bid is placed, but the reservation is maintained after the auction is closed and until the payment processing is complete.	concurrent-with
X during Y/ Y through X	Y has commenced before X; X commences while Y is in progress; X completes while Y is in progress. XXX YYYYYYY E.g., Log information informing that a user initiated an auction can be generated "during" the period of time in which the auction is initiated.	concurrent-with There are also situations in which Y can be a particular state in which the application or a component has to be in order for X to occur. In this case Y is an invariant for X. E.g., The user can buy/sell only "during" the period in which the auction is active. Then "the auction is active" is an invariant that needs to be satisfied while the "user can buy/sell".
X starts Y/ Y started by X	X has commenced simultaneously with Y; X completes while Y is in progress (from perspective of X). This is a subtype of during: XXX YYYYYY	
X finishes Y/ Yfinished by X	Y has commenced before X; X commences while Y is in progress; X and Y complete simultaneously. This is a subtype of during: XXX YYYYY	
X concurrent Y	X and Y are started and completed within the exact same interval: XXX YYY This operation is used for real-time synchronisation. For instance, requirements related to video and audio transmission for a movie need to be concurrent.	concurrent-with

element of the aspect bindings. Similarly to the RDL Base query, the pointcut in AO-ADL specifies the join points affected by aspectual components. These join points are defined in terms of the base components connected to some of the provided and/or required roles of the connector. In Fig. 10 the Base query matches the `User` component and the operations "`sell`" and "`buy`" defined on interfaces of such component (cf. lines 2–7 in Fig. 11). Additionally, the RDL Base contains an "operator" attribute that is mapped to an "operator" attribute of the AO-ADL binding. This mapping is shown in Tables 5, 6 and 7. For the first two columns of the tables X and Y depict two requirements and their spatial positioning towards each other in the table reflects their temporal ordering. The third column shows how the mapping of the temporal perspective of these requirements' interactions is realised in the AO-ADL.

Notice that the several variants that our RDL offers to express concurrency are all grouped under the same "concurrent-with" operator in our AO-ADL. This operator is a high-level operator that allows to specify the parallel occurrence of X and Y in high-level descriptions of the software architecture. The particular semantics of each RDL operator provide extra information about the synchronization of the operations of the involved components and may be mapped to a synchronization component. The synchronization component would be then related to the components to which X and Y refer.

For instance, let us consider the RDL operator "X starts Y", which as described above is interpreted in the RDL as that: (1) X and Y starts simultaneously, and (2) X completes while Y is in progress. In order to express these constraints at the architectural level, this is mapped to an AO-ADL startsSync synchronization component. The provided interface and the semantic of this component are described in Fig. 12. The semantics of the component are described in an adapted COOL notation [41], though with some differences. Concretely, it specifies the conditions under which the execution of X and Y must be suspended and reinitiated to satisfy the constraints specified by the RDL operator. Note that AO-ADL allows embedding of domain-specific languages, such as COOL. This feature is used by other ADLs as well (e.g., ACME [7]). Its main benefit is the possibility of reusing existing standard




```
//Semantic of the component
//Condition Vars
condition X_wants_to_start = false;
condition Y_wants_to_start = false;
condition Y_wants_to_finish = false;
condition X_finished = false;
beforeX → X_wants_to_start = true;
            if (Y_wants_to_start){
              notify(X); notify(Y);
            }else{
              wait(X);
            }
afterX  → X_finished=true;
            if (Y_wants_to_finish){
              notify(Y)
            }
…
```

```
…
beforeY → Y_wants_to_start=true;
            if (X_wants_to_start){
              notify(X); notify(Y);
            }else{
              wait(Y);
            }
afterY  → if (!X_finished){
              wait(Y);
            }

//Legend
// wait(X): blocks X
// notify(X): continues X execution
```

Fig. 12. Example of AO-ADL "startsSync" synchronization component

or well-adopted (formal) notations. The alternative is to define an equivalent XML Schema for synchronization and incorporate it to the XML Schemas of AO-ADL.

The synchronization components corresponding to overlap, during, finishes and concurrent RDL operators can be described in a similar way; only the semantic of the component will vary. Furthermore, these synchronization components are examples of AO architectural solutions to recurrent requirements operators, so they are considered predefined components that can be reused during the mapping process.

5.4 Structural Mapping of Meta-models

The discussion in Sects. 5.1, 5.2 and 5.3 has shown that there is not always a direct 1-to-1 mapping between the concepts of the RDL and AO-ADL. Our experience with the Auction system case study and the following reflection and review has led us to the structural mapping between the RDL and AO-ADL meta-models shown in Table 8.

Table 7. Mapping of RDL meta-model elements to AO-ADL meta-model

RDL Meta-model Elements	**Corresponding AO-ADL Meta-model Elements**
Concern	Composite component, component, interface, or role
Requirement	See Subject, Object and Relationship mappings below.
Subject & Object	Component, state attribute, interface, or role
Relationship	Operation of an interface (provided by the component representing the object in the SRO pattern and required by the component representing the subject in the SRO pattern)
Composition	Connector
Base	See Base Operator and Base Query Expression mappings below.
Base Operator	Binding type in AO-ADL
Base Query Expression	Role specification and Pointcut specification
Constraint	Binding section in aspectual-bindings. Also see Constraint Operator and Constraint Query Expression mappings below.
Constraint Operator	No direct mapping, but these are mapped via semantic roles realised by sub-elements from the constraint and base pointcuts.
Constraint Query Expression	Aspect, Role, and Advice in AO-ADL
Outcome	Constraints (post-conditions), critical execution paths

6 Mapping Examples and Discussion

The generalisation and summary of mapping guidelines discussed in Sect. 5 is provided in Appendix A. Below we provide some examples of application of these guidelines as well as discuss problems and future directions of this work.

6.1 Mapping Examples

As an example of the application of these guidelines (summarised in Appendix A) we discuss several RDL to AO-ADL mapping scenarios from the Auction case study.

Scenario 1: Security Concern. Let us consider the RDL description of the `Security` concern which was presented in Fig. 8a. Applying the concern mapping guideline (Appendix A:1 summarised from Table 1), we note that the `Security` concern is identified by an abstract noun which is a constraint in the RDL composition (see Fig. 10) and consequently this concern is mapped to a `Security` aspectual component. Using the Requirements Mapping Guidelines (Appendix A:3, 4, 5) we note that in this case the classic SRO pattern is used and the subject is mapped to a `User` component with the `log-on` operation in one of its required interfaces and the object is mapped to a `System` component with the `log-on` operation in one of its provided interfaces. Additionally, guideline 2 says that "a requirement is associated with the concern in which it is defined", and thus the `log-on` operation should be associated with the `Security` component.

That is the same as saying that `Security` is somehow related to the "log-on" interaction between `User` and `System`, since this binding information was extracted from a requirement of the `security` concern.

So far the mapping has resulted in the structure presented in Fig. 13a.

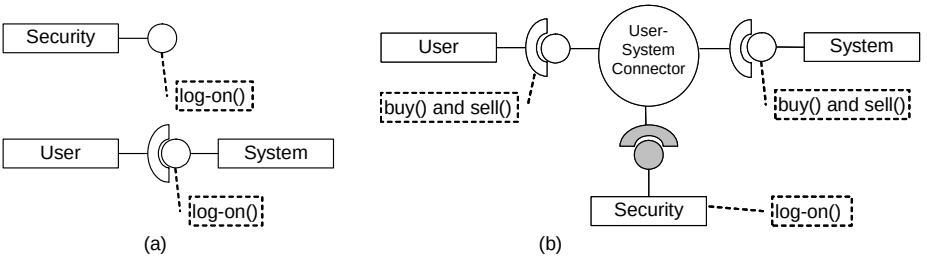


Fig. 13. Mapping Security concern: **a** without Composition **b** with Composition and a connector visual representation

Now, looking back at Fig. 10 in Sect. 5.3 we can relate the previous discussion to our composition mapping guidelines. Thus, applying these guidelines (Appendix A:6) we identify that the `LoginComposition` has the Causer, Member and Group roles to be filled in. We identify `Security` aspectual component as the Causer, and `User` component as the Member and System as the Group which the Member needs to join. Guideline 6 also instructs us to investigate all interfaces of the `Security` concern for acting as advice. Security has only one interface — log on — which is the only operation through which the Causer can enforce the `User` (Member) joining the `System` (Group). Thus, `log-on` is mapped onto an advice operation.

By applying the Base element mapping guideline (Appendix A:7), we map the base operator onto the type of advice operator in AO-ADL (the before* operator), and the set of roles (`buy` and `sell`) which are affected (i.e., advised) by this composition.

Application of the Outcome mapping guideline (Appendix A:8) implies that `Security` is a "critical" aspectual component, since the Outcome in the present composition re-enforces the need to ensure the composition requirement. No additional post-conditions are needed as the outcome does not add extra information about other requirements to be satisfied.

The ADL binding for the `LoginComposition` is illustrated in Fig. 11, while the visual representation of the concern and its composition is provided in Fig. 13b.

Scenario 2: The "Validate Bid" concern. Let us consider the RDL description of the "`Validate Bid`" concern and the "`ValidateBidComposition`" composition in Fig. 14. The "`Validate Bid`" concern has two requirements, each with a sub-requirement.

```
<Concern name="Validate Bid">
  <Requirement id="1">A new <Subject>bid</Subject>
    <Relationship type="General_Action" semantics="Compare">is valid</Relationship> if:
    <Requirement id="1.1">The <Subject>bid</Subject> must <Relationship type="General_Action"
        semantics="Compare"> be over</Relationship> the minimum
      <Object>bid increment</Object>
      <Requirement id="1.1.1">The <Object>increment</Object> is
        <Relationship type="Mental_Action" semantics="Solve">calculated</Relationship> purely
          on the <Object>amount </Object> of the previous high bid
      </Requirement>
    </Requirement>
    <Requirement id="1.2">The <Subject>bidder</Subject> <Relationship type="Rest"
        semantics="Maintain">has</Relationship> sufficient <Object>funds</Object>
      <Requirement id="1.2.1">The customer's <Subject> credit</Subject> with the system
        <Relationship type="General_Action" semantics="Compare">is [high]</Relationship>
        at least as high as the <Object>sum </Object>of all pending bids.
      </Requirement>
    </Requirement>
  </Requirement>
</Concern>
<Composition name="ValidateBidComposition">
  <Constraint operator="enforce">concern="Validate Bid"</Constraint>
  <Base operator="meet">all requirements where any relationship="bid"</Base>
  <Outcome operator="ensure"/>
</Composition>
```

Fig. 14. "Validate Bid" concern and "ValidateBidComposition" composition specified in RDL

The "`Validate Bid`" concern is mapped to a `ValidateBid` aspectual component, in accordance with the concern mapping guidelines. Notice that this is not a well-known crosscutting concern such as security, enrolment or concurrency. However, the identification of bid validation as an aspectual behaviour at the architectural level is possible due to the classification we made in Table 1, and the occurrence of "`Validate Bid`" as a constraint in the RDL "`ValidateBidComposition`" composition (Table 1 and Appendix A:1).

Now, the mapping of the requirements defined in the "`Validate Bid`" concern has to be done (Tables 2, 3, and Appendix A:3, 4, 5). Requirement 1 has subject `bid` and relationship `is valid`. As per guideline on mapping SRO (case 1, Table 2), these will map to a component and its required operation. Requirement 1.1 has `bid` subject, `is over` as relationship, and `increment` as object. These map to a

component, its required interface, and an attribute of a component over which the respective operation is applied (case 6, Table 3). In accordance with the Relationship mapping guideline (Appendix A:5), the component to which the attribute pertains will have the operation as part of one of its provided interfaces. Here it is necessary to identify the components to which the given attribute belongs. (This information is obtained from the mapping of other concerns/requirements in the RDL description, this being the reason for temporarily naming these components as "?". For instance, requirements in other concerns of the case study indicate that auctions have a "bid increment", or that it is the auction system component which manages the `credit/funds` of bidders.).

The next requirement has Subject as `increment`, Relationship as `calculated` and Object as `amount`. These map to two attributes and an operation over these attributes (case 3 Table 2 for SR, and case 6 Table 3 for RO mapping). Applying Relationship mapping guideline (Appendix A:5), `calculated` is mapped to a part of the required interface of the component containing the `increment` attribute and of the provided interface of the component containing the `amount` (of the previous high bid). Once again it is necessary to identify the components to which the given attributes belong.

Requirement 1.2 has SRO elements with `bidder`, `has`, and `funds` respectively. These map respectively to a component, operation, and attribute (case 6, Table 3). Finally, requirement 1.2.1 has SRO with `credit`, `is high`, and `sum` (of all pending bids) elements, which correspond to attribute, operation, and attribute respectively. Since credit and funds are synonyms, credit belongs to the same component as the funds. This mapping is visualised in Fig. 15a below.

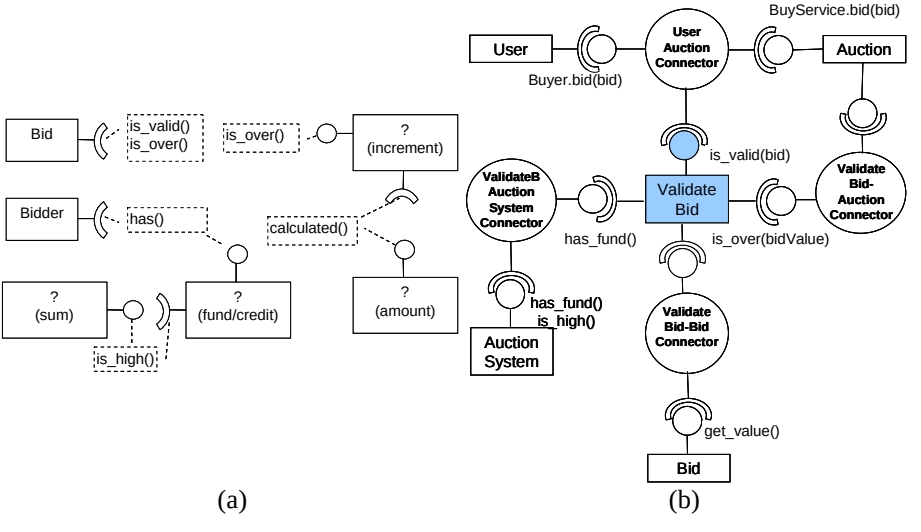


Fig. 15. a Mapping of the requirements in the "Validate Bid**"** concern; **b** Proposed architecture after the software architect's refinement

Additionally, guideline 2 (Appendix A) says that "a requirement is associated with the concern in which it is defined", and thus the `is_valid`, `is_over`, `has_fund`,

calculate and is_high operations should be associated with the ValidateBid component. Concretely, the resulting mapping in Fig. 15a provides information about the dependencies of the "ValidateBid" component with other components in the system during the bid validation process; dependencies that shall determine parts of the provided and the required interfaces of the "ValidateBid" component.

At this point the software architect should be involved since the decision of including each of the operations as part of either a provided or a required interface of the component is intuitive. For instance, checking the result of the mapping in Fig. 15a the software architect may identify the is_valid operation as the main service to be offered by the ValidateBid component and, therefore, this operation should be part of the provided interface of this component. Additionally, since the "ValidateBid" component is an aspectual component this means that there is a crosscutting influence between these two components. This is enforced by the RDL composition information shown in Fig. 14. The software architect identifies the rest of operations as the information that the ValidateBid component will need to obtain in order to validate the bid, and therefore, these operations should be part of the required interfaces of the ValidateBid component, and thus the validate component has a dependency on components providing this information. A further refinement may reveal that separating bid validation in an aspectual component may also imply elimination of some operations from the required interfaces of other components in order to avoid scattering of the validation functionality. For instance, in the current example the Bidder component may not require information on having funds, since that is now the responsibility of the ValidationBid component. The resulting software architecture is shown in Fig. 15b.

Please note that the architect could replace names of the attributes and operations with more meaningful ones. Such refinement was already included in Fig. 15b, where the has operation was renamed to has_fund and the calculated operation to calculate.

Now, applying the composition mapping guideline for Base mapping (Appendix A:6), the ValidateBid aspectual component is related to all the interactions in which the message name is "bid". In AO-ADL this is described using wild card in the definition of the provided and required roles of the Validation connector, as shown in Fig. 16 descriptions. The component and aspectual bindings are also specified in Fig. 16. First, the RDL base operator "meet" has been mapped to the AO-ADL "before" operator according to the operator mapping, as per Table 5. Second, the RDL "ensure" outcome is mapped to the definition of a critical path execution, indicating that this is a critical aspectual behaviour.

In the previously discussed Security scenario systematic application of the mapping guidelines immediately resulted in a satisfactory solution for the software architect. However, the final mapping may not always be apparent from mere guideline application, as demonstrated by the mapping of ValidateBid. In such more complex cases the judgement of either the requirements engineer and/or of the software architect may be required. Such issues are discussed in the following section.

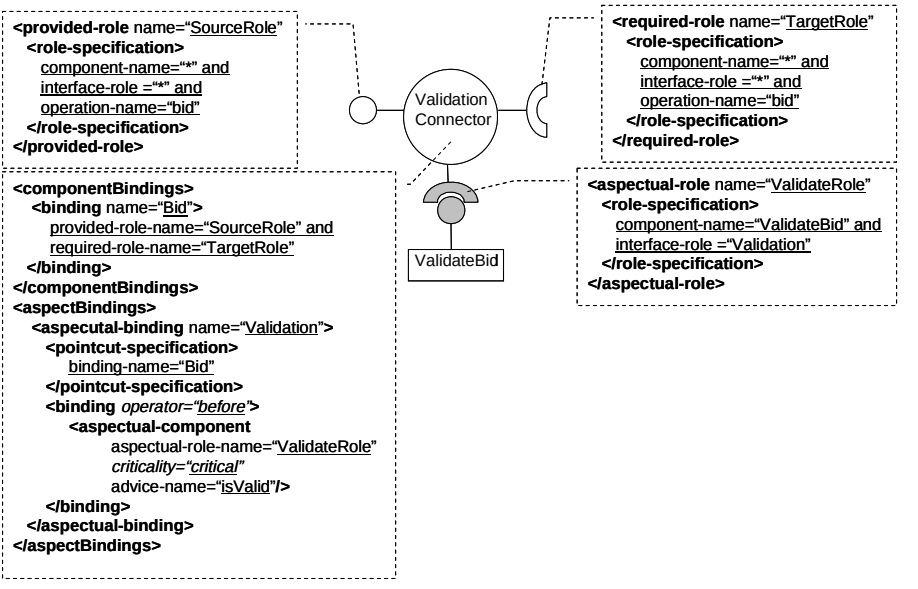


Fig. 16. Mapping Validation concern and its composition with a connector visual and XML representation

Scenario 3: Concurrency Concern. Fig. 17 shows the specification of the "Concurrency" concern and composition in RDL. Applying the concern mapping guideline (Appendix A:1), and according to the information shown in Table 1, this concern is mapped to the `Concurrency` (aspectual) component. This decision is enforced by the RDL composition in Fig. 17**,** where this concern is an RDL Constraint.

(a) RDL Concurrency concern

```
<Concern name="Concurrency">
   <Requirement id="1"> The
   <Subject>auction system</Subject>
   <Relationship type="General_Action"
   semantics="Constrain">is concurrent
   </Relationship> -
   <Subject>clients</Subject>
   <Relationship type="Move"
   semantics="Transfer_Posession">
   bidding</Relationship> in parallel, and a
   <Subject>client</Subject>
   <Relationship type="Move"
   semantics="Transfer_Posession">placing bids
   </Relationship> at different
   <Object> auctions </Object> and
   <Relationship type="Affect"
   semantics="Modify"> increasing</Relationship>
   his/her <Object>credit</Object> in parallel.
   </Requirement>
</Concern>
```

(b) RDL Concurrency Compocition

```
<Composition name="ConcurrencyComposition">
   <Constraint operator="enforce">
   concern="Concurrency"
   </Constraint>
   <Base operator="along">all requirements where
   any (relationship="bid" OR
   (relationship="increase" AND object="credit" ))
   </Base>
   <Outcome operator="ensure"/>
   </Composition>
```

Fig. 17. RDL Concurrency concern and composition

Next, the requirements of the Concurrency concern are mapped. From the requirements semantics mapping guidelines (Appendix A:3,4,5) the subjects/objects named "`Auction System`", "`Clients`", "`Auctions`" and "`Credit`" are identified — client is a synonym of user, already identified in Scenario 1. Also, the "`bidding`", "`placing bids`" and "`increasing`" operations are identified – "bidding" and "placing bids" are synonymous. After analyzing the relationship between the `Concurrency` component and elements of the SRO patterns (Appendix A:2) the software architect identifies the crosscutting influence between the `Concurrency` component and the relationships among components generated as part of the mapping of SRO patterns. The graphical representation of the applied mapping is shown in Fig. 18a.

The problem with the obtained mapping (see Fig. 18a) is that the aspectual binding of the Concurrency component is scattered in different connectors. The solution to this shortcoming requires a later refinement of the software architecture, making use of AO-ADL wildcard-based queries to generate the `ConcurrencyConnector` illustrated in Fig. 18b.

The information in Fig. 18b is aligned with the information obtained from the mapping of the RDL composition shown in Fig. 18c. In accordance with the composition mapping guideline (Appendix A:6) the "concern = Concurrency" RDL Constraint identifies concurrency as an aspectual behaviour. The base identifies the components that have `bid` and `increase` in their required interfaces, the components

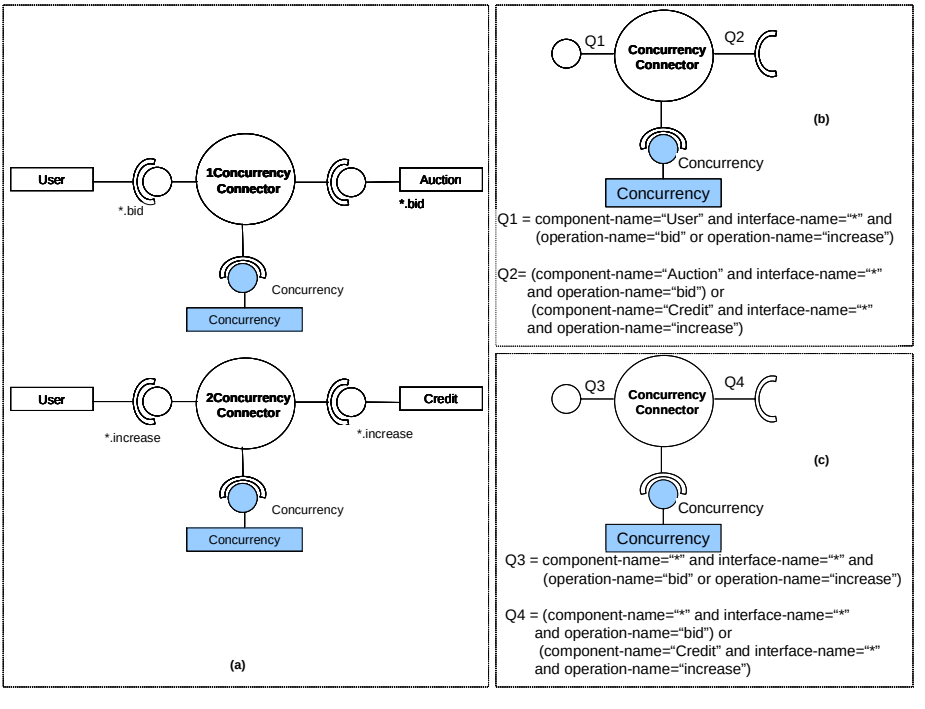


Fig. 18. a Initial mapping of the RDL concurrency concern; **b** Mapping in **a** refined with AO-ADL wildcard-based queries; **c** Mapping of the RDL concurrency composition

that have `bid` in their provided interfaces and the `Credit` component that has `increase` in its provided interface. The specification of the component and aspectual bindings, as well as the mapping of the operator and the outcome of the RDL composition, has been omitted in Fig. 18 due to lack of space.

Notice that the main difference among Fig. 18b and Fig. 18c is that the former identifies `User`, `Auction` and `Credit` as the components participating in the interactions, while the latter does not identify particular components. The election of one of these approaches is a typical problem in AOSD [70], where the use of quantifiers has to be carefully considered to avoid either that (1) very specific pointcut specifications need to be rewritten as the application evolves to be able to capture new join points that were not initially considered, or (2) very generic pointcut specifications that capture unexpected join points.

Notice that the information generated with the mapping process, shown in Fig. 18, is a high-level architecture design of concurrency. In a later refinement the software architect may generate a lower-level vision of concurrency, providing more information about, for instance, the synchronization policy. Otherwise, these details may also be postponed until the system design phase.

6.2 Discussion

In developing the work discussed in this paper, we have encountered a number of difficulties, some of which are discussed below.

6.2.1 RDL Related Issues

The most significant problem is that of automation of the natural language processing. While we have based our work on an existing natural language processor with a very high tagging accuracy (97% for part of speech, and 93% for semantic tagging), we have found that the general natural language corpus alone is not sufficient for adequate treatment of the domain-specific knowledge. Problems arise due to use of domain-specific terminology which reduces the precision of the general language based semantic tagger. This problem could be resolved if the semantic tagger can be trained on the domain-specific corpus, however this would not always be possible, and would also be a costly exercise. Another alternative is to complement the semantic tagger with a domain-specific lexicon. This is the approach we have taken with identification of non-functional concerns, as detailed in [57]. We believe it is worth developing lexicons for domains that can be reused in numerous projects, such as for instance, security or dependability related lexicons.

Another related area of difficulties is handling project-specific multi-word terms. General semantic taggers treat each word separately; general multiword expressions (e.g., look for) are detected as units since they are specifically searched for. On the other hand, project-specific multi-word terms are not identified as complete units. This problem can be resolved if text chunking tools complement the semantic taggers and domain-specific vocabulary. This work will constitute part of our future toolset development.

In Sect. 5 we have presented guidelines for mapping the RDL annotated requirements onto their corresponding AO-ADL counterparts in architecture. However, as noted in the guidelines, and reiterated in the RDL to AO-ADL metamodel mapping

in Table 8, there is no clear one-to-one mapping of elements. Presently some of the mapping guidelines require consideration of the element semantics, for instance, in deciding if a subject/object should map to a component or an attribute, one needs to consider if this subject/object represents an independent real-world entity, or some characteristic of an entity. Substantial parts of such analysis could be automated via use of additional part of speech tags (e.g., to distinguish between tangible and abstract nouns), detailed domain ontology, or other techniques. Indeed, this is one of the main directions of our future work. Nevertheless, we do not expect that a completely automated mapping process will be feasible, or even desirable. This is because, for instance, often many possible mapping alternatives will be available and a judgement will be required on the preferred one. We expect that at best a guided automated mapping will result.

Another issue currently encountered while realising mapping is that of missing information on identification of individuals. We already have sufficient information from the Wmatrix induced annotation to formulate rules for distinguishing between references to types and instances. For instance, use of both plural and singular forms for the same noun (e.g., auction and auctions) indicates that the software system should operate on both instances (i.e., individuals) and collections (i.e., types). Similarly, use of "a" determiner reflects reference to a type, while "the" refers to an instance. These rules could be included into a mapping supporting tool. However, presently we are still investigating means of automation for identifying and enforcing anaphoric references (which can be considered the natural language equivalents of the bound variables). This is a complex task, mainly due to the context dependent nature of such references. For instance: "The pen lay next to the book. Alan took it". Here "it" is an anaphoric reference, which could refer either to "pen" or to "book", depending on the previous context: was Alan looking for pen or for a book?

In AO-ADL the distinction between types and instances is done in the "attachment" section (not discussed in this paper for lack of space). At the connector level roles can have a "multiplicity" (1, n, n..*) that indicates the number of components of the same type that can be connected to the given role. In this paper we omit this information and by default assume multiplicity "1".

In the present paper we have assumed that there is no project-specific ontology, mainly because we did not have such an ontology to use in our work. However, if an ontology is available, it will be an additional input for both composition specifications in RDL and the mapping. In particular, the RDL composition queries can be defined not only in terms of the actual requirements and their SRO content, but also in terms of the ontological relationships of these SRO elements. For instance a semantic query stating (subject="user" and relationship = "is a") could be defined with "is a" ontological relation. Of course, from the perspective of automation support, the specific ontology relations may require implementation of specific operators.

Similarly, the mapping of the SRO will be complemented by the mapping of their ontological relations. For instance the normal "is a" relation between ontology terms (e.g., seller is a user) could map to inheritance, while "account number is a part of account" could map to a containment, etc. We have not defined specific guidelines for such mappings, as the mapping between elements will depend on the semantics of the specific ontology relationships.

It is worth noting that the details in the first draft of the architecture will depend on the details and structure of the requirements specification used for the initial mapping. For instance, if the use-case based structure is used, the mapping will result in an architecture outline with clear differentiation of the interface between the computer system to be developed and its users. These users could then be considered to be completely external to the software system, or made a part of it by including their software representations. However, since we do not make any formal restriction on the type or content of requirements documents used as input for COMPASS, we cannot expect to have the responsibilities of the software system and environment clearly defined prior to the mapping. Instead, we suggest that upon mapping elements can be qualified as being within the software system boundary, or outside of it. Moreover, in the present work we discuss the mechanism for transferring the information from the requirements representation to architecture, producing only a very rough draft of the intended architecture. When refining this draft, the architect needs to address a number of additional questions about the architecture improvement and, verification and validation, such as, for instance, verifying that the states and events represented in the architecture sufficiently accurately represent those of the real-world users, etc. These issues are not addressed in the present paper.

6.2.2 AO-ADL Related Issues

The generation of an architectural specification that realises both quality attributes, and architectural design goals from a given requirement specification, is not a trivial task. Our aim is to produce an initial version of the architecture that includes all the relevant concerns and relationships captured during the requirements specification. This means that the software architect does not have to deal with architecture specification from scratch. Also, the mapping from requirements to architecture does not have to be ad hoc. As we stated in Sect. 2, several architectural design strategies are in play during the architecture definition, so the output of the mapping process is not the definitive architecture.

This preliminary architecture provides the initial set of elements that should form a part of the architecture, as well as their interactions. Therefore, in order to produce a satisfactory architecture design of the system, it is necessary to carry out a refinement or rebuilding process of the initial architecture.

On the one hand, the influence that crosscutting concerns identified during the requirement elicitation have in later phases of the software lifecycle frequently depends on the nature of the particular concerns [46, 54]. Examples of these influences for well-known aspects at the requirement level can be found in [54], which show, for instance, that although both the Security and Availability concerns are aspects at requirements level, their mapping to later phases differ. Thus, while Security is usually mapped to an aspect and has influences at both architecture and design levels, Availability is usually mapped to an architecture decision and has influence only at the architecture level. These differences can only be understood on the grounds that the software architect knows the nature of these concerns and understands what a secure and available system implies for each particular case, e.g., taking into account domain knowledge, resource availability, client needs, etc. Consequently, it is very difficult to account for such variability while defining a

systematic mapping process from requirements to architecture. Thus, such issues are addressed as part of the refinement of the initial mapping.

On the other hand, as mentioned before, the level of details in the architecture largely depends on the level of details of requirements. Typical information that usually needs to be refined/completed by the software architect is: (1) the names of components, operations and attributes to provide names that clearly identify the role played by that element in the architecture, (2) the grouping of several operations in significant provided and required interfaces, (3) the identification of repeated or synonymous operations and/or interfaces, and (4) the definition of the parameters of operation, among others. As an example, in the scenarios described above we have seen how the mapping generates, for instance, operation names such as `has`, which does not provide significant information about the purpose of the operation. However, this operation is related with "a bidder having funds" and thus, renaming to `has_fund` would be required.

Even more important are those situations where the systematic mapping results in ambiguities or in unclear parts of the software architecture. For instance, if the requirements in a particular concern refer to attributes, but the components containing those attributes are not mentioned in the given concern and need to be identified later. Another example of an ambiguous situation is when a set of relationships of a concern maps to operations that should be part of the interface of a component, but it is unclear if these are services provided by this component or required by it. Examples of such situations are present in the validate bid scenario described above.

We expect that some of these mapping weaknesses will be resolved in our future work. Yet questions will remain that could not be resolved without the experience, and therefore participation, of either a requirement engineer or a software architect. Thus, it is necessary to complement the mapping process with tools to aid a software architect with visualising and verifying properties of the AO-ADL architectural specification. Presently, we are modifying a component and aspect repository tool and an architecture description tool [25] previously developed for DAOP-ADL to use it for AO-ADL. The component and aspect repository tool will allow us to have a repository of COTS components and aspects that can be reused in different software architectures. The architecture description tool will avoid the manipulation of XML, allowing the editing, modification and verification of software architectures. Both tools are linked, allowing to import previously developed components and aspects into the description of a new architecture. Moreover, we are planning to extend the architecture description tool to provide traceability information from requirements to architecture. This will allow the architect to review the mapping results and link back from architecture to requirements specification if some ambiguities or errors are identified. Thus, the tool will link parts of the architecture generated via the mapping process with the relevant RDL specification.

6.2.3 Mapping Patterns

In this paper we have discussed how the RDL annotated requirements can be mapped to the architecture. The eventual goal of this work is to define reusable patterns on the RDL to ADL mapping built around the RDL verb types. The above presented SRO pattern mapping realizes the first stage of such pattern building. To provide an

indication of the types of such mapping patterns, we show two examples from our ongoing work in this direction.

Example 1: Mapping of Order type
The Order type verbs contain the Speaker, Addressee, and Message main roles, where message can be a complex structure itself. Mapping of this verb type is illustrated in Fig. 19 via a requirement stating that "`Customers ask` (Order) `the system to debit` (Modify) `a certain amount from their credit card`". In this requirement, we have the `ask` Order type verb with `customer` acting as the Speaker, and `system` as the Addressee. The Message generally can be a direct speech, a THAT complement clause, or a (FOR) TO complement clause. In the present case the Message is a TO complement clause: "`to debit` (Modify) a `certain amount from their credit card`". Message itself has a verb "`debit`" of Modify type which requires the Agent role (which here is played by `system`) changing the state of the Target role (played by `credit card`) using a Manip role (played by `amount`). Note that, though the Message does not have "`system`" mentioned directly, we have identified `system` as the Agent because in Order verb types when the subject is missing in the Message, it is coreferential with the Object of the main clause, i.e., the `system` in our example. The mapping of the Order type verbs is outlined in Fig. 19, where connectors are omitted for simplicity:

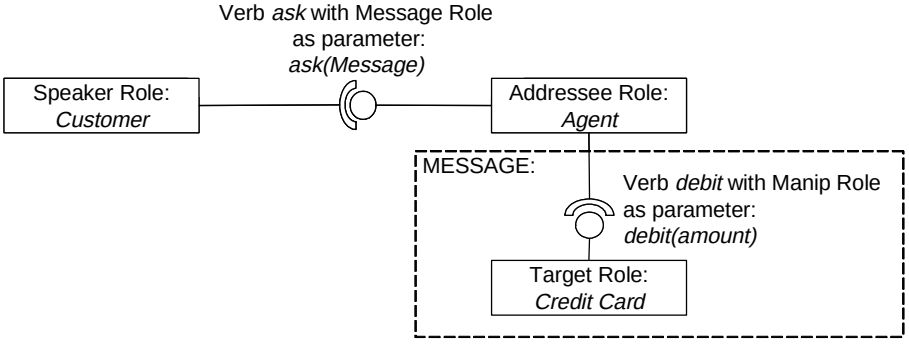


Fig. 19. Mapping of Order verb type and roles

As a result of the Order verb (ask) mapping, the Speaker has communicated a Message to the Addressee. As a result of Modify verb mapping from the Message (debit) the state of the Target has been changed by the Agent (which is the same as the Addressee of the previous Order type verb, i.e. the system) using the Manip (amount).

Example 2: Mapping of Transfer Possession type
This is illustrated with mapping of requirement stating: "`users provide their credentials to the auction system`", we have the Transfer Possession type (`provide`) with defined roles: the Donor (`user`) which provides, the Gift (credentials) to the Recipient (`auction system`). This would correspond to an

architectural pattern with two components and one operation with one parameter, the parameter being the Gift, as shown in Fig. 20.

As a result of the Transfer Possession verb type (`provide`) a Gift (`credentials`) has been passed from the Donor (`user`) to the Recipient (`system`).

Definition and validation of such patterns is the main direction of our mapping work.

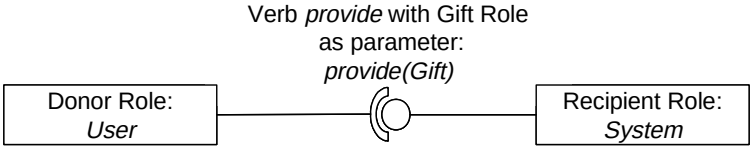


Fig. 20. Mapping of Transfer Possession verb type and roles

7 Related Work

The COMPASS mapping guidelines are in the same spirit of aspect traceability as the aspectual use case mappings [31], mapping of themes from requirements to design [18] and the proof obligations derived from aspectual requirements and design in PROBE [35].

The AOSD with use cases approach [31] treats *extend*, *include* and *infrastructure* use cases as aspects and allocates them to architecture per class as in traditional OO mapping. The issue of crosscutting in architecture is accounted for once an OO architecture has been established. In contrast, COMPASS facilitates clear compositional reasoning about the relationships among requirements-level aspects [53] and provides aspect aware mappings resulting in an architecture fully informed about crosscutting influences and their potential dependencies, trade-offs and interactions.

Theme [18] provides a direct mapping between requirements themes and design themes. However, this mapping is not driven by composition as in COMPASS — Theme defers composition to design. As shown in Sect. 5.3, COMPASS can be used to complement Theme by representing Theme/Doc themes as RDL concerns with the subsequent transition to an AO-ADL architecture using the mapping guidelines before a detailed Theme/UML design is derived.

Though PROBE [35] aims to link aspectual requirements and designs to concrete proof obligations about an implementation, the focus is on independently deriving proof obligations from requirements and design and integrating them. In contrast, COMPASS focuses on systematic mapping of aspectual requirements to architectures. As such we can extend COMPASS to provide architecture-to-design and design-to-implementation mappings (or exploit the Theme/UML mappings for the latter) with PROBE used to derive the associated proof obligations to validate the implementation resulting from the mappings.

COMPASS can also be employed for iterative and concurrent requirements and architecture derivation as in the Twin Peaks model [46]. The mapping can be started

as soon as a basic RDL specification is available with both the RDL and AO-ADL specifications refined as more requirements are added and existing ones refined.

Our approach also bears relationship with goal-oriented requirements engineering approaches [32, 33, 39] and architecture derivation in such approaches. In contrast to the mappings in [32, 33, 39] which are based on initial rigorous formal representation of the requirements with KAOS, our mapping is based on the correspondence of the RDL and AO-ADL elements, especially their composition semantics. While the more formal specification and mapping could be quite desirable in some circumstances, in others it may incur unacceptable costs and effort. Besides, the starting points of these two approaches are rather different: goal-oriented approaches begin with goal analysis, while we start with a set of natural-language requirements (e.g., using manuals, interview transcripts, for initial requirements structuring). Thus, we consider the two approaches to be complementary, aiming for different development circumstances.

Our work can also be considered closely related to the work on Domain Theory [65] in that both these efforts try to identify patterns in requirements which could assist in architecture derivation. The main difference between these efforts is that the Domain Theory turns to the repository of previously developed software for recurrent task and pattern identification, while we are looking at such task identification based on the natural language semantics and structure. When we have completed work on our pattern identification, the detailed comparison of patterns from both approaches will form the next topic of our study.

In [8] the focus is on defining a set of *architectural tactics* which help to manipulate some parts of the quality attribute model. Tactics can be dependent on each other, as choice of one alterative will often restrict/dictate a specific choice for another. In [11] the authors investigate initial ideas of expressing tactics and architectural reasoning elements in terms of AOSD concepts. The work presented in [8, 11] mainly attends to ensuring that the quality-related requirements are realised in a certain architecture. Thus, it focuses only on mapping of the quality requirements to architecture, assuming that the other (i.e., functional) architectural elements are already defined. This approach is complementary to ours, as we support the derivation of the initial set of architectural elements which can then be elaborated into a quality-attribute-focused architecture by applying the tactics of [8].

The CBSP approach [23, 28] describes a set of steps and a refined requirement representation language which guide the derivation of an intermediate model which bridges the requirements and architectural activities. This approach is similar to COMPASS in that it helps in identification of the initial architectural elements and their relationships from the requirements. However, it requires requirements to be classified and refined into a single architectural element category which is a skill-intensive manual task aimed to assist the mapping. In our case such mapping does not require additional preliminary preparation, but is facilitated through direct correspondence of RDL and AO-ADL meta-models and the mapping guidelines. Besides, the end result of our approach is expressed in the architectural-level terms (i.e., AO-ADL constructs), rather then the intermediate boxed requirements notation used by [23, 28].

Aspect-Oriented Component Engineering (AOCE) [29, 30] extends the JViews approach to incorporate aspects. Unlike our approach, AOCE does not propose a

systematic mapping process and guidelines to go from requirements to architecture. Instead it relies on the components and aspects that are identified at the requirements-level to be present throughout the rest of development phases.

8 Conclusion

In this paper we have presented COMPASS, an approach that supports modular representation of the requirements-level interactions and dependencies between concerns and their systematic mapping to aspect-oriented architectures. The means supporting our *composition-centric mapping* realization are our RDL, AO-ADL, and the mapping guidelines relating them. Our requirements and architecture description languages capture the semantics of requirements and architecture level abstractions respectively; while the guidelines have been derived and validated through several practical mapping case studies.

The key contributions of COMPASS are: (1) the compositional information provided by the RDL, (2) the extended semantics of our AO-ADL connectors to specify aspectual compositional information, and (3) mapping compositional patterns of requirements to patterns of architectural connections.

Concretely, using the RDL and its compositional information, COMPASS defines a mapping process in which not only information about the crosscutting nature of concerns is mapped to architecture, but also the precise information about their influence on other architectural concerns is captured. Moreover, this influence has a direct mapping onto AO-ADL connectors, which is the natural place to specify component interactions in an ADL, even when some of the components have a crosscutting effect on others.

In supporting the modular representation of requirements compositions and their corresponding ADL bindings, we take another step towards supporting *modular reasoning* about crosscutting concerns during requirements engineering and architecture design. But more than that, the semantics-based operators of COMPASS compositions provide key insights into its *compositional reasoning* features [53]: by building on our understanding of concern interrelationships and the roles played by concerns participating in these relationships we aim to develop a catalogue of interactions patterns which can be detected at the requirements analysis stage, mapped to their architectural counterparts and realised in implementation. This will form the focus of our future work.

In working towards this goal, in the present paper (in addition to the mapping of compositional information) we set out the mapping of a major RDL pattern: the set of Subject-Relationships-Object elements. The SRO pattern bears the main semantic load of a requirement stated in natural language. We have presented how it is represented in RDL, and mapped to operations, interfaces, components and architectural interaction patterns in AO-ADL.

Acknowledgments

This work is supported by European Commission FP6 Grant AOSD-Europe: European Network of Excellence on AOSD (IST-2-004349) and UK Engineering and Physical

Sciences Research Council (EPSRC) Grant MULDRE; Multi-Dimensional Analysis of Requirements-level Trade-offs (EP/C003330/1), and Spanish Commission of Science and Technology; contract/grant number: TIN2005-09405-C02-01.

References

[1] IBM Patterns for e-business (December 2006), http://www.128.ibm.com/developerworks/patterns/
[2] W3C XSL Transformations (XSLT) (1999), http://www.w3.org/TR/xslt
[3] ACME, Carnegie Mellon University, and Dave Wile at USC's Information Sciences Institute, USA (2006), http://www.cs.cmu.edu/ acme/
[4] British National Corpus, Oxford University Computing Services, UK (2006), http://www.natcorp.ox.ac.uk/
[5] SketchEngine, Lexical Computing Ltd (2006), www.sketchengine.co.uk
[6] WMATRIX, Lancaster University, UK (2006), http://www.comp.lancs.ac.uk/ucrel/wmatrix/
[7] Allen, R., Dounce, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
[8] Bachmann, F., Bass, L., Klein, M.: Deriving Architectural Tactics: A Step Towards Methodical Architectural Design, Carnegie Mellon University & Software Engineering Institute, Pittsburgh, PA, Technical Report CMU/SEI-2003-TR-004 (March 2003), http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tr004.pdf
[9] Baniassad, E., Clements, P., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering Early Aspects. IEEE Software 23(1), 61–70 (2006)
[10] Barais, O., Cariou, E., Duchien, L., Pessemier, N., Seinturier, L.: TranSAT: A Framework for the Specification of Software Architecture Evolution. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, Springer, Heidelberg (2004)
[11] Bass, L., Klein, M., Northrop, L.: Identifying Aspects Using Architectural Reasoning. In: Workshop on Early Aspects (held with AOSD 2004), Lancaster, UK (2004)
[12] Berry, D.M.: Natural Language and Requirements Engineering - Nu? In: International Workshop on Requirements Engineering, Imperial College, London, UK (2001)
[13] Chitchyan, R., Pinto, M., Fuentes, L., Rashid, A.: Relating AO Requirements to AO Architecture. In: Workshop on Early Aspects (held with OOPSLA 2005), San Diego, California, USA (2005)
[14] Chitchyan, R., Rashid, A.: Tracing Requirements Interdependency Semantics. In: Workshop on Early Aspects (held with AOSD'06), Bonn, Germany (2006)
[15] Chitchyan, R., Rashid, A., Rayson, P., Waters, R.: Semantics-Based Composition for Aspect-Oriented Requirements Engineering. In: Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), Vancouver, British Columbia, Canada, pp. 36–48. ACM Press, New York (2007)
[16] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of (Aspect-Oriented) Analysis and Design Approaches, Lancaster University, Lancaster, Survey Report AOSD-Europe-ULANC-9 (May 2005), http://www.aosd-europe.net/
[17] Chitchyan, R., Sampaio, A., Rashid, A., Sawyer, P., Khan, S.: Initial Version of Aspect-Oriented Requirements Engineering Model, Lancaster University, Lancaster AOSD-Europe project report (D36) No: AOSD-Europe-ULANC-17 (February 2006), http://www.aosd-europe.net/

[18] Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: the Theme Approach. Addison-Wesley, Reading (2005)
[19] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures. Addison-Wesley, Reading (2002)
[20] Dashofy, E.M., Hoek, A.v.d., Taylor, R.N.: An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In: 24th International Conference on Software Engineering (ICSE2002), Orlando, Florida, USA, May 19-25, 2001, pp. 266–276 (2002)
[21] Dijkstra, E.W.: On the role of scientific thought, vol. EWD 447. Springer, New York (1982)
[22] Dixon, R.M.W.: A Semantic Approach to English Grammar, 2nd edn. Oxford University Press, Oxford (2005)
[23] Egyed, A., Grunbacher, P., Medvidovic, N.: Refinement and Evolution Issues in Bridging Requirements and Architecture - the CBSP Approach. In: From Software Requirements to Architecture Workshop (held with ICSE 2001), Toronto, Canada, May 12-19, 2001, pp. 42–47 (2001)
[24] Fuentes, L., Gámez, N., Pinto, M.: DAOPxADL: An Extension of the xADL Architecture Description Language with Aspects. In: DSOA'06 workshop (held with JISBD'06), Barcelona, Spain (2006)
[25] Fuentes, L., Pinto, M., Troya, J.M.: Supporting the Development of CAM/DAOP Applications: an Integrated Development Process. Software Practice and Experience 37(1), 21–64 (2007)
[26] Garcia, A., Chavez, C., Batista, T., Sant'Anna, C., Kulesza, U., Rashid, A., C.J.P.d.L.E.: On the Modular Representation of Architectural Aspects. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 82–97. Springer, Heidelberg (2006)
[27] Garlan, D., Monroe, R.T., Wile, D.: ACME: Architectural Description of Component-Based Systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press, Cambridge (2000)
[28] Grunbacher, P., Egyed, A., Medvidovic, N.: Reconciling Software Requirements and Architecture: the CBSP Approach. In: First IEEE International Symposium on Requirements Engineering (RE'01), pp. 202–211. IEEE Computer Society Press, Los Alamitos (2001)
[29] Grundy, J.: Multi-perspective Specification, Design and Implementation of Software Components using Aspects. International Journal of Software Engineering and Knowledge Engineering 20(6), 713–734 (2000)
[30] Grundy, J.: Supporting Aspect-Oriented Component-Based Systems Engineering. In: International Conference on Software Engineering and Knowledge Engineering, pp. 388–395. KSI Press (1999)
[31] Jacobson, I., Ng, P.-W.: Aspect-Oriented Software Development with Use Cases. Addison Wesley Professional, Reading (2005)
[32] Jani, D.: Deriving Architecture Specifications from Goal Oriented Requirement Specifications, University of Texas at Austin, Masters Thesis (May 2004), http://users.ece.utexas.edu/ perry/work/papers/R2A-DJ-thesis.pdf
[33] Jani, D., Vanderveken, D., Perry, D.E.: Deriving Architecture Specifications from KAOS Specifications: A Research Case Study. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 185–202. Springer, Heidelberg (2005)
[34] Kandé, M.M.: A Concern-Oriented Approach to Software Architecture, PhD Thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland (2003), http://infoscience.epfl.ch/getfile.py?mode=best&recid=54726

[35] Katz, S., Rashid, A.: From Aspectual Requirements to Proof Obligations for Aspect-Oriented Systems. In: International Conference on Requirements Engineering (RE'04), pp. 48–57. IEEE Computer Society Press, Los Alamitos (2004)
[36] Krechetov, I., Tekinerdogan, B., Pinto, M., Fuentes, L.: Initial Version of Aspect-Oriented Architecture Design Approach, 2006, University of Twente: Twente. AOSD-Europe Deliverable D37, AOSD-Europe-UT-D37 (February 2006)
[37] Kulesza, U., Garcia, A., Lucena, C.: Generating Aspect-Oriented Agent Architectures. In: Workshop on Early Aspects (held with AOSD 2004), Lancaster, UK (March 22, 2004)
[38] Kulesza, U., Garcia, A., Lucena, C., Staa, A.v.: Integrating Generative and Aspect-Oriented Technologies. In: 19th ACM SIGSoft Brazilian Symposium on Software Engineering, Brazil (October 3-7, 2005)
[39] Lamsweerde, A.v.: From System Goals to Software Architecture. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 25–43. Springer, Heidelberg (2003)
[40] Levin, B.: English verb classes and alternations: a preliminary investigation. The University of Chicago Press, Chicago (1993)
[41] Lopes, C.V.: D: A Language Framework for Distributed Programming. PhD Thesis, College of Computer Science, Northeastern University, Boston, USA (1997), http://www.isis. vanderbilt.edu/projects/core/bibliography/core/AOPdistribProg.pdf
[42] Medvidovic, N., Taylor, R.N.: A classification and Comparison Framework for Software Architecture Description Languages. IEEE Transitions on Software Engineering 26(1), 70–93 (2000)
[43] Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: 22nd International Conference on Software Engineering (ICSE'00), Limerick, Ireland, June 4-11, 2000, pp. 178–187 (2000)
[44] Moreira, A., Araujo, J., Rashid, A.: A Concern-Oriented Requirements Engineering Model. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 293–308. Springer, Heidelberg (2005)
[45] Moreira, A., Araujo, J., Rashid, A.: Multi-Dimensional Separation of Concerns in Requirements Engineering. In: 13th International Requirements Engineering Conference (RE'05), Paris, France, August 29 - September 2, 2005, pp. 285–296 (2005)
[46] Nuseibeh, B.: Weaving Together Requirements and Architectures. IEEE Computer 34(3), 115–117 (2001)
[47] Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E.: PRISMA: towards quality, aspect-oriented and dynamic software architectures. In: Third International Conference on Quality Software (QSIC'03), Dallas, Texas, USA, November 6 - 7, 2003, pp. 59–66. IEEE Computer Society, Los Alamitos (2003)
[48] Pessemier, N., Seinturier, L., Duchien, L.: Components, ADL and AOP: Towards a Common Approach. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, Springer, Heidelberg (2004)
[49] Pinto, M., Fuentes, L., Troya, J.M.: DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. In: Pfenning, F., Smaragdakis, Y. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 118–137. Springer, Heidelberg (2003)
[50] Pinto, M., Fuentes, L.: AO-ADL: An ADL for describing Aspect-Oriented Architectures. In: Early Aspect Workshop (held with AOSD'07), Vancouver, Canada (March 13, 2007)
[51] Pinto, M., Gámez, N., Fuentes, L.: Towards the Architectural Definition of the Health Watcher System with AO-ADL. In: Early Aspect Workshop (held with ICSE'07), Minnesota, USA (May 2007)
[52] Quirk, R., et al.: A Comprehensive Grammar of the English Language. Longman, New York (1985)

[53] Rashid, A., Moreira, A.: Domain Models are NOT Aspect Free. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 155–169. Springer, Heidelberg (2006)
[54] Rashid, A., Moreira, A., Araujo, J.: Modularisation and Composition of Aspectual Requirements. In: 2nd International Conference on Aspect-Oriented Software Development (AOSD'03), pp. 11–20. ACM Press, New York (2003)
[55] Rayson, P.: UCREL Semantic Analysis System (USAS) (2005), http://ucrel.lancs.ac.uk/wmatrix.html
[56] Sampaio, A., Chitchyan, R., Rashid, A., Rayson, P.: EA-Miner: a Tool for Automating Aspect-Oriented Requirements Identification. In: 20th Automated Software Engineering Conference (ASE'05), Long Beach, California, USA, November 7-11, 2005, pp. 352–355 (2005)
[57] Sampaio, A., Rashid, A., Chitchyan, R., Rayson, P.: EA-Miner: Towards Automation in Aspect-Oriented Requirements Engineering. In: Transactions on Aspect-Oriented Software Development, 2007 (accepted for publication)
[58] Sawyer, P., Rayson, P., Cosh, K.: Shallow Knowledge as an Aid to Deep Understanding in Early Phase Requirements Engineering. IEEE Transactions on Software Engineering 31(11), 969–981 (2005)
[59] Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. John Wiley and Sons, Chichester (2000)
[60] Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20(5), 19–25 (2003)
[61] Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Software 20(5), 42–45 (2003)
[62] Shaw, M., DeLine, R., Zelesnik, G.: Abstractions and Implementations for Architectural Connections. In: Third International Conference on Configurable Distributed Systems (ICCDS '96), pp. 2–10 (1996)
[63] Sommerville, I.: Software Engineering, 7th edn. Addison-Wesley, Reading (2004)
[64] Sommerville, I., Sawyer, P., Viller, S.: Viewpoints for requirements elicitation: a practical approach. In: International Conference of Requirements Engineering (ICRE'98), Kyoto, Japan, April 19-25,1998, pp. 74–81. IEEE Computer Society, Los Alamitos (1998)
[65] Sutcliffe, A.: The Domain Theory: Patterns of Knowledge and Software Reuse. Lawrence Erlbaum Associates, Mahwah (2002)
[66] Tarr, P.L., Ossher, H., Harrison, W.H., Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: 21st International Conference on Software Engineering (ICSE 1999), Los Angeles, CA, USA, May 16-22, 1999, pp. 107–119 (1999)
[67] Tekinerdogan, B.: ASAAM: Aspectual software architecture analysis method. In: 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04), Oslo, Norway, June 12-15, 2004, pp. 5–14. IEEE Computer Society Press, Los Alamitos (2004)
[68] Waters, R. W.: MRAT - The Multidimensional Requirements Analysis Tool, MSc. Thesis, Computing Department, Lancaster University, Lancaster (2006)
[69] Whittle, J., Araujo, J.: Scenario Modeling with Aspects. IEE Proceedings - Software (Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design) 151(4), 157–172 (2004)
[70] Wloka, J.: Towards Tool-supported Update of Pointcuts in AO Refactoring. In: Linking Aspect Technology and Evolution Workshop (held with AOSD'06), Bonn, Germany, March 20-24, 2006, pp. 20–24 (2006)

Appendix A Summary of Mapping Guidelines

Our mapping guidelines (discussed in Sect. 5) can be generalised and summarised as follows:

Appendix A:1 Concern Mapping Guidelines

1. Concerns from an RDL specification can be candidate simple components, composite components, interfaces, or parts of components in AO-ADL. The specific corresponding element is determined by the semantics of the concern (i.e., whether it represents a viewpoint, feature, etc.), its name, and its participation in requirements-level composition.
2. In the same way as a requirement is associated with the RDL concern in which it is defined, the architectural information obtained from the requirements is associated with the architectural representation of its RDL-level concern.
   - If a concern in the RDL is mapped to a component, the aspectual and non-aspectual components identified from the requirements of this RDL concern should be checked for being synonymous (i.e., identical) to the concern's component, being sub-components of this component, or being related to it.
   - If a concern in the RDL is mapped to an interface in AO-ADL, the operations identified from that concern's requirements should be checked for belonging to the component's interface, or for belonging to the interfaces of the components identified from the corresponding requirements of this RDL concern. Also, it should be checked if the interface to which the concern is mapped is a provided and/or required interface of the components to which subjects and objects are mapped.

Appendix A:2 Requirements Semantics Mapping Guidelines

3. From requirements we get information about *subjects*, *objects*, and *relationships*. *Subjects* and *objects* contribute to the identification of components or parts of components. *Relationships* contribute to forming the component's interfaces. Additionally, SRO patterns may contribute to the description of binding information between components.
4. *Subjects* and *objects* may be mapped to:
   - components, either as synonyms of already identified components from concerns, or potentially new components.
   - parts of components, i.e., state attributes (e.g., an auction's start and end dates) or roles (e.g., seller and bidder are roles played by users).
5. *Relationships* are mapped to operations of an interface. Each operation will be associated with an interface of the components of its corresponding subject/object. An operation is likely to be provided by the component of its object and required by the component of its subject.
   - If synonymous operations are identified they should not be repeated (i.e., they should be merged into a single operation).
   - Operations related to a subject/object mapped to a state attribute will be part of the required/provided interface of the component to which that attribute pertains, respectively.

- After processing all the relationships in the requirements of a particular concern, and identifying operations, single operations/interfaces should be grouped to form consistent provided and/or required interfaces.

Appendix A:3 Composition Mapping Guidelines

6. Constraint element mapping:
   - In the given composition the operators define the roles to be filled in by the elements selected by the query expressions of the constraint and base elements.
   - The constraint query expression maps onto one or more aspectual bindings in the AO-ADL.
     - If the query statement is expressed in terms of *relationships*, these relationships map to their corresponding *operations* in AO-ADL.
     - If the query expression is expressed in terms of *subjects* or *objects* then (i) the requirement sentences to which these subjects and objects belong should be identified; (ii) the operations derived from the relationships related to these subjects and objects must be identified, and (iii) these operations must be considered as potential advice.
     - If the query statement is expressed in terms of concerns, all operations of all interfaces of the corresponding architectural element must be investigated for being potential advice.
7. Base element mapping:
   - The query expression of the base element maps to the *pointcut* expression in the AO-ADL *composition*.
   - The *operator* maps to a *type of binding operator in AO-ADL*.
8. Outcome element mapping: An outcome element maps onto a constraint (post-condition) in the AO-ADL or onto information about the critical execution paths governed by aspectual components.

In addition, our case study has also led to some guidelines for the mapping process itself. These are summarised below.

Appendix A:4 Review Provided Interfaces

Review the identified set of provided interfaces of the components. If an interface is provided by more than one component, and those components were obtained from mapping the requirements in the same concern, a crosscutting influence is likely to be involved.

This means that when the following occur:

i) There is a concern X that is mapped to a component *x*.
ii) The X concern contains a set of requirements. From the mapping of those requirements components *y* and *z* are obtained. An interface *i* is also obtained.
iii) *i* is a provided interface of components *y/z*. *i* is also a provided interface of component *x* since the requirement from which *i* is derived is part of X.

Then, a crosscutting influence may be involved between *x* AND (*y* OR *z*).

In this situation, consider if:

iv) a separate concern with the given interface can be elicited from these concerns. In this case the newly identified concern is a crosscutting concern.

v) one of these concerns is already incorporated within the other(s). In this case either the incorporated concern is the crosscutting concern or a redundant concern or interface has been identified.

Appendix A:5 Mapping Repeated or Incomplete Requirements

While mapping RDL elements to AO-ADL, issues of repetition and incompleteness may need to be addressed.

- Requirements may be repeated within same or different concerns either by mistake (i.e., due to poor requirements structuring), or intentionally (e.g., the same requirement may be stated in different concerns from the perspective of different stakeholders). In either case, when a repetition of an already mapped requirement occurs, it would already be represented in the mapping.
- In a requirement made up from several clauses, or in a hierarchically structured requirement, pertinent information may be omitted in a clause or sub-requirement (e.g., the subject may not be repeated), when it is included into the previous clause or the parent requirement. Consequently, if missing information is found in a clause or sub-requirement its previous clause or parent requirement should be consulted. However, if the missing information cannot be obtained, it may be necessary to consult the requirements engineers or the stakeholders for further input.

As an example of such incompleteness consider a requirement from the "Sell" concern, which states that "A customer that wishes to sell initiates an auction by informing the system". This requirement has two clauses: "Customer wishes to sell" and "Customer initiates an auction". However, the customer subject is not repeated for the second clause, so it must be obtained from the first clause.

Appendix A:6 System Boundary

When following the mapping procedure all subjects and objects from requirements will be mapped to components or alike in the architecture. In order to elaborate the mapping into a realisable architecture, an architect needs to review the elements and establish a system boundary. That is, s/he needs to check if each of these components is "implementable" within the architecture or should be treated as external systems/sub-systems with which the system to be realised will interface. For example, the credit card processing in our case study is not realized by the Auction system itself but instead sub-contracted to an external credit card processing system.

An alternative is to establish the system boundary before the mapping process. This may be done for human users and other external systems that can be clearly identified from requirements as actors of the system. Anyway, even limiting the system boundary before the mapping, we still need to obtain, during the mapping process, the information about the interactions of the "software" components with users and other external systems. In both cases, after the mapping, the software architect needs to check if the external systems will provide all the information required from the system under construction and if it is going to be able to consume all the information generated by the system. Moreover, in some cases the information generated during the mapping process about an external actor may be used to "implement" a software counterpart for it. Examples of this are software representation of the customer in the auction system, or of an ATM in a bank system.

Aspects at the Right Time*

Pablo Sánchez[1], Lidia Fuentes[1], Andrew Jackson[2], and Siobhán Clarke[2]

[1] Dpto. de Lenguajes y Ciencias de la Computación,
ETSI Informática,
Universidad de Málaga Spain
{pablo, lff}@lcc.uma.es
[2] Distributed Systems Group,
Dept. of Computer Science,
Trinity College Dublin Ireland
{Andrew.Jackson, Siobhan.Clarke}@cs.tcd.ie

Abstract. At different stages of the aspect development lifecycle, there are different properties of aspects that need to be considered. Currently, there is no integrated approach to defining the appropriate characteristics of aspects at the appropriate stage, or of tracing decisions made for evolution purposes. Our focus is on the early aspects stages of development — requirements analysis, architecture design, and detailed design — where there are already many different approaches that provide useful constructs and mechanisms to capture the different properties of aspects that are in play at the relevant stage. However, it is difficult to move between stages using different approaches. In this paper, we describe an aspect mapping from requirements to architecture to design: in particular, Theme/Doc (requirements), CAM (architecture) and Theme/UML (design). The mapping includes heuristics to guide as to the right time to specify the right aspect properties. In addition, it allows aspect decisions captured at each stage to be refined at later stages as appropriate. While this provides an integrated approach for aspect specification, it is not enough to facilitate the traceability of aspect decisions. To this end, we also describe a means to record decisions that capture the alternatives considered and the decision justification. This information is crucial for managing aspect evolution at the right time.

1 Introduction

Aspect-Oriented Software Development (AOSD) [1] is a relatively new paradigm that introduces new means for modularising and composing crosscutting behaviours at each development stage. These new modularisation and composition techniques change and expand the constructs available to, and decision support for, the software engineer at each stage of the software development lifecycle.

Our focus is on the early aspects stages of development — requirements analysis, architecture design, and detailed design — where different properties of aspects that

* This work has been supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004–2008.

A. Rashid and M. Aksit (Eds.): Transactions on AOSD IV, LNCS 4640, pp. 54–113, 2007.

are in play at the relevant stage need to be captured. An aspect property is a piece of information that characterises the aspect and must be specified when the aspect is created. The relevance of aspects themselves can emerge or be diminished when moving between different stages of development. Through a comprehensive survey [2], we have identified many approaches that provide elegant solutions for addressing aspects in early phases of software development. Each has constructs and mechanisms for capturing the aspect properties of interest to that approach. However, one important outcome of the survey is a realisation that there is no integrated approach to defining the appropriate characteristics of aspects at the appropriate stage, or of tracing decisions made for evolution purposes. The software engineer has no clear guidelines on how to consistently address aspects across the early stages of the development lifecycle.

In this paper, we address this gap and describe a mapping from Aspect-Oriented (AO) requirements to architecture to design, illustrated with a concrete mapping based on: Theme/Doc [3] (requirements), CAM [4] (architecture) and Theme/UML [3] (design). Our choice of these specific approaches is driven by our previous experience developing, using and extending them. The mapping includes heuristics to guide as to the right time to specify the right aspect properties. In addition, it allows aspect decisions captured at each stage to be refined at later stages as appropriate. We map the constructs used to define aspects in Theme/Doc to those in CAM with related heuristics as to the properties to be captured along the way. Similarly, we map the constructs used to define aspects in CAM to those used in Theme/UML with corresponding related heuristics.

While this provides an integrated approach for addressing aspects in development, it is not enough to support aspect evolution. To this end, we also describe a means to record decisions that captures the alternatives considered and the decision justification. We illustrate that, used correctly, these recordings can be used as a traceability mechanism. With improved traceability support, software engineers may better assess the potential impact of any planned change. To achieve this, instances where artefacts of one phase have been derived from artefacts of the previous phase in accordance with a defined mapping are recorded. Often, when mapping some artefacts from one development phase into the next one, several choices could exist, each one corresponding to a different mapping strategy or potential solution. Decisions considered optimal at a specific time may no longer be the best choice as system environment, user needs or business rules evolve.

To concretely illustrate the mapping and the traceability support, an online auction system is used as an example throughout this paper. An initial development of this example is performed, and then, some changes over this initial system are illustrated. These changes serve to show explicitly the benefits of the previously stored traceability information when evolving the auction system.

The remainder of this paper is structured as follows. Section 2 presents the online Auction System example, and provides introduces the specific AO approaches used in this paper: Theme/Doc, CAM and Theme/UML. Section 3 introduces integration across the three phases. Section 4 illustrates: a mapping from requirements to

architecture using Theme/Doc and CAM; how traceability decisions are captured; and how this information supports change. Section 5 discusses moving from architecture to design with CAM and Theme/UML. Section 6 describes related work and Section 7 provides a discussion about our approach. Section 8 summarises the paper and outlines our future work.

2 Basis of Our Approach

In this section, we present the background for this work as well as some discussion about how to move from requirements to architecture, and from architecture to detailed design. First, we describe an Auction example that will be used throughout the paper. Then, an introduction to the Theme/Doc, CAM and Theme/UML approaches is provided. For each approach, the properties of aspects are discussed, the decisions that are needed to resolve these properties are described and constructs used to specify these properties are identified and explained. These properties are captured for each approach (where appropriate) under the high-level groupings of:

1. *Aspect identification* (identifying the behaviours that can be encapsulated as an aspect),
2. *Aspect triggering* (consideration of the base behaviours that are augmented with aspect behaviour),
3. *Aspect composition* (specification of how aspects are composed)
4. *Aspect elements* (identification of the artefacts that are encapsulated or impacted by the aspect).

These generic groups span all three stages of early aspects, and allow us consider the common motivations for the requirements analyst, the architect and the designer.

2.1 The Auction System

Motivating scenarios from an Auction System example (previously used in [5]) are used to illustrate how we can move between the early phases of the development life cycle and how we record decisions taken in requirements engineering, architecture design and detailed design in order to perform evolution management at the right time.

The Online Auction System allows subscribed users to negotiate over the buying and selling of goods in the form of English-style auctions. To participate in an Auction, a user must first join it. Once enrolled, a user may make a bid. All the other users that have joined the auction are notified about the placement of a new bid. The system has to guarantee that customers that place bids are solvent. User accounts are maintained by the Auction System. Customers can deposit and withdraw funds from their system accounts using theirs credit cards. Once an auction closes, the system decides the winner, and deposits the highest bid price minus the commission taken for the auction service into the virtual account of the seller.

Further detailed requirements relevant in this paper are presented in Table 1.

Table 1. Auction system requirements

Id	Requirement description
R1	All potential users of the system must first enroll with the system
R2	Once enrolled they have to log on to the system for each session
R3	Customers are able to log off once logged onto the system
R4	A customer can request the Auction System to auction an item that the customer offers
R5	Customers are able to make bids on auctions
R6	Customers are able to close their account
R7	Customers are able to increase their credit by asking the system to transfer a certain amount from their credit card
R8	Customers can navigate to or select to engage in auction features or browse the auctions active in the auction system
R9	The customer account balance is transferred back to their credit card when the customer closes their account
R10	A customer that wishes to offer an item for auction initiates the minimum increment price and reserve price for the offered goods, the start period of the auction, and the duration of the auction
R11	A customer has the right to cancel the offer as long as the auction's activation date has not been passed
R12	A customer can define an activation time for an offered item to go on Auction, can indicate that the Auction should be activated immediately or become activate based on some event (for example the closure or activation of another auction)
R13	Customers that wish to follow an active auction must first join the auction
R14	Once a customer has joined the auction, they may make a bid or post a message on the auction's bulletin board
R15	Customers are allowed to place their bids until the auction closes
R16	An auction closes when a duration time for the auction elapses
R17	Once an auction closes, the system calculates whether the highest bid meets the reserve price given by the seller
R18	If the highest bid meets the reserve price the system transfers the highest bid credit minus the commission taken for the auction service into the credit of the seller
R19	When the auction closes, a message is sent to the winner of the auction congratulate the winner
R20	Messages are sent to the other customers who are interested in the auction to indicate the outcome of the auction
R21	Customer information, bids, credit transfers, offers for auction must be stored in persistent containment
R22	Credit transfers must be processed in less than half of a second
R23	Bids must be processed in less than a second
R24	The auction system must be available as a website
R25	The auction system can be unavailable for a maximum of 1 h per night for maintenance and batch jobs
R26	Information displayed to customers must be available in English, German, French, Italian, Portuguese and Spanish

2.2 AO Requirements Analysis with Theme/Doc

Requirements analysis is mainly concerned with reasoning about the problem domain and formulating an effective understanding of the stakeholders' needs, either

functional or systemic, in order to provide a bridge between the problem domain and the solution domain. The requirements specification obtained from stakeholders is the basis for the derivation of a system architecture and design.

Aspect Oriented requirements analysis improves the state-of-art of traditional requirements analysis providing systematic means for the identification, modularisation, representation and composition of crosscutting properties [6]. The main task of the AO requirements analyst is to partition requirements into base and aspect concerns. Then, both kinds of identified concerns are associated with the different requirements, and the composition of these concerns is specified.

Aspect-Orientation introduces new constructs and decisions that the analyst must adopt. The Theme/Doc constructs for elaborating an AO requirements specification are described below. The properties of aspects that need to be specified as well as the decision the analyst must adopt are presented in Sect. 2.2.2.

2.2.1 Theme/Doc

Theme/Doc [3] is a requirements analysis approach that supports the identification, classification and modularisation of concerns as well as the identification of crosscutting relationships between aspect and other concerns at the requirements stage. In recognition of how the word "concern" is overloaded, the Theme Approach (which includes Theme/Doc and Theme/UML) defines a theme as the *unit of modularity* encapsulating a concern, as opposed to the concern itself. A crosscutting concern is modularised in an aspect theme and a non-crosscutting concern is modularised in a base theme.

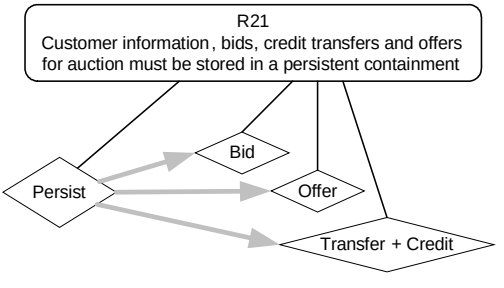


Fig. 1. Persistence theme — crosscutting view

The main elements of Theme/Doc used in this paper (illustrated in Figs. 1, 2) are:

Requirement. A requirement defines a property, constraint or feature that must be exhibited by a system in order for it to solve the business problem for which it was conceived. A requirement is represented as a rectangle. A requirement name is used to identify which requirement is being referenced. Figure 1 depicts the requirement `R21` from Table 1 expressed in Theme/Doc.

Theme. A theme is a modularisation construct that encapsulates a concern. A theme is visually depicted as a diamond that contains the name of the concern the theme represents. In Fig. 1, `Bid`, `Offer`, `Transfer+Credit` and `Persist` represent themes.

Association relationship. A requirement has an association relationship with a theme if the requirement contains some text that references the theme. An association relationship is depicted as a line that connects a requirement and a theme. In Fig. 1, `R21` is associated with `Bid`, `Offer`, `Transfer+Credit` and `Persist`.

Crosscutting relationship. A theme has a crosscutting relationship with another theme if the behaviour of the former theme is triggered by the latter. A crosscutting relationship is a directed relationship that is depicted as a heavy grey arrow between themes. The theme at which the arrow originates is the crosscutting theme and the destination of the arrow designates a crosscut theme. In Fig. 1, `Persist` is crosscutting `Bid`, `Offer` and `Transfer+Credit`.

Entity. An entity represents an object in the domain that is of importance to the theme. Entities can also represent stakeholders. An entity is represented as a rectangle. In Fig. 2, `Customer`, `Auction` and `AuctionSystem` represents entities.

Entity relationship. An entity has an association relationship with a requirement if the requirement contains some text that references the entity. An entity relationship is depicted as a line that connects a requirement and an entity. In Fig. 2, for example, `Customer` is associated with `R21`, `R10` and `R4` from Table 1.

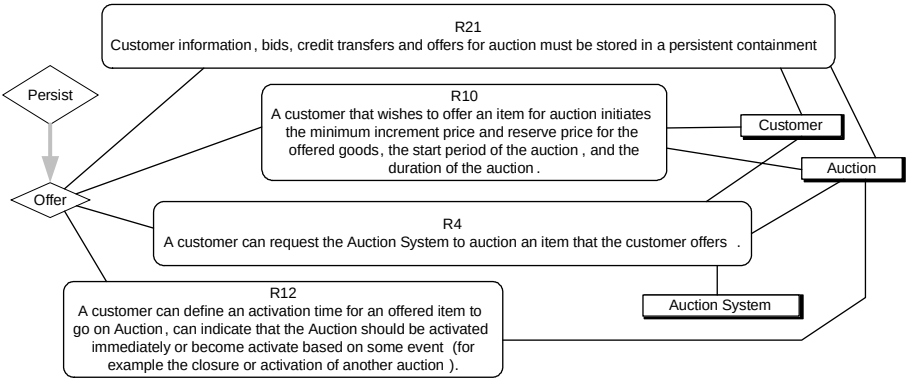


Fig. 2. Individual theme view

Theme/Doc specifications are structured in three views or diagrams that aid the analyst in the task of constructing a requirements specification:

1. *Theme relationship view*: Illustrates the relationships between requirements and themes. It contains themes, requirements and association relationships.
2. *Crosscutting view*: Illustrates the crosscutting relationships between themes. It contains themes, requirements, association relationships and crosscutting relationships.
3. *Individual theme view*: Captures the entities that are relevant for each theme. It contains themes, requirements, association relationships, entities and entity relationships.
4. *Theme group view:* For grouping themes into broader-level system goals.

Since Theme/Doc modularises requirements based on fine-grained concerns, we propose the addition of a new construct *Theme Group*, defined as a set of related themes. A Theme Group is depicted as a rectangle with two compartments, the upper for the name, and the bottom for depicting the themes the group contains. The *Theme Group View* is a new view that allows the creation of simple concern hierarchies. These hierarchies are very useful for deriving architecture components and objects in later stages of development, and help to group operations in interfaces, determine protocols and construct state-machines associated with objects. This concept is also present in other requirement analysis approaches. Use Cases [7] use `<<include>>` relationships; AORA [8] uses LOTOS-based [9] operators; (Knowledge Acquisition in Automated Specification) KAOS [10] uses graphs to decompose goals into sets of more fine-grained goals.

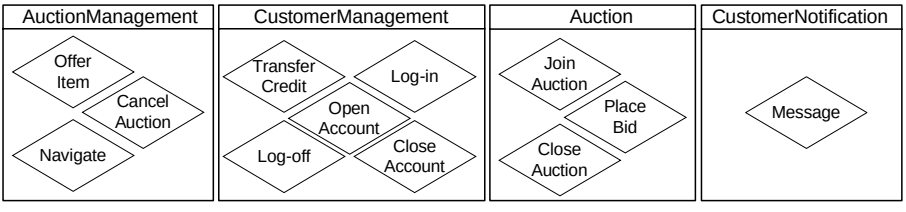


Fig. 3. Grouping themes into packages that describe broad goals

In Fig. 3, `CustomerManagement`, `Auction` and `CustomerNotification` are Theme Groups. For instance, `JoinAuction`, `PlaceBid` and `CloseAuction` are grouped into the same Theme Group, `Auction`, as they all satisfy the same broad concern of auctioning a single Item.

2.2.2 Aspect Properties at Requirements Level

Table 2 presents the properties of aspects that need to be considered at requirements level; the decisions that need to be taken; and the Theme/Doc constructs to which these decisions map.

To identify themes, the analyst considers the requirements and selects key concerns based on domain knowledge and experience. This is an iterative process, with the final set of themes emerging after further analysis of the results of each selection (i.e., the Theme/Doc views). The views support consideration of which concerns are sufficiently key to be separated into modules (or themes). After theme identification, the analyst produces a Theme Relationship view that illustrates themes and their associated requirements. Table 3, column one, contains themes identified and classified during the requirement analysis phase for the Auction System. We show only those themes which are used in examples through this paper.

After identifying themes, the first decision to be made is whether a theme is base or aspect. Candidate aspect themes are those that share requirements with other themes. For example, in Fig. 1, `Bid`, `Offer`, `Transfer+Credit` and `Persist` share `R21`.

Table 2. Properties of requirements level aspects

Property	Decision	Construct
Aspect identification: dominance over requirements	Which concern dominates a requirement?	Requirement/theme/ association relationship
Aspect triggering: triggering	Is a concern triggered by other concerns?	Requirement/theme/ crosscutting relationship
Aspect elements: related entities	What entities are related to a concern?	Theme/entity/entity relationship

Table 3. Main themes identified in Theme/Doc

Theme	**Base/aspect**
Bid	Base
Offer	Base
Persist	Aspect
Join	Base
Transfer credit	Base
Enroll/open account	Base
Availability	Aspect
Performance/processed	Aspect
Accessibility	Aspect
Multilingual	Aspect
Notification	Aspect

The first step to decide what themes are aspects is to analyse the theme's dominance over shared requirements. The analyst must decide what requirements are dominant in a specific theme, and therefore in which theme a shared requirement should be encapsulated. Where a theme encapsulates requirements that are not shared by other themes, then the theme is likely to be a base theme. On the other hand, consider the `Persist` theme, illustrated in Fig. 1, which is an example of a candidate aspect theme. Following Theme/Doc guidelines [3], we try first to rewrite `R21` as several separate requirements for persistence, but this will not remove sharing. Since the requirement cannot be usefully re-written, the analyst considers the question of dominance. The action described in `R21` is indicative of persistence and as such has a stronger relationship with the `Persist` theme than the concepts represented by the `Bid`, `Offer` and `Transfer+Credit` themes. Therefore, `R21` will be associated with `Persist`, and `Persist` is a candidate aspect that will crosscut `Bid`, `Offer` and `Transfer+Credit`.

However, a second property, the *aspect triggering*, needs to be considered before finally deciding that a theme is an aspect. In Theme/Doc, triggering of behaviour modularised in a theme by other themes is indicative of crosscutting. For instance, the `Persist` theme is triggered by behaviours represented by the `Bid`, `Offer` and `Transfer+Credit` themes. The triggering characteristic emerges from the sharing of

R21 as a direct consequence of the text in the requirement. Therefore, Persist is identified finally as an aspect that crosscuts Bid, Offer and Transfer+Credit.

Themes that share requirements with aspect themes but do not crosscut other themes are base themes[1]. The Bid, Offer and Transfer+Credit themes are an example of a base theme. Table 3, column two shows the final classification for the themes of the Auction System. The outcome of this classification will be a Crosscutting View, where crosscutting relationships are depicted explicitly as heavy grey arrows.

Finally, the analyst constructs the Individual Theme View, which helps to identify domain objects related to each theme. It is also a task of the requirements analyst to identify domain objects from initial requirements documents, because they will be required in later stages of the development process as basis for creating classes, components, and other structural elements. Figure 2 demonstrates an individual theme view for the Offer theme. The key entities in the Offer theme are the Customer, Auction and AuctionSystem while the Persist theme has a crosscutting relationship with it.

2.3 AO Architecture Design with Component and Aspect Model (CAM)

The software architecture of a system is a high-level representation of the structure of the system, which comprises software elements, the externally visible properties of these elements, and the relationships between them. The software architecture has to fulfil the system requirements, both functional and non-functional (quality attributes). Therefore, the architecture can be viewed as a set of components and their interconnections plus a set of design decisions adopted in order to meet quality attributes. An example of such a design decision is the selection of a specific architectural style in order to provide scalability. The creation of software architectures should:

1. Facilitate communication with different stakeholders,
2. Provide means for directing high-level design decisions and their evaluation,
3. Enable a problem effective partitioning, which promotes the independent development of components, and
4. Provide more opportunities for (re)use since component interfaces are defined as independently as possible.

However, some behaviour can not be adequately encapsulated following a traditional architectural component-base decomposition, hindering architecture comprehension, maintenance, evolution and component reuse. AO software architectures approaches [11–13] introduce specific mechanisms to specify how aspect behaviour is triggered at the component interactions. Aspect components are considered as units of modularisation of crosscutting of crosscutting concerns.[2]

[1] It is also possible that a base theme does not share any of its related requirements with any other theme.

[2] The term *aspect component* or *aspectual component* is used with an absolutely different meaning of the definition by Lieberherr et al. [55]. In our work an aspectual component is a component which encpasulate crosscutting behaviour. Lieberherr et al. define an aspectual component as a new component construct for programming class collaborations.

Aspect-orientation introduces new constructs and decision points that the software architect must address. The properties of aspects that need to be observed at architecture level as well as the decisions the architect must address are presented in Sect. 2.3.2.

2.3.1 Component and Aspect Model (CAM)

Component Aspect Model (CAM) [4] is a UML metamodel for describing AO software architectures. It extends concepts of Component-Based Software Engineering (CBSE) [14] with AO concepts. The main elements of CAM used in this paper are:

Components: CAM uses the Szyperski's definition of component: "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only" [14]. CAM components are depicted using common UML 2.0 notation for components. In Fig. 4, `Customer` and `Auction` are components.

Interfaces: Components in CAM have interfaces that can be provided or required. A provided interface describes those services and attributes exposed by a component. A required interface specifies context dependencies explicitly, indicating what services are required from specific components. They are depicted using common UML 2.0 notation. In Fig. 4a `IAuction` is an interface, which is required by the `Customer` component and provided by the `Auction` component.

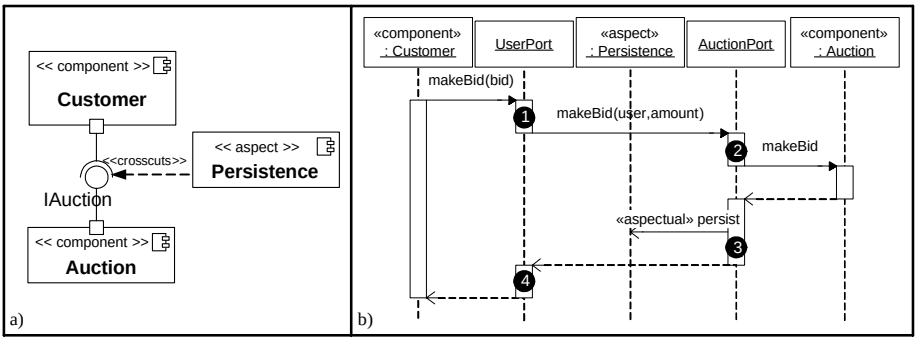


Fig. 4. Excerpt of the Auction System architecture in CAM: **a** Structural view, **b** Behavioural view

Base component composition: Composition between base components is performed by assembling provided/required interfaces. It is represented using common UML 2.0 notation. In Fig. 4a, the `Customer` and `Auction` components are connected through the interface `IAuction`.

Aspect components: These are units encapsulating crosscutting behaviours. Aspect components crosscut interactions between components. Aspect components are represented as common UML 2.0 components stereotyped as `<<aspect>>`. In Fig. 4a, `Persistence` represents an aspect component.

Messages: Components communicate by interchanging messages. CAM components never interchange messages directly. Instead, they communicate through their ports.

Messages sent to a port from outside a component are forwarded to component internals, while messages sent to the port from the inside are forwarded to the connected external components. This approach enables the sender to declare required interfaces, and to send messages to its own ports when communicating with the environment, rather than identifying an external target component directly. Communication between components is depicted using common UML 2.0 sequence diagrams. Figure 4b illustrates how the `Customer` component sends a message `makeBid` to its port, which redirects it to the adequate port of the `Auction` component. This port redirects the message to the `Auction` component internals.

Aspect composition: Following a black-box approach, aspect components in CAM crosscut only the public behaviour of components: component creation/destruction, message sending/receiving and event and exception rising. How aspects are triggered on component interactions is different for each CAM view, and explained after introducing them.

CAM models are structured in two different views:

1. *Structural view*: Details components, ports, interfaces and connections. It is depicted using UML 2.0 class diagrams, which can only contain components, aspect components, ports and interfaces.
2. *Behavioural view*: Describes components interactions and how aspects are triggered on component interactions. It is depicted using UML 2.0 sequence diagrams.

In the structural view, an aspect that crosscuts an interface can be indicated by means of a dependency stereotyped as `<<crosscuts>>` from the aspect to the crosscut interface. This relationship is optional and only serves for the purposes of drawing attention to relevant crosscutting relationships in structural views. However, this can lead to cluttered diagrams, and so `<<crosscuts>>` relationships should be used only when they are really needed.

The triggering of an aspect within a base component's interaction is indicated in the Behavioural View. A message, stereotyped as `<<aspectual>>`, from a port to an aspect, indicates when an aspect has to be executed. Aspects can be evaluated by intercepting the message delivery at the following four points: `BEFORE_SEND`, `BEFORE_RECEIVE`, `AFTER_RECEIVE` and `AFTER_SEND`, indicated with the numbers 1, 2, 3 and 4 in Fig. 4b. For instance, in Fig. 4b, the `Persistence` aspect is triggered after the `Auction` component receives the message `makeBid`. Although an aspect triggering is modelled as an explicit method call, what it actually means is that between the time when a port receives a message and when this message is dispatched, the crosscutting behaviour has to be executed. This is done using some kind of aspect-oriented technology (e.g., an aspect-oriented middleware) without the knowledge of the affected component.

2.3.2 Aspect Properties at Architecture Level

Table 4 presents the properties of aspects that need to be considered at the architecture stage; the decisions that the architect must adopt to specify these properties; and the constructs used for expressing these decisions.

Table 4. Characteristics of architecture level aspects

Property	Decision	Construct
Aspect identification: architectural aspects identification	1. How do requirements-level aspects manifest in the architecture? 2. Are there other crosscutting behaviours not identified at the requirements level?	Component, aspect component
Aspect triggering: join points designation	Where on components public behaviour is crosscutting behaviour executed?	Aspect composition (sequence diagrams)
Aspect triggering: toin point execution time	When is aspect behaviour applied relative to the point on the component's public behaviour the aspect crosscuts?	Aspect composition (sequence diagrams)

The first decision the architect must make is to define what aspect components will be in the architecture. It is possible that not all aspects identified at requirements analysis are going to map into aspect components. As we will discuss in Sect. 4.1 each requirements-level aspect may map into a base component, an aspect component, an architectural decision, or an aspect architectural decision. Additionally, some aspects not identified at requirements can appear during architecture definition. For instance, `Caching` can appear as an aspect component after selecting a client-server architectural style.

The second decision is the identification of the specific points of an application execution where crosscutting behaviour must be added. In CAM, these emerge from the public behaviour of components, as message sending/receiving and component creation/destruction and are guided by the AO requirements analysis and/or architect knowledge and experience.

The last decision is to identify when an aspect must be executed. For example, the possible times in CAM for a synchronous message sending/receiving are `BEFORE_SEND`, `BEFORE_RECEIVE`, `AFTER_RECEIVE` and `AFTER_SEND`. This decision is taken based on requirements analysis, business rules, domain knowledge and experience.

2.4 AO Design with Theme/UML

Software design is the process of specifying how the system should be implemented. Detailed structural and behavioural considerations are specified, with both systemic and functional elements from the requirements taken into account. The Unified Modelling Language (UML)[3] is the standard language for capturing object-oriented software designs. There are a number of different model types for capturing structural specifications, such as class diagrams, object diagrams and collaboration diagrams, and also for capturing behavioural specifications, such as sequence diagrams, state diagrams and activity diagrams. The process for specifying a design is largely

[3] http://www.uml.org

intellectual, with designers drawing on their domain knowledge and experience. Of course, resources like design patterns are useful, as are methodologies such as the Rational Unified Process, which is designed to be complementary to the UML.

2.4.1 Theme/UML

Theme/UML [3] is an extension to the UML for Aspect-Oriented Design (AOD). Its extensions support the modularisation and composition of crosscutting and non-crosscutting concerns in design. From a general design process perspective, the differences with standard oo design with UML are also centred on the design considerations for such modularisation and composition specification. These are additional and complementary to a standard process.

The main elements of Theme/UML used in this paper (illustrated in Figs. 5,6) are:

Base theme: Theme/UML represents non-crosscutting concerns as base themes. Base themes are extensions of UML packages. Typically, these packages contain (at least) one class diagram and sequence diagrams that describe the interactions between the classes. Standard oo design techniques are used to model the design of the requirements that are encapsulated in a theme, as analysed using Theme/Doc. Figure 5 contains abridged examples of the designs of three base themes — `Bid`, `Join` and `Offer` from the Auction System.

Base composition relationship: How to compose themes is specified using a composition relationship. Base themes may be composed using a merge composition relationship that specifies that common concepts are unified. In Theme/UML, concepts are expressed as classes, methods and attributes. The effect of merging classes and methods that represent the same concept is their unification. Figure 5 illustrates a merge composition relationship (multi-directional broken arrow) and Fig. 6 presents the resulting composite theme. The base themes subject to merge (`Bid`, `Join` and `Offer`) all contain the classes `AuctionView` and `Auction`. The methods defined on these classes (`makeBid()`, `offer()` and `join()`) are defined in different themes. In the composite theme `AuctionSystem`, the methods defined on the `Auction` classes across multiple themes are now unified in the composed `Auction` class.

Aspect theme: An aspect theme encapsulates crosscutting structure and behaviour in a package. Templates (or parameters) to the package capture a representation of operations that trigger the aspect behaviour, or other base elements affected by the crosscutting behaviour. These parameters can be referenced where required within the aspect theme, for example, in sequence diagrams where crosscutting behaviour is defined relative to the triggering behaviour. `Persist`, illustrated in Fig. 5, is an example of an aspect theme. In this example, the template parameters are `SomeObject.call(..)` and `entity`. The `SomeObject.call(..)` parameter represents points in the execution of a base theme that should be crosscut by persistence behaviour. In other words, persistence behaviour will be augmented on any base operation that binds to `SomeObject.call(..)`, as specified in the sequence diagram. `entity` represents an instance that needs to be persisted when crosscutting occurs. This means that any object that binds to `entity` will be passed as a parameter to the persistence operations. By the template names, the designer can

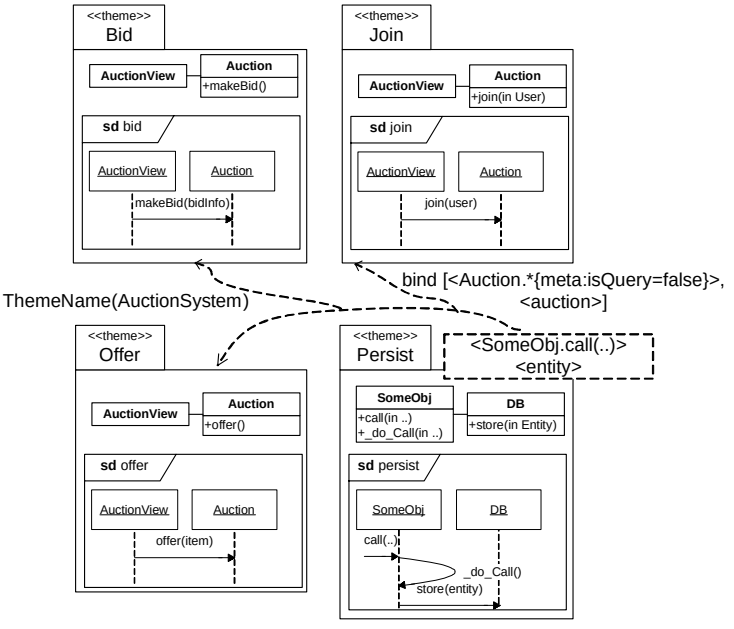


Fig. 5. Modularised theme with composition specification

specify what happens in any base elements that bind to those templates, without explicitly referencing those base elements. In general, crosscutting behaviour is triggered by base behaviour. For example, in the `persist` sequence diagram, any base operation that binds to `SomeObject.call(..)`triggers the persist crosscutting behaviour. Using the sequence diagram, the designer specifies the order of execution of the crosscutting behaviour relative to the actual base behaviour. The actual base behaviour is referred to within the sequence diagram as an operation named using the corresponding template name pre-pended with `_do_`. For example, processing the call to the real base behaviour represented by `SomeObject._do_Call(..)`occurs *before* the execution of the crosscutting behaviour, which is `DB.store(entity)`.

Aspect composition relationship: The specification of how to merge aspect themes with other themes is done with a composition relationship annotated with information related to the binding of base elements to template parameters. Figure 5 illustrates the composition specification for the `Persist` theme. Triggering behaviour to be bound to the aspect theme's exposed parameters is specified using a `bind` statement on the relationship arrow. In this example, the `bind` statement defines a selection of any call (`*`) on instances of the `Auction` class that change the state of the auction (`{meta:isQuery=false}`). It also defines a selection of the `Auction` instance as "context" (in this case, the instance to be persisted). Both the call and context are bound to the aspect theme's parameters. In the resulting composite in Figure 6 we see that each join point selected (`Auction.[makeBid(), join(), offer()]`) results in a new sequence diagram, in which the crosscutting behaviour persists the auction context.

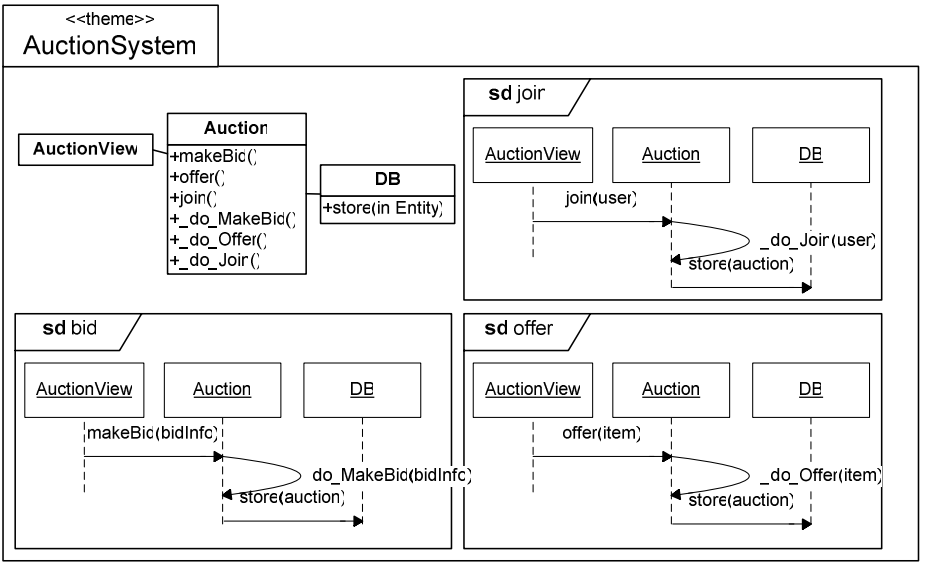


Fig. 6. Composed themes

2.4.2 Aspect Properties at Design Level

In detailed design, the designer's goal is to provide a detailed outline for a software solution that meets requirements for themes. Table 5 presents some of the characteristics of aspects that need to be considered in design to achieve this goal; the decisions that need to be taken to address these characteristics; and the constructs that these decisions map to.

Table 5. Characteristics of design level aspects

Characteristic	Decision	Construct
Aspect identification: identification of emergent aspects	Are there other crosscutting behaviours not identified at the requirements level? Are there any technical concerns that arise only in detailed design?	Base theme, aspect theme
Aspect composition: scope of composition	What concerns does the aspect affect and not affect?	Base and aspect theme; composition specification
Aspect triggering: join points that trigger aspect behaviour	What points of execution are to be crosscut?	Base and aspect theme; composition specification
Aspect composition: When an aspect should execute in relation to a join point	How the aspect behaviour is executed in relation to the join point where it is applied.	Aspect theme

One of the first decisions to be addressed by the designer is to identify all the base and aspect themes. The themes that the designer begins considering are based on those identified using Theme/Doc. However, during the design, the designer may identify emergent aspects that may be business or technical concerns. Business concerns arise due to incomplete requirements. Stakeholders may have reviewed the requirements and pointed out concerns that were not addressed. Technical concerns are concerns that only exist in the software domain. An example of a technical concern that arises in the Auction System example is `Distribution`. The DB class in the `Persist` theme represents an interface to a database. This database runs on a different server and must be accessed over a network. A `Distribution` theme is introduced to deal with the network issues.

For each aspect concern addressed in design the scope of the composition of the aspect needs to be decided. An aspect specifies behaviour that cuts across base design themes, and so the designer must select those parts of the base design that will be augmented with the aspect behaviour. This will generally be the set of theme behaviours that are required for the system configuration under design. Since Theme/UML has divided up the system design into concerns, different application configurations can be selected from the full set. In Fig. 5, the scope of the composition of the `Persist` theme is defined, using the composition relationship, as `Bid, Join` and `Offer`. This constrains the join point selection to those themes.

The designer also decides the join points that trigger the aspect behaviour. Here, the Theme/Doc models are useful as they illustrate considerable information on aspect-base theme relationships, and the sharing of entities. This information can be used to select and specify the triggering join points and the related context. In the example presented in Fig. 5, the designer needs to ensure that the `Auction` instance context can be selected at the relevant join points before the persistence behaviour can be defined.

Central to the aspect behaviour is when it should execute in relation to a join point. The designer decides this based on the requirements, or on technical considerations or experience. For example, it makes sense to persist changes only after a change has been made. Therefore, in the case of the `Persistence` aspect, the storing behaviour should execute after the join point executes. The positioning of the `_do_Call()` method in the `persist` sequence (Fig. 5) indicates this.

The composition order is another characteristic of aspects. When multiple aspects are triggered together, the designer must decide the order in which they are applied.

3 Towards a General, Integrated Process for Early Aspects

In Sect. 2 we have outlined some of the decision-making and properties of aspects addressed at each phase of the software development. Most of the current work that considers aspects at requirements, architecture and design levels [2] has been done independently, without taking into consideration how to map aspects from requirements to architecture and to design. There is no integrated approach to defining the appropriate characteristics of aspects at the appropriate stage, or of tracing

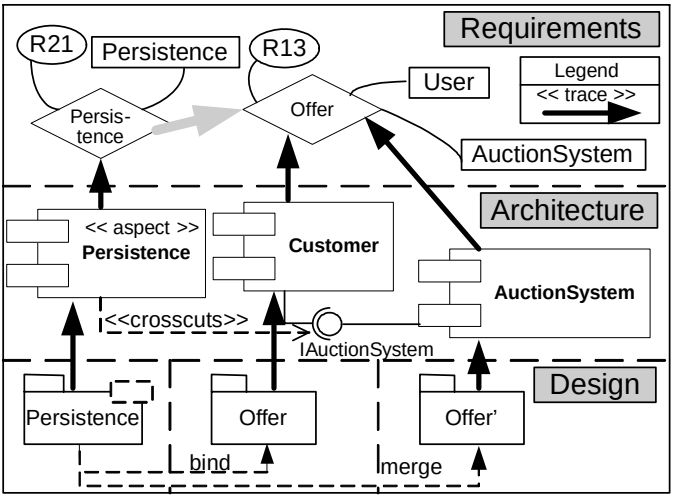


Fig. 7. Mapping from requirements into architecture, and from architecture into design

decisions made for evolution purposes. This means that the software engineer does not have clear guidelines on how to consistently address aspects across the early stages of the development lifecycle.

Although there are some differences in the identification and specification of crosscutting concerns in each development phase, it is clear that the concepts that emerge in these early stages are quite similar. Figure 7 presents a high-level view of how constructs and aspects flow through the requirements analysis (top), architecture design (middle) and design (bottom). This illustrates at a trivial level that there are commonalities in concepts across the three phases. In the following sections, we explore this further and propose a general, integrated process that captures the appropriate decision-making and provides a mapping between the stages. Further, we illustrate the benefits of integration when evolution occurs.

Aspect Oriented approaches at each stage of the development life cycle improve modularisation and composition of concerns, encapsulating crosscutting concerns in separate units. However, depending of the stage of development, the way in which we want to modularise our system may differ. At requirements analysis we are interested in identifying functionalities and their relationships. At architecture design, the definition of the main elements of the system and their interconnections are of interest. At design level, we are interested in providing a detailed description for each piece of functionality encapsulated by a component, by detailing component internals, data structures and algorithms. Each of these is appropriate, and so any integrated solution must support the most adequate view at each phase of development. It must also provide the correspondent mappings and traceability mechanisms to be able to switch from one kind of view to the other as required, supporting the use of the appropriate modularisations used at the appropriate stage.

Concretely, Theme/Doc, CAM and Theme/UML define different artefacts and composition mechanisms, and no single, one-to-one mapping can be defined. During

the mapping process, some of the decisions to be taken rely on the expertise of the developer, so it is not possible to define an automated process that gives a deterministic result for every mapping scenario. In other words, software development is an intellectual endeavour, and the software developer must make decisions during the mapping process. We define heuristics to guide this process, and we also define a traceability schema to record the justifications of these decisions for the future. Summarizing, our aim is to alleviate the software developer task by providing guidelines for decision-making, and the means to re-visit those decisions if required.

4 AO Requirements to AO Architecture

The first step in our development process is to design a system architecture that fulfils the requirements, either functional or non-functional/quality attributes. Any requirements specification identifies a core set of actions the system must perform and a set of domain objects interrelated with these actions. Components at the architecture stage are identified as the main elements that require or carry out actions. Actions are mapped into operations, which are placed (or grouped) in interfaces. Interfaces encapsulate related functionalities that are realised or required by components.

Software architects use a combination of sources to produce the most appropriate architecture. Initially, a first version of the architecture is likely to be constructed driven exclusively by requirements. Subsequently, the software architect may refine the initial architecture considering reuse strategies of Commercial-Off The Shell (COTS) components [14], relevant architectural patterns, knowledge about the application domain and experience. Different strategies to address conflicting sets of quality attributes (e.g., response time versus security) should also be considered to produce the final architecture.

In this section, we discuss the creation of an architecture from a starting point of Theme/Doc models produced from analysing the requirements. We provide a set of heuristics for considering the mapping of these models to an AO architecture. The process described here should be considered as complementary to existing practice.

The section is divided into two main blocks, one that describes the mapping of base elements and one that describes the mapping of aspect information. For the mapping of base elements, traditional techniques can be used [7, 15–18]. This paper does not try to improve the current state-of-art of these traditional techniques; instead, it complements them by making explicit how to address mapping of aspect requirements, expressed in Theme/Doc, into the architecture. In addition to base elements, a new mapping of aspect requirements to aspect component is presented.

4.1 Theme/Doc to CAM: Base Elements

Table 6 outlines mappings from base elements in Theme/Doc to CAM. There are several alternatives for entities and theme requirements in particular, and the table outlines the heuristics the architect considers in making the mapping. In the following subsections, each alternative is explained in more detail and illustrated with examples from the Auction System.

Table 6. Mapping base elements: Theme/Doc to CAM

Theme/Doc	CAM	Heuristic
Entities from individual view	Component	At architectural level, components are coarse-grained encapsulations of relevant entities in the system. It is likely that entities interrelated with actions that are mentioned in multiple themes are component candidates. Traditional techniques will also assist here
	No mapping – ignored	Not all identified entities are components, only the main entities identified in the themes. Entities that are not mapped are postponed until design, where they will appear as component internals. Traditional techniques will also assist here
Theme Group	Interface	The coarse-grained group perspective of a set of themes is likely to motivate an interface to components that is at a good level of granularity. Traditional techniques will also assist here
Base theme requirement	A decision	Some requirements might be satisfied by adopting some decisions at the architectural level. Alternatively, some requirements could justify or motivate the adoption of some decisions. Again, traditional techniques will assist here
	A constraint	Many requirements describe constraints on other structure/behaviour. Traditional techniques will assist here
Base theme requirement	One or several operations plus provided/required relationships on components map to one or several interfaces	In general, it is most likely that a requirement defines some required behaviour. Depending on the level of granularity of that behaviour, it may be manifest at the component level. One indicator of that is whether it was previously decided that an entity mentioned in the requirement mapped to a component. If so, then the behaviour is likely to map to an operation on an interface previously derived from the Theme Group. The entity mentioned in the requirement that requires the behaviour will have a required relationship with that interface. Accordingly the entity that carries out the behaviour will have a provided relationship with the interface. An individual behavioural description could lead to several operations in the same or different interfaces, as we will illustrate later. Traditional techniques will also assist here.

Table 6. (*continued*)

	Postponed requirements	Some requirements may be at too low a level of granularity to be addressed at architectural level, or may be more appropriately addressed at the design or implementation phases. Such requirements may be postponed. Traditional techniques will assist here
Base theme	Interaction between CAM components (sequence diagram) that fulfils all requirements associated with the theme	A base theme encapsulates an action concern and often has multiple requirements associated with it. It is mapped by means of mapping each one of its associated requirements. A sequence diagram for the structural view of the architecture should be constructed. This sequence diagram will contain an interaction between CAM components that satisfies all the theme requirements. Traditional techniques will assist here

4.1.1 Entities to Components

Entities identified in Theme/Doc individual views are used for identifying architectural components. Using the heuristics in Table 6, four components are initially identified for the Auction System example from the shared entities in the Theme/Doc individual views:

- **`AuctionSystem`:** This encapsulates the core and administrative functionalities of the system.
- **`Auction`:** This is responsible for performing the auction of a single item.
- **`CreditCard`:** This represents an external service that interacts with the `AuctionSystem` component in order to transfer funds from customers' real accounts to system accounts.
- **`Customer`:** This encapsulates the interface for managing the interaction with the system customers.

Of course not all components map to entities derived from stated requirements. Just as when using standard techniques, architectural experience and deliberations may cause the architects to consider additional components from an overall system perspective. For example, in the Auction System there is a notion of `SystemFundsAccounts` that contains the funds associated with customers' accounts. These accounts are used for ensuring buyer's solvency on an auction, as stated in the requirements of Table 1.

Initially, when mapping entities from requirements into architecture, software architects have to decide whether each entity deserves to be managed as a separate component. Introducing a fine-grained component with little functionality could make no sense, since it could lead to a more complex architecture and decrease system performance. The potential to use third party components should also be considered. Where the use of COTS components is not possible, the software architect should

consider whether it is worth specifying coarse-grained components with a view to reuse in future applications.

However, mapping `SystemFundsAccounts` as an individual component would lead to an architecture with a potentially high number of component instances, each one representing a single `SystemFundsAccount`. Software architects, using their experience, decide to create a `VirtualBank` component, which will be responsible for storing and managing the set of `SystemFundsAccounts`. This has the potential to be reused in multiple applications. The decision for creating this component is based on the information specified in the system requirements, software architects' knowledge and expertise, and the knowledge of the application domain. Traditional techniques [7, 15, 16, 17, 18] may assist to adopt this kind of decision.

4.1.2 Theme Groups to Interfaces

Once components have been created, the next step is to assign responsibilities to the components. These responsibilities are determined by the set of interfaces provided by each component. Using the Theme Group View, introduced in Sect. 2.2.1 and illustrated in Fig. 3, themes are grouped to capture broader system goals. These groups map to interfaces.

For the Auction System, four theme groups are identified: `AuctionManagement`, `CustomerManagement`, `Auction` and `CustomerNotification`. Using this view, four corresponding interfaces are created for the initial version of the architecture: `IAuctionManagement`, `IAuction`, `ICustomerManagement` and `ICustomerNotification`. The external service `CreditCard` identified in the previous step imposes its own interface `ICreditCard`. The architect must also consider the interface for the component `VirtualBank` from the perspective of reusability and application integration for which it was defined. Requirements about `VirtualBank` responsibilities are not explicitly shown in themes, but may be derived from the requirements implicitly. To achieve this, the architect mines `VirtualBank` responsibilities from existing themes. The `VirtualBank` has been created for centralising the management of customers' funds transferred to the `Auction System`. Therefore, a corresponding interface called `IFundsManagement` is created. It is also considered as a new theme group, as illustrated in Fig. 8. Themes that contain requirements that imply management of `SystemAccounts` are placed in this group.

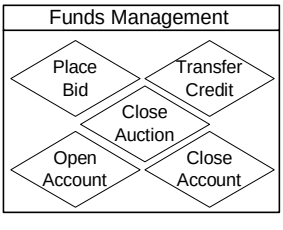


Fig. 8. New theme group

4.1.3 Requirements to Decisions, Constraints, Operations or Postponed Requirements

Requirements associated with base themes are analysed for filling interfaces with operations. Most requirements will express behaviours which are required by an entity (a *requester*) and carried out by another entity (a *performer*). The required behaviour maps to an operation that will be placed in the interface of the Theme Group that contains the theme. This interface will be required by the component that maps to the requester entity and provided by the performer entity.

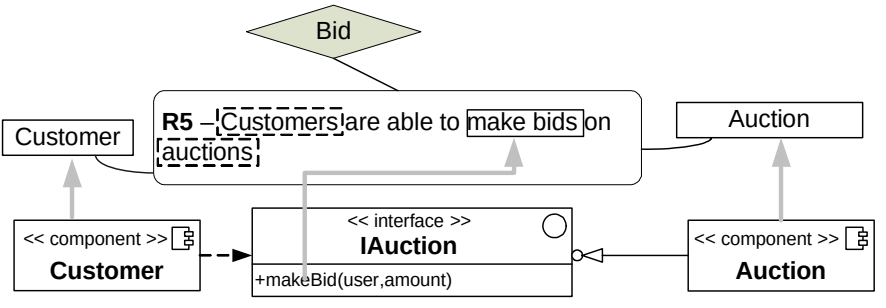


Fig. 9. Mapping an individual requirement

For instance, as illustrated in Fig. 9, `R5` "Customers are able to make bids on auctions" motivates the operation `makeBid` on the `IAuction` interface. Customers request these operations, so a required relationship is created between the `Customer` component and the `IAuction` interface. Auctions must carry out the placement of bids; therefore, a provided relationship is added from the `Auction` component to the `IAuction` interface.

Additionally, each time a bid is placed, the bid amount has to be blocked in the system account. Thus, `R5` also motivates the creation of an operation `reserveCredit` on the `IFundsManagement` interface. Funds blocking is requested by the `Auction`, and executed by the `VirtualBank`. Therefore, required and provided relationships from `Auction` and `VirtualBank` components, respectively, are added to the `IFundsManagement` interface.

All individual requirements are considered in this manner with the resulting architecture illustrated in Fig. 10.

Some requirements associated with base themes may map into constraints and/or decisions. For instance, `R15` "Customers are allowed to place their bids until the auction closes", motivates the creation of the `makeBid` operation, and it also adds the precondition "The auction must be open" for this operation.

Requirements like `R7`, "Customers are able to increase their credit by asking the system to transfer a certain amount from their credit card" maps to several operations. `R7` also influenced the decision of adding the `VirtualBank` component to the architecture.

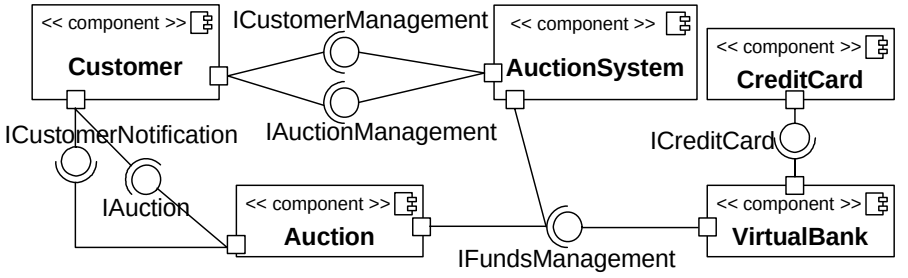


Fig. 10. Architecture derived from analysis of base themes

Finally, some requirements are about component internals, and consequently are postponed to later phases. For instance, `R17`, "Once an auction closes, the system calculates whether the highest bid meets reserve price given by the seller" is about the internals of component `Auction`, and consequently, it is associated with this component for consideration at design time.

4.1.4 Base Themes into Component Interactions

A base theme is mapped into an architecture by means of mapping each of its individual requirements. Each theme is specified with an interaction (sequence diagram) between two or more components in a CAM behavioural view. For example, Fig. 11 shows the sequence diagram for the theme `Bid`, which satisfies requirements for placing solvent bids

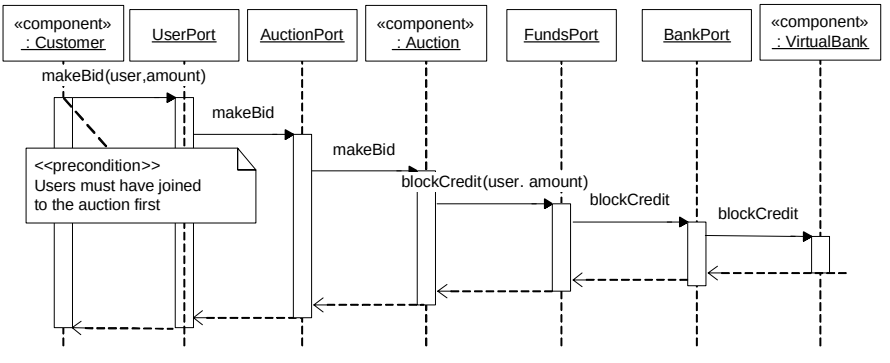


Fig. 11. Sequence diagram associated to the `Bid` theme

4.2 Theme/Doc to CAM: Aspect Elements

Table 7 outlines mappings from aspect elements in Theme/Doc to CAM. The mappings are grounded in previous work described in [19, 20] that captures heuristics for how crosscutting requirements, such as those captured in aspect themes, can be transformed into different kinds of architectural artefacts. As discussed in [19, 20], a crosscutting requirement can map into a component, an aspect component, a base or

aspect decision relating to the architecture or outside the architecture. Each of these alternatives corresponds to a different strategy that fulfils the crosscutting requirement. In this paper, we extend this discussion including heuristics to allow requirements to be *postponed*. If a requirement is likely to result in a scattered and/or tangled representation at a later stage in development, it is considered as a crosscutting postponed requirement. We illustrate these concepts with examples in the following sections.

The heuristics for both mapping and postponing benefit from a further categorisation of the kinds of concerns that are captured by an aspect theme. In particular, it is convenient to think about aspect themes as either "*verb-based*" or "*adjective-based*".

- **Verb-based aspect themes:** Define behaviour (or actions) for which the theme is responsible and that cut across other themes. For example, a `Persist` theme is likely to define behaviour that involves storing elements in persistent storage.
- **Adjective-based aspect themes:** Describes other themes in terms of constraints or properties other themes must fulfil. For example, a `Performance` theme is likely to add a time-based constraint on the processing of actions in a related theme.

Table 7. Mapping Aspect Elements: Theme/Doc to CAM

Theme/Doc	CAM	Heuristics
Aspect Theme	1. A component 2. An aspect component 3. A base or aspect decision relating to the architecture or outside the architecture	Heuristics for these mappings are presented in [19, 20]. In addition, verb-based themes are likely to map to aspect components, while adjective-based themes are likely to map to constraints and decisions on components
	A set of (base or aspect) postponed requirements	Some requirements may be at too low a level of granularity to be addressed at architectural level, or may be more appropriately addressed at the design or implementation phases. Such requirements may be postponed
Crosscutting Relationship	An aspect triggering in a sequence diagram, marked with the <<aspectual>> stereotype in the sequence diagram of the behavioural view, and optionally (not recommended) a <<crosscuts>> dependency in the architecture view	Theme/Doc's crosscutting relationship illustrates the themes that are crosscut by an aspect theme. When the theme has been mapped to an aspect component, the architect can mark where crosscutting behaviour happens in CAM's component interaction diagram, and also illustrate it, if required, at a higher level in the structural view

Table 7. (*continued***)**

	An indicator of which components are affected by architectural decisions and aspect architectural decisions	If the aspect theme is an adjective-based theme, then the crosscutting view guides the architect as to which components should be constrained.
	An indicator of which components have to address aspect postponed requirements during its design	Again for adjective-based themes, components may have crosscutting postponed requirements that need to be constrained
Requirement associated with an aspect theme	Operations on a component or aspect interface	As in base themes, requirements may define some required behaviour. Depending on the level of granularity of that behaviour, it may be manifest at the component level and it will map to operations on aspect component interface. It is most likely that these requirements are associated with verb-based aspect themes
	An architectural element participating in a (base or aspect) decision	Some requirements might be satisfied adopting some decisions at the architectural level. Additionally, some requirements could justify or motivate the adoption of some decisions. It is more likely that these requirements are associated with adjective-based aspect themes
Requirement associated with an aspect theme	A set of constraints attached to one or more components	Crosscutting requirements can be satisfied by attaching constraints that cut across several components, interface and/or connections in the architecture. It is more likely that these requirements are associated with adjective-based aspect themes
	A (base or aspect) postponed requirement, attached to one or more components	Some requirements may be at too low a level of granularity to be addressed at architectural level, or may be more appropriately addressed at the design or implementation phases. Such requirements may be postponed. Traditional techniques will assist here

As can be seen, the verb-based verses adjective-based distinction is influential, and so in the following subsections, we illustrate examples from the Auction System under those categories.

4.2.1 Verb-Based Aspect Themes Mapping

Verb-based aspect themes motivate the incorporation of crosscutting behaviour in the system architecture. If the modularisation of these behaviours outside base components can help to achieve the architecture goals like component reusability or parallel development, they should be encapsulated in aspect components. In other cases, the software architect should consider postponing the inclusion of this theme behaviour for later. We illustrate these concepts using the `Persist` and `Multilingual` verb-based themes. For the `Persist` theme, a straightforward decision is made to introduce a `Persistence` aspect component in the architecture. In order to determine the interfaces of component interactions this aspect component crosscuts, the crosscutting view and package view of Theme/Doc are used (see Fig.(1, 3). Figure 12 shows how `Persistence` is mapped. The `Persist` theme maps to the `Persistence` aspect component. `Persist` crosscuts the theme `Bid`, which is contained in the `Auction` theme group (see Fig. 3), mapped into the `IAuction` interface. Therefore, the `Persistence` aspect component will crosscut the `IAuction` interface (Fig. 12, label 4). Similar crosscutting indicators are made to the `IAuctionManagement` and `ICustomerManagement` interfaces (Fig. 12, labels 1, 2, and 3).

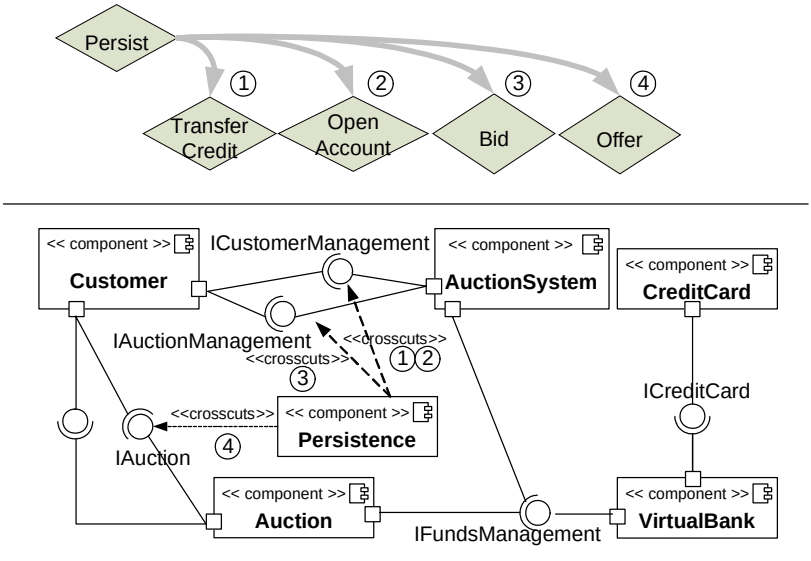


Fig. 12. Mapping `Persist` from Theme/Doc to CAM

Sequence diagrams corresponding to the base themes that are crosscut by the `Persist` theme need to be updated in order to reflect when the triggering of persistence crosscutting behaviour must happen. This means that the software architect decides the exact point (e.g., before sending, after receiving) where `Persistence` behaviour will be triggered. For instance, Fig. 13 illustrates the sequence diagram that captures the addition of persistence to the theme `Bid.` The software architect, for reasons explained in the next section, decides to persist the bids after they are processed by the `Auction` component.

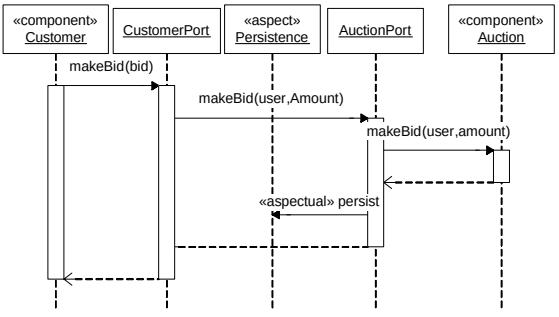


Fig. 13. Sequence diagram: `Bid` theme with persistence behaviour

The second verb-based theme we consider is `Multilingual`. It contains the requirement `R26` that states "Information displayed to customers must be available in English, German, French, Italian, Portuguese and Spanish". The crosscutting view of Theme/Doc illustrates this theme only crosscuts the `Customer` component. Initially, the software architect considers creating a `Multilingual` aspect component that performs the translation of messages shown to users. However, it only crosscuts the `Customer` component. Since this aspect component is strongly linked to the `Customer` component only, to model it separately does not improve either architecture understanding or promote parallel design/reuse. Therefore, the architect decides to postpone this theme. At the design phase we will see that `Multilingual` can be designed as an aspect contained in the internals of the components. At the architectural level, how `Multilingual` is designed/implemented is hidden in internals of the component.

4.2.2 Adjective-Based Themes Mapping

Mapping adjective-based aspect themes is not as straightforward as verb-based themes. They express constraints, properties or quality attributes of the system that might be satisfied by means of different alternative solutions. Each alternative implies the introduction of different kinds of artefacts into the architecture.

An adjective-based aspect theme, which represents a system property, can add to the architecture:

- Base components providing services to satisfy (partially or fully) the property.
- Aspect components encapsulating crosscutting behaviours that satisfy (partially or fully) the property.
- Decisions useful to satisfy (or simply the satisfaction of) the required property. These decisions can even be outside the scope of the architecture, for example they may be related to organisational issues such as, maintenance, planning and cost. The consequence of these decisions may affect the architecture outline indirectly.
- Often the granularity of an adjective-based aspect theme is too small to be addressed at architecture level and is simply postponed. The set of requirements related to such a theme must be associated to architecture components. Note these requirements may affect either single components, or be scattered across several of them.

We illustrate these concepts with the adjective-based aspect themes identified in the Theme/Doc analysis of the Auction system: `Accessibility`, `Availability` and `Performance`.

`Accessibility` imposes specific constraints on the way the Auction System will be designed and implemented. As the system has to be accessible as a website, the `Customer` component providing the system interfaces will be a set of web pages, which communicates with the `AuctionSystem` and `Auction` components via HTTP. System designers will have to make use of appropriate technologies such as, for example, cookies or sessions for web application design. However, we are not interested in this level of detail at the architectural phase. As the architecture presented in Fig. 10, 11, 12 and 13 is perfectly compatible with this aspect theme, it is postponed to be addressed in later phases. No further decision on this is required at the architectural level.

`Availability` requires that the system can be only shut down for 1 hr at night for maintenance. This crosscutting requirement does directly affect the architecture or design: it is mapped to specific constraints and decisions about the planning for maintenance, which should be designed for being completed in less than an hour. In this case we have an example of an adjective-base theme that maps to a non-architectural decision. However, designing a non-monolithic and loosely-coupled architecture should help in performing maintenance jobs faster and easier, as components are smaller and more independent. For instance, we could have decided to consider the `Auction` component as an internal part of the `AuctionSystem` component, instead of a separated component. `Availability` favours such separation because the `AuctionSystem` component could be stopped without interrupting the individual `Auction` component instances execution. In this way, we see an adjective-based theme justify some architectural decisions, such as the separation of `Auction` from `AuctionSystem`.

Several strategies and mappings for fulfilling the `Performance` aspect are possible. The encapsulated requirement (`R22`) states that “Bids must be processed in less than a second”. These could be mapped into different architectural artefacts as follows:

- **An aspect component for replication and load balance:** This crosscutting concern was not explicitly identified during the requirements, and appears now as a consequence of an aspect theme mapping. Here is a good example of identifying and modelling aspects at the right time. Replication and load balancing are technical concepts, and so normally would not appear in requirements documents.
- A **decision about adopting a specific communication style:** For example, software architects might decide to create a computer network with one host, where the `AuctionSystem` component would be deployed, plus a set of auctioneer hosts, where the `Auction` component instances are deployed. A `Resource Manager` component would monitor the auctioneer hosts and each time a new `Auction` is created; this will be deployed in the customer host. This is similar to the previous approach, but does not replicate auctions. In this case, the crosscutting requirement would introduce a `ResourceManager` component, plus a set of aspect components which monitor auctioneer.

- **A set of postponed requirements:** Software architects could decide that a careful design, efficient implementation and buying high-performance hardware is enough to meet the time constraints demanded. Then, the crosscutting requirement is postponed and recorded in the traceability file in order to be observed in later phases.

We decide to adopt the cheaper option, and postpone the theme to the design phase. This avoids the cost of additional servers.

4.3 Recording Decisions for Traceability

As we have illustrated, when designing a software architecture, some decisions need to be made. The reasoning behind the decisions needs to be captured in order to be able to react to future change adequately, and ensure decisions taken now are still valid later. This information allows us to trace the dependencies between the architecture and the requirements and understand why these dependencies exist. Recording alternatives, justifications for each alternative and the alternative selected during the process can help to make changes faster. Moreover, the decisions and justifications made could be reused in similar architecture derivation processes. To record this information, the XML schema (presented in [19] and demonstrated in [21]) shown in Fig. 14 is employed.

```
<module-type id="..">
  <alternative id="1">
    <!-detail -->
    <justification> text </justification>
    <advantage>     text </advantage>
    <disadvantage>  text </disadvantage>
  </alternative>
  ……
  <alternative id="n"> … </alternative>
  <alternativeSelected id="n">
    <justification> text </justification>
  </alternativeSelected>
</module-type>
```

Fig. 14. XML template for recording traceability information

The “module type” may be a component, aspect or requirement with the “role” capturing its unique identifier. Each “alternative” is outlined separately, with the “detail” capturing any relevant technical characteristics of the alternative. “Justification” for the alternative may also be included, together with advantages and disadvantages. The “alternative selected” references one of the alternatives listed, with the justification providing the context for that decision. Any of the available XML tools (i.e. Avolta XML spy[4]) that help to introduce data and validate instantiated schemas could be used by the software architect.

In this section, we summarise the different kinds of decisions that have to be made when designing an architecture and detail how these decisions are recorded.

[4] http://www.altova.com/products/xmlspy/xml_editor.html

Table 8. Decision making for adding a component to the architecture

Alternative	Detail for VirtualAccount
<alternative “id=1”>	<mappedIntoComponent name= “VirtualAccount”\> *Justification text:* This component is responsible for managing a single user’s virtual account
<alternative “id=2”>	<mappedIntoComponent name= “AuctionSystem” \> *Justification text:* The `AuctionSystem` component’s responsibilities include managing user virtual accounts. *Disadvantage:* Virtual accounts are a recurrent element in several systems within the company
<alternative “id=3”>	<mappedIntoComponent name= “VirtualBank” \> *Justification text:* This component is created as a generic service for managing user virtual accounts. It is designed to support potentially multiple different applications running within the same company
<alternativeSelected “id=3”>	*Justification text:* Stakeholder considers it an excellent idea to create a central repository for managing users’ virtual accounts for all their internet applications

4.3.1 Base Information

The decisions taken by the software architect when mapping base information are primarily related to granularity, number of components and interface definitions. The

```
<requirement id="7">
  <alternative id="1">
   <mappedInto op="transferCredit" interface="ICustomerManagement" requiredBy="Customer">
      <refinedRequirement id="7.1">
         Customers can request to increase their credit by asking to transfer
         a certain amount from their credit card
      </refinedRequirement>
   </mappedInto>
   <mappedInto op="transferCredit" interface="ICustomerManagement" providedBy="AuctionSystem">
      <refinedRequirement id="7.2">
        AuctionSystem must be able of delegating adequately this request to the
        IFundsManagement interface
      </refinedRequirement>
   </mappedInto>
   <mappedInto op="transferCredit" interface="IFundsManagement" requiredBy="AuctionSystem">
      <refinedRequirement id="7.3">
         AuctionSystem must be able of delegating adequately this request to the
         IFundsManagement interface
      </refinedRequirement>
   </mappedInto>
   <mappedInto op="transferCredit" interface="IFundsManagement" providedBy="VirtualBank">
      <refinedRequirement id="7.2">
         VirtualBank  must provide funds transferring from actual credit cards to
         system accounts.
      </refinedRequirement>
   </mappedInto>
   .....
  </alternative>
  ...
</requirement>
```

Fig. 15. Traceability information about requirement mapping

corresponding traceability file records the justification of defining each of the components.

Table 8 contains the relevant information from the XML schema that captures the information recorded when mapping the `SystemAccount` theme group from the Theme/Doc analysis. Capturing the relevant details for each alternative is straightforward.

Decisions related to the mapping of theme groups to interfaces are similar to components. Each alternative is recorded, together with the interface (or set of interfaces created), and, for each interface, the set of themes where this interface participates. This will help us to reconstruct the theme view at design level. Table 9 shows an example of an alternative for a theme group mapping into interfaces. More alternatives are not shown for the sake of brevity.

Table 9. Decision making for mapping a theme to an interface

Alternative	CustomerManagement
<alternative “id=1”>	<mappedInto> <interface id= “i1” name=ICustomerManagement> *Themes* OpenAccount, CloseAccount, Login, Logoff, TransferCredit *Justification text:* This interface represents a direct mapping from themes. All operations about customer management are encapsulated in the same interface.

Table 10. Decision making for mapping an aspect theme to an aspect component

Alternative	Detail for persistence
<alternative “id=1”>	<!-- Persist bids before being processed --> <mappedInto aspect= “Persistence” describedBy “persistence(0).xmi” \> *Justification text:* Bids are valid when they are received by the Auction System and so they must be persisted as soon as they arrive.
<alternative “id=2”>	<!-- Persist bids after being processed --> <mappedInto aspect= “Persistence” describedBy “persistence(1).xmi” \> *Justification text:* Bids are not valid until they are processed, and so they may not be persisted before processing.
<alternativeSelected “id=2”>	*Justification text:* Alternative 2 conforms to a general business rule in the company that applies to all applications.

The link between requirements and operations also need to be recorded to reconstruct themes when reaching the design phase. In this case, a base requirement will be satisfied by means of interaction between components. Each message in the interaction realises part of a theme. In order to promote parallel and independent development, we refine these requirements with the purpose of associating them to individual operations. A refined requirement contains only the information strictly needed to design/implement the operation correctly.

For example `R7` (funds transferring from users' credit card to system accounts) is satisfied by means of an interaction between the components `Customer`, `AuctionSystem`, `VirtualBank` and `CreditCard`. Figure 15 illustrates how the `R7` is mapped into several operations, how it is refined, and the information recorded in the traceability file. Likewise, the same process is applied when requirements map to decisions, constraints or are postponed.

4.3.2 Aspect Information

Recording decisions related to aspect themes mapping is even more crucial than for base themes. Aspect mapping, especially adjective-based aspect mapping, is quite complex and several non-straightforward alternatives often exist. Documentation about the decisions made will be of considerable help during future system change activity. Normally, verb-based themes are mapped directly to aspect components. However, some minor decision-making may be required. In particular, the composition specification describes when crosscutting behaviour happens relative to base behaviour. If the requirements do not state this information explicitly, then the architect may need to consider the different options, and select one. It is likely that the architect will rely on his knowledge of the domain and/or previous experience to assist here.

Coming back to our example, it is not explicit in the requirements whether bids should be persisted before or after processing them. In this case, the software architect decides to persist the bids after processing them, which is the normal practice in the company's persistence management. This decision is recorded in the traceability file as illustrated in Table 10.

Mapping adjective-based crosscutting requirements requires consideration of a wide range of alternatives, and consequently, a bigger effort is required from a decision-making perspective. Because they describe properties or constraints on other behaviour (as opposed to defining behaviour themselves), they may or may not map directly to an architecture artefact. The architect is likely to draw on his domain knowledge and experience in deciding what is to be done. Each alternative, the artefacts that it would generate, and the rationale, advantages and disadvantages are all recorded in the traceability file.

The `Performance` theme is a good example of this, where possibilities range from defining new components to explicitly handle performance, or postponing its consideration to a later stage in development by simply being explicit about the required constraints. Table 11 illustrates the capturing of relevant information in the traceability file related to `Performance` mapping.

Table 11. Decision making for handling an adjective-based theme

Alternative	Detail for performance
<alternative “id=1”>	<!-- Replication and Load Balancing (RLB) --> <mappedInto strategy= “RLB” describedBy “rblStrategy.xmi” \> *Justification text:* Introduces a replication and load balancing aspect component to increase performance *Advantage:* - Well-known and widely-used *Disadvantages:* - Not easy to design/implement, as replicated bids have to be reconciled for winner selection. - Increases cost of deployment as needs more servers
<alternative “id=2”>	<!-- Resource manager and auctioneers network (RMA) --> <mappedInto strategy= “RMA” describedBy “rmaStrategy.xmi” \> *Justification text:* Provides a way to monitor and relocate auctions to better-performing servers. *Advantage* - Similar to replication and load balancing, but without replicating auctions. *Disadvantage:* - Performance can not be fully guaranteed if the load relating to an individual auction increases quickly - Increases cost of deployment as needs more servers
<alternative “id=3”>	<!-- Resource manager and auctioneers network (RMA) --> <mappedInto postponedRequirements=”true”> <postponedReq id=”R23” constraint= “Auction”> Bids must be processed in less than a second. </postponed> <postponedReq id=”R22” constraint=”AuctionSystem, Customer, VirtualBank, CreditCard”> Credit transfers must be processed in less than half a second. </postponed> *Justification text:* Performance achieved with current hardware and efficient design. *Advantage:* -No extra servers are required. *Disadvantage:* -If current hardware and an efficient design are not enough to satisfy this requirement, the architecture will have to be refined. This would imply an increase in the development cost and time-to-market
<alternativeSelected “id=3”>	*Justification text:* Company is young, so cheap solution is best

4.4 Using the Traceability Information to Support Change Management

To illustrate the value of recording architectural decisions, we consider a scenario in which various changes are required. Changes to an architecture are stimulated by

changes to requirements or changing system contexts (e.g., new company policies). These types of changes can occur when using any type of methodology and are not the reserve of aspect-based methods.

4.4.1 Requirements Change Management

In Sect. 4.3.1 we discussed how requirement `R7`, which is associated with the `TransferCredit` theme, is mapped to a component interaction. Let's consider a situation where requirement `R7` changes: customers can now transfer funds from a debit card, in addition to using a credit card. This impacts the architecture because it has to deal with new functionality.

To assess the scope of the impact, the architect first needs to consider the architectural artefacts to which the original `R7` mapped. This information was recorded in the traceability file as illustrated in Fig. 15 and now helps the software architect to determine the change impact by following the trace information. Alternatives and justifications recorded should be reviewed to determine if decisions adopted in the past are still the best choices in the light of change.

4.4.2 System Context Change Management

Decisions are often motivated by the environment where the system is developed and deployed. For example, initially, we decided to adopt the cheapest option for the `Performance` aspect theme because the main goal of this company was to save as much money as possible until the system was fully working and producing benefits.

However, a high-performance hardware acquisition and a careful design are not enough if the number of concurrent clients exceeds a limit. Above a threshold, system performance is degraded, and the time constraints in requirements `R22` and `R23` (see Table 11) are satisfied.

We now consider a situation where the company becomes very successful with its Auction System, and the number of concurrent clients is much higher than expected, causing a breach of the performance threshold. Clearly, the previously selected alternative is no longer sufficient. From the traceability information (Table 11), we see that the decision was made on cost. But now cost is not so important, and so the conditions under which this decision was made have changed. In general, the architect re-examines the advantages and disadvantages of all previously considered alternatives, and may also add new ones.

After analysing all the alternatives, the architect decides that employing load balancing and replication is the best means for addressing `Performance` over the long term. This new decision is made based on analysing the usage statistics of the system as well as the cost and availability of components that can be reused to design/implement this strategy. These additional considerations are added to the alternative selected justification to support future changes.

5 Architecture to Design

The next step in our development process is to provide a design that details the internal structure and behaviour of the components specified in the system architecture. At the

design stage, we provide the data structures and algorithms that specify the required functionalities of each base and aspect architectural component. To a large extent, standard oo design approaches are used to create Theme/UML models. As with architects, designers also rely on design patterns, general domain knowledge and experience. One additional dimension that is in the decision-making realm of the designers is whether to reuse an existing COTS component, or whether to create custom designs from scratch. Again, as with the requirements-to-architecture mapping section, the mapping heuristics described in this section do not substitute for the use of patterns, domain knowledge or experience. Rather, we provide a set of heuristics for mapping an AO architecture to an AO design.

5.1 CAM to Theme/UML: Base Elements

Applying the general guidelines outlined in the previous section, we construct a specific mapping between CAM and Theme/UML. As illustrated in Sect. 4.1, themes

Table 12. Mapping base elements: CAM to Theme/UML

CAM	Theme/UML	Heuristics
Base component	Base themes	A component is derived from a Theme Group when mapping requirements to architecture. Each requirements level theme in that group maps to a Theme/UML design theme. Each theme contains a specification component, the interfaces that the component exposes and the ports through which these interfaces are realised, which are relevant to that theme. When moving from architecture to design it may be possible to reuse a COTS component instead of designing and implementing a new component. If a suitable COTS component is available, the themes to which the component maps represents the component with no internal design. When a suitable COTS component is not available it must be designed. When a component is designed each theme contains the component and a partial design of the component that satisfies the theme requirements
	Aspect themes	Some concerns that do not arise during requirements analysis emerge when moving from architecture to design. These concerns are typically technical concerns such as distribution that support the underlying business logic that is captured in the base component. These concerns are generally crosscutting concerns and are mapped to aspect themes in design. These themes provide design solution for these emergent concerns
	Composition relationship	A merge composition relationship between the themes is specified to define how these themes are composed to define the complete component design
Base component composition	Not mapped	Themes describe the design of components. The connectors that assemble required and provided component interfaces are not defined in the detailed design as they are specified in the architecture

identified in requirements analysis are assigned to base architectural elements. Using the traceability information generated when mapping requirements to architectural elements, designers can determine what subset of the themes a component needs to address; what functionality each theme within the component is responsible for; and what requirements need to be satisfied. It should be remembered that when mapping requirements to interface operations (see Fig. 15, Traceability information about requirement mapping), requirements may be refined (i.e., rewritten) to support the architecture interface specification. Table 12 defines specific guidelines for mapping the CAM architecture to Theme/UML.

5.1.1 Base Components to Themes

A component derived from a Theme Group maps to Theme/UML design themes. Each requirements level theme in that group maps to a Theme/UML design theme. In Figure 16 an abridged mapping of the architectural `AuctionSystem` component to themes in design is illustrated. One of the interfaces exposed by the `AuctionSystem` component is the `ICustomerManagement` interface. When mapping from requirements to architecture, a group of themes are mapped to that interface (see Sect. 4.1.2). The Theme Group mapped to the `ICustomerManagement` interface is depicted in the top left corner of Fig. 16. The designer uses this information to define a design theme for each requirements level theme related to the `ICustomerManagement` interface. Two of these themes, `Transfer` and `CloseAccount`, are illustrated in Fig. 16. In both themes, only the part of the internal structure and behaviour of the `AuctionSystem` component relevant to the theme is designed.

At this stage, standard oo design techniques are used to design the internals. For example, in the `TransferCredit` theme, a class named `Dispatcher` is defined. As illustrated in the `Transfer` sequence diagram fragment, instances of this class check to ensure the user ID is valid before redirecting the request to the `IFundsManagement` interface. In the `CloseAccount` theme, an `AccountManager` class is defined. Instances of this class hold references to the `Account` objects; each object represents a user account in the Auction System. The behavioural flow specified for closing an account is illustrated in the `CloseAC` sequence diagram fragment. Here the `closeAccount(user)` call on the `IFundsManagment` interface is made to transfer users account balance back to their credit card. Once this has been completed, the `Account` object is destroyed closing the users account.

Figure 16 also illustrates how component internals are designed. When moving from architecture to design it may be possible to reuse a COTS component instead of designing and implementing a new component. When a COTS component is reused, the component does not require design. In this case, the component maps to design themes that define the component but do not specify the component's internal design. This is useful if the component requires adaptation. Although the internals of the component are not designed, adaptation logic can be defined for the reused component. The reuse of a COTS component is illustrated later in Sect. 5.2.1.

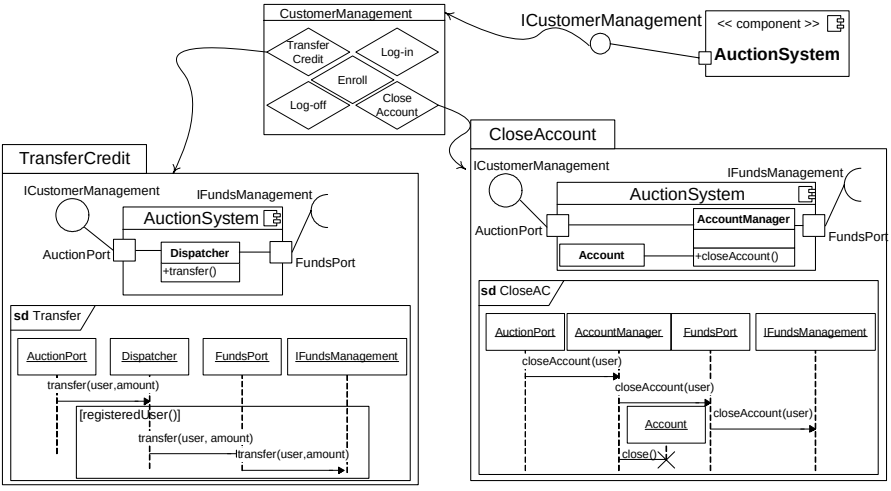


Fig. 16. Mapping `AuctionSystem` component to base themes

5.1.2 Base Components to Aspect Themes

As described in the previous section, a component maps to design themes. A component can also map to themes that provide designs for emergent concerns. When designing the `AuctionSystem` component other concerns must be addressed that are not identified from requirements. An example of such a concern is distribution. The `AuctionSystem` component is required to provide services over the internet and as such, requires distribution support. Emergent concerns such as distribution are likely to be designed as aspect themes. Distribution is a concern that is applicable to all components that are distributed. It is specified as a `Distribution` aspect theme that crosscuts the behaviours that realise the required and provided interfaces of distributed components.

The top of Fig. 17 shows that the `Customer` and `AuctionSystem` components are distributed and that these components are connected via the `ICustomerManagement` interface. Through this interface, the `Customer` component can call `TrasferCredit`, `Log-in`, `Log-off`, `Enroll` and `CloseAccount` behaviour on the `AuctionSystem` component. At the bottom of Fig. 17, the `Distribution` aspect theme is illustrated. Unlike the `TransferCredit` and `CloseAccount` themes, illustrated in Figure 16, this theme does not specify components. Instead it defines behaviour and structure that crosscuts the `TrasferCredit`, `Log-in`, `Log-off`, `Enroll` and `CloseAccount`, themes which detail the internals of the `Customer` and `AuctionSystem` components. The `Distribution` theme is defined to specify how distributed calls should be handled at run-time. When composed with the other themes that detail the `AuctionSystem` component, it redefines specific methods to ensure that they support distribution.

5.1.3 Base Component to Theme Composition Relationship

A merge composition relationship between the themes is specified to define how these themes are composed into a complete component design. Once all stated and

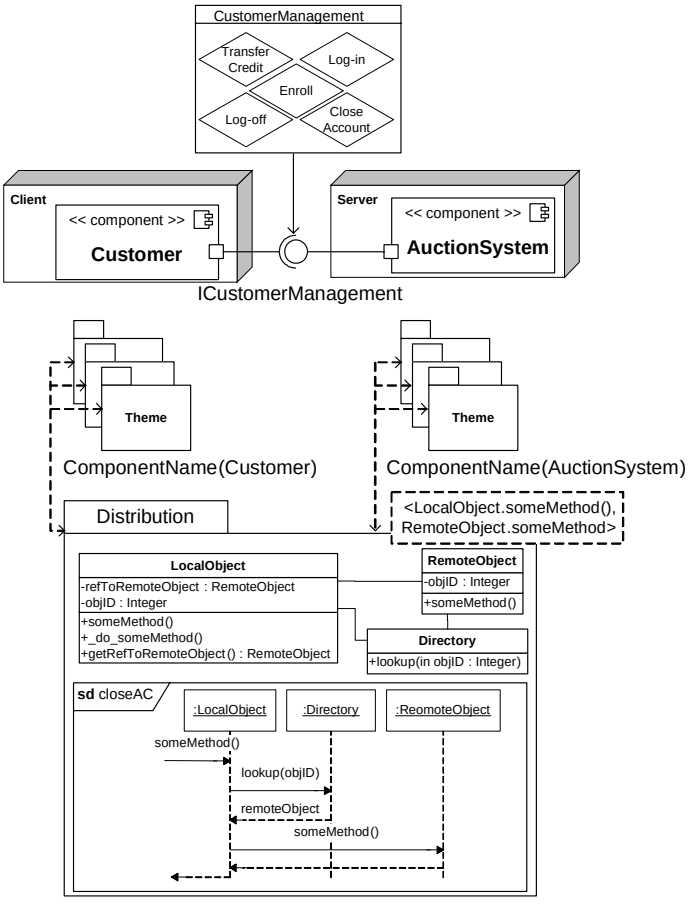


Fig. 17. Mapping `AuctionSystem` component to aspect theme

emergent themes for the `AuctionSystem` are specified, they need to be merged to produce the composed component.

As illustrated in Fig. 17, composition relationships are defined between the themes that map to the `Customer` and `AuctionSystem` components. These composition relationships include specifications for the composition of base themes which will include the name of the component that will result from composition and may include resolution statements to remove any design inconsistencies that may cause a theme composition. It can also include bind specifications needed when aspect themes are mapped to the component (e.g., the `Distribution` aspect theme from the previous section).

5.1.4 Base Component Composition Are Not Mapped

Base component composition is defined as a connector assembly in which required and provided component interfaces are bound to one another. These connectors are

Table 13. Mapping aspect elements: CAM to Theme/UML

CAM	Theme/UML	Heuristics
Aspect component	Aspect themes	Mapping aspect components to design is similar to mapping base components. The significant difference is that at least one aspect theme needs to describe crosscutting behaviour exposed at the interface of the aspect component. This theme defines how the component crosscuts other components
	Composition relationship	A merge composition relationship between the themes is specified to define how these themes are composed to define the complete aspect component design
Aspect composition rules	Join point execution	Each aspect composition rule specifies when (e.g., BEFORE_SEND, AFTER_RECEIVE) the aspect behaviour is executed in relation to the crosscut component. This information is used to construct the sequence diagram, which describes the sequence of crosscutting behaviour relative to a template parameter representing the base flow
	Bind relationship	The information about the specific components affected by aspects is used to construct the "bind" relationship between an aspect theme and the base themes. This relationship specifies how template parameters of an aspect theme are instantiated with specific elements of base themes
Postponed requirements	Aspect theme	Postponed verb-based requirements have behaviour that is likely to crosscut possibly multiple themes, and should be designed as a standard aspect theme, with corresponding composition rules that link the aspect with the base modules
	Constraints on design	Postponed adjective-based requirements should be taken into account (at least) wherever the corresponding Theme/Doc crosscutting relationships are defined

not mapped to the design. This is because connectors are fully specified at the architectural level.

5.2 CAM to Theme/UML: Aspect Elements

Aspect elements captured during the CAM process must also be considered at design level. Table 13 captures the mappings and heuristics for addressing these elements in design.

5.2.1 Aspect Component to Aspect and Base Themes

The mapping of an aspect component to design is similar to mapping a base component to design. The significant difference is that at least one aspect theme is

required to describe crosscutting behaviour exposed at the interface of the aspect component. In Fig. 18, we illustrate the mapping of the persistence aspect component to the `Persist` aspect theme. From the persistence Theme Group, we can see that the `Persistence` component is derived from the `Persist` theme. In this case, the aspect component is realised by reuse. A third-party `OraclePersistence` component is acquired from a COTS repository. Therefore, nothing about the internals of the component is shown in the `Persist` aspect theme.

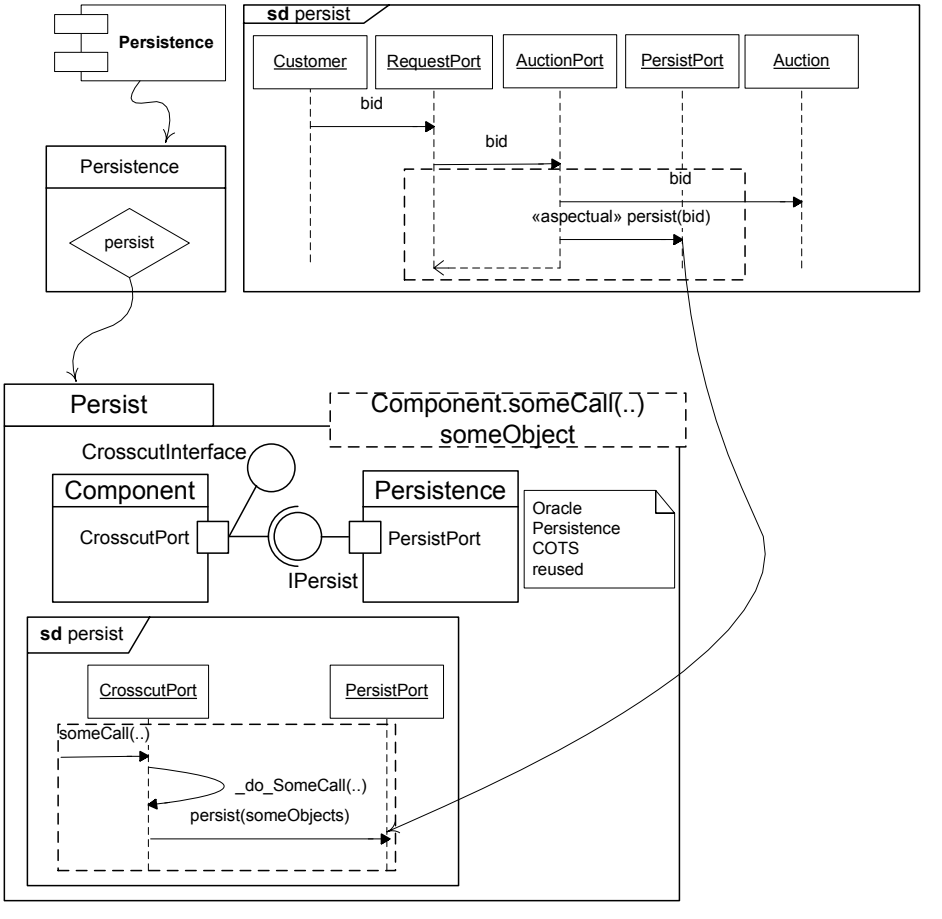


Fig. 18. Mapping `Persistence` component to aspect theme design

Aspect components contain crosscutting behaviour, but may also map to base themes that specify behaviour that is not crosscutting. When aspect components are mapped to base themes, these themes may be either composed with the aspect theme defined to realise the components crosscutting interface, or with other non-crosscutting interfaces exposed by the component. In the former case, base themes are

used to extend the behavioural flow of the crosscutting behaviour defined in the aspect theme. In the latter case, the base themes are composed to form complete designs which realise non-crosscutting component interfaces.

5.2.2 Aspect Composition Rules to Join Point Execution

At the architectural level, the manner in which crosscutting occurs between components is specified in a sequence diagram as illustrated in the `persist` sequence diagram fragment at the top of Fig. 18. This maps to the sequence diagram fragment defined in the `Persist` theme. These sequence diagrams indicate: how the aspect component is executed on architectural join points; the base design elements that represent join points; and the contextual information used in the crosscutting persistency behaviour.

The difference between these architecture and design sequence diagrams is that the one defined at the architectural level is not generic and crosscutting behaviour by tagging it as «`aspectual`». At the design level, the sequence diagram describes the crosscutting behaviour exposed by the aspect components around parameter templates exposed by the aspect theme. The sequence diagram at the architectural level identifies the `persist` call on the `PersistPort` as an aspectual method call and specifies that this occurs after a bid is made on the auction. The same call is also defined in the `Persist` theme. In the `persist` sequence diagram the aspectual method call is specified in terms of template parameters that represent all join points that are crosscut by the persistence behaviour.

When mapping architectural sequence diagrams to theme sequence diagrams, the first step is to assess where and how crosscutting occurs. Once all join points are identified then the manner in which crosscutting occurs can be identified (e.g., `BEFORE_SEND`, `AFTER_RECEIVE`, …) This information can then be used to design the generic crosscutting behaviour in the theme.

5.2.3 Aspect Composition Rules to Bind Statements

As mentioned before at the architectural level the manner in which crosscutting occurs is illustrated by showing how composed base and aspect behaviours will execute. At the architecture level there is no join point designation. Join points are explicitly identified by their proximity with messages stereotyped as «`aspectual`» . In contrast at the design level join points are selected through the designation of join points using Theme/UML join point selection mechanisms.

In Fig. 18, the `persist` message, stereotyped as `<<aspectual>>`, is placed in-between the dispatching and return of the message to the internals of the Auction component. According to CAM semantics (see Sect. 2.3), this means the crosscutting behaviour is applied after receiving the message. The designation that corresponds to that join point is illustrated in Fig. 19 as a bind statement. This bind statement, labelled (1), defines criteria to select this join point. Other designations labelled (2), (3) and (4) represent other join point designations that select join points crosscut by the `Persistence` component.

When mapping architectural join points identified in sequence diagrams to bind specifications at the design level there are two alternatives to consider. The first is to

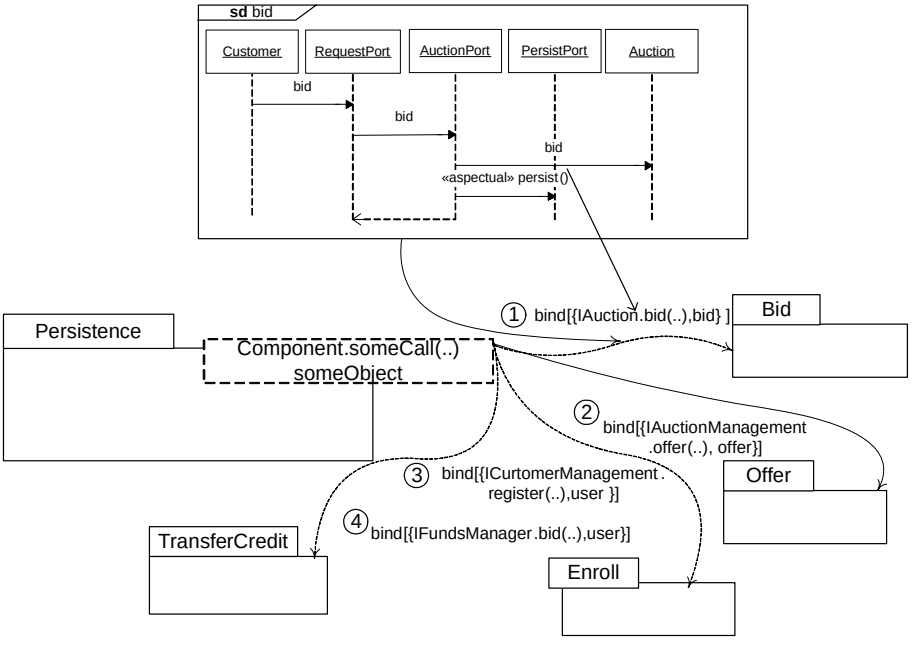


Fig. 19. Mapping join points to designators and aspect components to composition relationships

use completely qualified signatures for selecting join points in the bind statements. The second is to look for means of defining more general but appropriate selection criteria in the bind statement that can select multiple join points.

The first alternative is more applicable when join points occur in highly heterogeneous places and there are a small number of join points identified in the architecture. The second alternative is more applicable when join points occur in highly homogeneous places and there are a large number of join points identified in the architecture.

5.2.4 Aspect Component to Theme Composition Relationship

Aspect components map to at least one aspect theme that describes how the aspect component crosscuts other components. In Fig. 18 we saw that the `Persistence` component is reused. However, even though reuse may be preferable over designing and implementing components from scratch, reuse is not always feasible. When this case does arise, the aspect component may map to a number of aspect and base themes which define the internal details of the component. These themes must be composed to provide a full definition of the aspect component's design.

5.2.5 Postponed Requirements to Aspect Themes or Constraints on Design

A subset of themes identified in requirements analysis may not be addressed in architecture design, but are recorded as postponed in order to be addressed in the

design. Two examples of this in the Auction System scenario are the `Multilingual` theme and the `Performance` theme.

The `Multilingual` theme can be expressed as a theme in design. It is identified in requirements as an aspect theme, but not mapped to the architecture as an aspect component because it affects the internals of only one component. The `Multilingual` theme crosscuts the `Notification` theme which is part of the `Customer` component as illustrated in Fig. 20. The `Multilingual` theme contains behaviour for translating English text into other languages (such as Spanish). The `Customer` component is the interface through which customers interact with the `AuctionSystem` component. The interface is used to display messages to the Auction System users. The `Notification` theme modularises behaviour for sending messages to customers who are interested in the auction to indicate the outcome of the auction. The `Multilingual` theme crosscuts the message theme to translate messages into different languages for auction customers who do not understand English.

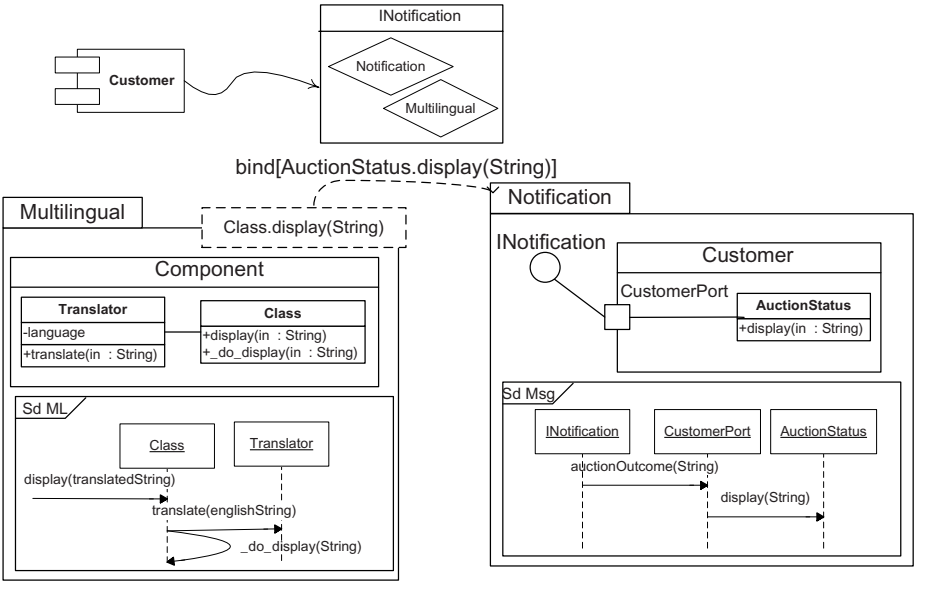


Fig. 20. Multilingual support

The `Performance` theme on the other hand cannot be represented as a theme in design. Instead, it is mapped to specific design decisions in which performance is a factor. For example, when designing the `AuctionSystem` component, performance is a constraint that restricts the use of heavy weight algorithms or data structures. The performance constraint also restricts the amount of crosscutting behaviour that can be added into the behavioural flows of the `AuctionSystem` component.

5.3 Recording Decisions for Traceability

As in the mapping from requirements to architecture, different alternatives may be considered when mapping architecture into design. The identified alternatives, their justifications and the decisions adopted during the mapping are recorded. This information helps reconstruct the development process, improve the system's understandability and maintainability, and assist engineers when dealing with change. In this section, the decisions addressed when mapping from architecture to design are summarised. The XML schema used to map traceability decisions for requirements-to-architecture mapping is the same as that used for architecture-to-design mapping.

5.3.1 Base Information

When designing base components, the primary decision that needs to be made is whether to custom-design or reuse. As such, the first activity in moving from architecture to design is to search external and internal component repositories for suitable pre-built components that fit our needs.

Table 14. Decision making for re-using a component

Alternative	Detail for VirtualBank
<alternative "id=1">	<designedbyModule="CreditCard(1).xmi">
<alternative "id=2">	<reused from ="MyVirtualBank"> <repository uri="….."> *Justification text:* Supports credit transfer with credit card companies
<alternative "id=3">	<reused from ="C&DVirtualBank"> <repository uri="….."> *Justification text:* Supports credit transfer with both credit and debit card companies
<alternativeSelected "id=2">	*Justification text:* Two possible reuse options are available, which makes alternative 1 impractical. Of the two reuse opportunities, the COTS component referenced in alternative 2 is cost-effective, and also cheaper than alternative 3. Alternative 3 is more expensive because it supports not only credit card transfers but also debit card transfers, As `R7` states that credit transfer with the Auction System is only done with credit card companies, then alternative 2 is sufficient

If one exists, the next decision is whether it is cost-effective, i.e., is it cheaper than designing it from scratch? If several candidate components are cost-effective, we select the most suitable to our needs. An example of this from the Auction System is the `VirtualBank` component. Table 14 contains the relevant information from the XML schema.

In Table 14 several alternatives for realising the `VirtualBank` component are outlined. The first alternative (row one) is to design the component from scratch. The other alternatives (rows two and three) are to reuse two different existing components

— `MyVirtualBank` and `C&DVirtualBank`. The reuse of the `MyVirtual` bank is the alternative selected (row four). As we can see the selection of this alternative is based on opportunity to reuse, functionality and cost.

When reuse is not an option there may be various several alternative design approaches available that satisfy the component's requirements. Each alternative would represent a different configuration of component internals, data structures or algorithms. An example of this from the Auction System is the `AuctionSystem` component. Table 15 contains the relevant information from the XML schema.

Table 15. Decision making for designing a component

Alternative	Detail for AuctionSystem
<alternative "id=1">	<designedbyModule= "AuctionSystem" href="../AuctionSystem(1).xmi"> *Justification text:* We map each concern to a theme
<alternative "id=2">	<designedbyModule = "AuctionSystem" href="../AuctionSystem(2).xmi"> *Justification text:* The structure of internals supports a better performance time for `makeOffer()` operations because it requires a smaller number of message interchange
<alternativeSelected "id=2">	*Justification text:* Alternative 2 improves the performance time of `makeOffer()` operations. The requirement to consider performance was previously postponed, and needs to be addressed in design

In Table 15 two alternatives for realising the `AuctionSystem` component are outlined. The first alternative (row one) is to follow the general mapping rules defined in Table 12. The second alternative (row two) deviates slightly from the general mapping rules to ensure improved performance. The second row is selected (row three) as performance is a concern that must be addressed in design.

The `AuctionSystem` component is mapped to a number of requirements level themes. One of these themes is the `TransferCredit` theme. At requirements level the `TansferCredit` theme is crosscut by the `Performance` theme. The `Performance` theme is not addressed at architectural phase of development and is postponed to be addressed during design. As we can see in Alternative 2, the influence of the crosscutting `Performance` theme is evident. Traceability information related to the `Performance` theme is described further in Sect. 5.3.2.

5.3.2 Aspect Information

The decisions and alternatives when mapping an aspect component are similar to the base components — we search for pre-built aspect components, instead of base components. Currently, aspect component repositories are not widespread, although some approaches, such as CAM/DAOP [22], have made inroads here.

Table 16. Decision making for re-using an aspect component

Alternative	Detail for persistence
<alternative “id=1”>	<designedbyModule=“Persistence”>
<alternative “id=2”>	<reused from =“OraclePersistence”> …
<alternative “id=3”>	<reused from =“SybasePersistence”> …
<alternativeSelected “id=2”>	*Justification text:* The application is to be deployed in an environment that is already using Oracle. In addition, it is cost-effective to use this aspect, and it covers the requirements

In the Auction System, several alternatives were considered for the `Persistence` aspect component, captured in Table 16. This traceability schema should also be used when capturing custom-designed aspect information.

Table 16 defines three alternatives (rows one to three) for realising the persistence component. The first alternative is to design the component from scratch. The next alterative is to reuse a persistence component provided by Oracle and the final alternative is to reuse a persistence component provided by Sybase. The second alternative is selected as Oracle software is already being used which ensures technical conformance, the component meets requirements and the component is cost-effective.

Table 17. Decisions for dealing with a postponed verb-based requirement

Alternative	Detail for multilingual
<alternative “id=1”>	<designedbyModule=“Multilingual”> *Justification text:* define as a separate aspect theme so the translation behaviour can be applied to various parts of the interface
<alternative “id=2”>	<designedbyModule=“Message”> *Justification text:* Integrate the translation behaviour with the message displaying behaviour
<alternativeSelected “id=1”>	*Justification text:* It is better to define a separate `MultiLingual` theme as it is likely that translation behaviour will be used across the interface

Postponed crosscutting requirements can map into aspect themes, several design artefacts or influence some decisions, depending on their nature. For example, the verb-based postponed requirement `R26` maps into an aspect theme `Multilingual`. Several alternatives are possible, as before, which should be recorded using the traceability schema.

Table 17 defines two alternatives (rows one and two) for realising the postponed `Multilingual` theme. The `<postponedRequirement id="R26">` tag identifies the requirement as postponed. The alternatives include the creation of a new

Multilingual theme to represent a design to satisfy R26 or the extension of the Message theme with this behaviour. The first alternative is selected as it enables the behaviour to be reapplied in various places rather that only being available to messaging behaviour.

Table 18. Decisions for dealing with a postponed adjective-based requirement

Alternative	Detail for performance **Identifying Tag:** <postponedRequirement id="R23">
<alternative “id=1”>	<constrains=“Bid”> *Justification text:* Only use lightweight algorithms and data structures and do not allow heavy behavioural flows that are heavy weight crosscut the Bid behaviour

A postponed adjective-based crosscutting requirement may not map into any specific design artefact, but could influence decisions adopted over other artefacts. For example, the postponed Performance aspect theme could influence the design of aspects similar to the manner in which it influenced the design of components, as illustrated in Table 18.

As mentioned in Sect. 5.3.1, the Performance aspect theme crosscuts themes that are mapped to the AuctionSystem component, such as the TransferCredit theme. In Table 15, we see that the relationship between the postponed Performance theme and the TransferCredit theme at the requirements level influences the design of the AuctionSystem components design.

5.4 Using the Traceability Information for Supporting Change Management

Section 4.4 illustrated how traceability information supports making changes to an architecture that may be required because of changes to the requirements. Of course, such architectural or requirements changes may also have an impact on the design. Here, we illustrate these impacts by considering the same stimuli considered at the architectural stage; changing requirements and changing system contexts.

A requirements change that has an impact on the architecture is likely to result in the introduction, removal, replacement or modification of any of the elements in specified in the component architecture.

5.4.1 Architecture Change Management

In Sect. 4.4.1 we illustrate how traceability information can be used to determine the impact of change in requirements at the architectural level. Changes in architecture must be propagated into design to ensure that the design conforms to the architecture.

The architectural change based on the alteration to R7 in Sect. 4.4.1 affects the VirtualBank component. Originally, the VirtualBank component could interact with credit card companies to perform credit transfers. The change requires the facility to perform both debit and credit card transfers. As illustrated in Table 14 the VirtualBank component is realised by a pre-built component MyVirtualBank which supports credit transfer. Table 14 illustrates that there are other reuse

alternatives that satisfy the changed requirements. In this instance, another component named C&DVirtualBank is available that satisfies the changed architecture.

When this change is made all design themes in which the VirtualBank component is represented are changed. The component can be represented as an empty (reused) component in some themes, adapted in others or in a theme where a connector attached to the component is detailed. When these elements change the composition relations that specify how these elements should be composed also changes.

5.4.2 System Context Change Management

In Sect. 4.4.2 we describe how changing system contexts can result in architectural changes. In that section, we describe how the architectural decision to postpone dealing with the performance theme is overturned. The architecture is changed to add a new load balancing and replication server (or cluster). This architectural change causes several changes to the system design.

The Performance theme added constraints to the design of the AuctionSystem component and the VirtualBank component (see Table 11). When the Performance theme is addressed at the architectural level, this removes the constraints on these designs. An example of these constraints is documented in Table 18. Once the Performance theme is addressed at the architectural level this allows new design decisions to be made in the design of the AuctionSystem and VirtualBank components. As illustrated in Table 15, alternatives when designing the AuctionSystem component were selected based on the presence of the performance constraints. With these constraints lifted other alternatives may become preferable.

Another example of how changing system contexts can result in architectural changes is where the company using the Auction System changes its Oracle DBMS to an in-house option. As illustrated in Table 16 the OraclePersistence was selected to realise the Persistence component because the company has already Oracle running. When the presence of an Oracle database is no longer a factor in the decision making process, the alternatives available need to be reconsidered. In this case the decision is to design a new Persistence component from scratch. Using the traceability information available, the designer can re-engineer the design themes to which the Persistence component is mapped to ensure the design is fully aligned with the component architecture.

6 Related Work

This paper describes an approach for addressing the early phases of software development using AO approaches. The intention of this approach is not to reinvent existing state-of-practice OO methods, but to illustrate how they can be extended to better modularise crosscutting concerns across early phases of the development lifecycle. Object-Oriented methodologies [7, 15–18] have been developed to outline best-practices [23] and provide guidance in the development of OO software. These methods describe how to develop architecture and design OO models from requirements. Each method provides mappings between model elements used in

requirements, architecture and design. These mappings are used to guide the development of architectures from requirements and detailed designs from architectural designs.

The Unified Process [15] and Catalysis [18] are OO software development methodologies based on contemporary best practices. These best practices include the use of component-based architecture, visual modelling and change management. Both the Unified Process and Catalysis describe how requirements are analysed and engineered into models from which architectures and detailed OO designs can be developed. Development in these methods is based on mappings between the requirements, architecture and design models based on the OO paradigm. These methods do not provide guidance for using AOSD approaches to modularise crosscutting concerns across the development lifecycle. Our approach extends existing OO methods and follows the best practices that they outline. It is component based, uses visual modelling and supports change management through improving traceability between models. The approach extends existing OO methods to provide support for AOSD early in the development lifecycle.

There are a number of separate approaches that provide AO solutions at requirements, architecture or design phases of development [2]. A major subset of these approaches support the identification, modularisation and composition of crosscutting concerns in only one development phase [24–27]. The approaches in the remaining subset, which includes [3, 28–31], cover more than one development phase and provide mappings between models specified at each phase.

Grundy [28] and Jacobson and Ng [29] define comprehensive approaches that provide consistent support for AOSD in all three early phases of development. Grundy's approach is an extension of a traditional component based method which incorporates AO principles. Although this approach provides a mapping from AO requirements to AO architectures, the design of the component's internals is an OO design. In this approach the specification of crosscutting modules is only possible at the architectural level. At the architectural level the specification of crosscutting is based solely on component interfaces. As such, concerns that crosscut the internal design of a component cannot be separated into distinct modules during design. While this approach enforces the principle of encapsulation [32] it reduces the degree to which crosscutting concerns can be both modularised and composed. This differs to our approach, which facilitates the specification of crosscutting concerns at both the architectural level as aspect components and at the design levels as aspect themes. Because of this, our approach supports both encapsulations at the component level but also the modularisation of crosscutting concerns at design level.

Jacobson and Ng's use case driven approach [29], in contrast, supports crosscutting at the design level but not at the architecture level. As such, no distinction is made between crosscutting in the architecture and design models. The concept of architecture is weakly defined in this approach. In addition, the mapping between use cases and architecture is unclear and the guidelines to develop AO architectures from AO use cases are not precise. Our approach makes a clear distinction between crosscutting at the architecture and design levels. Moreover, our approach defines a clear mapping between AO architectures from AO requirements and provides guidelines for designing architectures from requirements.

Clarke and Baniassad [3], Magno and Moriera [30] and Sánchez et al. [31] define early aspect approaches that provide mappings between requirements and design or requirements and architecture.

The Theme approach, from Clarke and Baniassad, defines mappings between the requirements and design models. This approach supports the definition of a concern and composition architectures [33] but not component architectures. The inclusion of component architectures and mappings from AO requirements and to AO design follows best practices outlined in existing OO methods. Concern and composition architectures also do not facilitate reasoning over deployment. Components are units of deployment [14]. By including component architectures in our integrated approach, deployment can be considered at the architectural development stage.

Magno and Moriera [30] and Sánchez et al. [31] provide approaches which map AO requirements to AO component architectures. In their work, Magno and Moriera [30] introduce the requirements engineering to software architecture framework. This approach focuses on the resolution and trade-off analysis between concerns. An example of a trade-off is a need to satisfy two competing requirements such as performance and encryption. This approach improves the extent to which AO architectures can be derived from the AO requirements through a preparatory engineering approach. The approach focuses on the mapping of adjective-based aspect requirements and to a number of possible alternative AO architectures.

Sánchez et al. [31] also present a similar approach for deriving AO architectures from AO requirements. In this approach model transformations are defined to automatically create AO architectures from AO requirements. Each model transformation encapsulates a means for deriving a potential architectural solution for aspect requirements from a set of alternatives. The approach enables the selection of an appropriate alternative within the alternative space. Once selected this alternative automatically creates a piece of architecture that satisfies the aspect requirements.

These approaches provide mappings from requirements to architecture but do not provide further mappings to design as in our approach. They can be used to complement our approach in terms of improving support for trade-off analysis and supporting automation of the integrated approach. A significant shortcoming of these approaches is that they do not support the postponement of aspect concerns at the architectural level. Postponement is important, as not all concerns need to be addressed at every stage of development. The concept of postponement allows for concerns to be addressed at the appropriate time. As our approach supports requirements, architecture and design phases of development it allows concerns that could be addressed during architectural design to be postponed when appropriate and addressed in the design phase of development.

Moreira et al. [20] present an AO requirements engineering approach, where a mapping from AO requirements to AO architecture is outlined. This mapping is used as the foundation for defining explicit XML based traceability support in Chitchyan et al. [19]. Jackson et al. [21] extend this XML base traceability support to cover mappings from architecture to design. In our approach we extend and combine the XML-based support traceability defined in Chitchyan et al. [19] and Jackson et al. [21]. We adapt these traceability schemas for the Theme/Doc–CAM–Theme/UML models and extend them to reflect the mapping between requirements, architecture and design.

The use of XML to record traceability links between models has also been proposed by Maletic et al. [34]. In their approach XML is used to record links between models. The choice for using XML for recording traceability information is based on the ease with which tooling can be provided for automating the recording of traceability information and visualising traceability links between models. We have followed this approach for the same reasons.

Our approach supports traceability between AO requirements, architecture and design. This is vertical traceability. As illustrated by Sabetzadeh and Easterbrook [35] and Van den Berg et al. [36] compositional traceability is also useful in AOSD where traceability links exist between elements that are specified to be composed. This traceability is horizontal traceability. Horizontal traceability is not currently supported in our XML based traceability solution. However, our approach is not constrained to vertical traceability and could be extended to provide horizontal traceability.

7 Discussion

In this section, we consider the benefits of AOSD at different phases of development and discuss the benefits our integrated approach. We also discuss how the traceability mechanisms should be used to ensure that the benefit of recording design rationale and alternatives outweighs the costs.

7.1 Benefits of AOSD

Initial results indicate that our integrated AOSD method supports improved maintainability over an OO counterpart. An evaluation of Theme/UML compares it with a state-of-the-art OO design method [37]. The comparison quantifies the variances in the maintainability of each method used in the design of a large scale digital publishing system. Both designs are based on the same set of use cases and as such both deliver equivalent functionality. The maintainability of each approach is determined by measuring various indicators of maintainability, including complexity, separation of concerns, size, cohesion and ease of change. Preliminary results from this stage of evaluation show that the AO method provides a better separation of concerns, lower design complexity, and provides a greater degree of maintainability than the OO method [38, 39].

Sant'Anna et al. [40] present a quantitative comparison of AO and traditional architectures (for multi-agent systems). The AO architecture design method that is used in this comparison is similar to CAM (detailed in Sect. 2.3), used in our integrated method. In this comparison, the modularity of AO and traditional architectures is quantified in the presence of architectural crosscutting concerns. The comparison is based on two medium-sized applications from which fundamental indicators of modularity are measured. These indicators include separation of concerns, coupling, component cohesion and interface simplicity. The results from empirical experiments reveal that the AO architecture improves separation of concerns and component cohesion (with a 50% of improvement compared to the non-AO version) and helps to considerably reduce

cohesion and interface complexity.[5] These measures provide an initial empirical validation that AO architectures improve the degree to which architectural components can be decoupled, scalability through reducing complexity and maintainability by providing better variability and adaptability. The same authors extend this empirical evaluation to cover a third system, with has produced similar results [41]. This work strengthens their initial validation and identifies that decoupling of components (which are used to modularise crosscutting concerns such as Persistence) is the main contribution of architectural aspect-orientation. The evaluation process of Sant'Anna et al. has also been applied to the CAM approach [4], where the architecture of a Pictionary game was the case study [42]. The results of this evaluation further strengthens the initial validation through its findings that the AO architecture provides better modularity in terms of separation of concerns, component coupling, component cohesion and interface complexity than the non-AO version.

Besides these initial studies, empirical evidence of the benefits of early aspect approaches is generally lacking. So too is guidance on the appropriate use of aspects at early stages of development. Existing empirical evaluations [43, 44] of AOSD have been focused on the implementation stage rather than the early stages of development. These studies have illustrated that AO solutions are not always optimal and have offered some basic guidance to aid the AO practitioner to make the right decisions during implementation.

Lopez and Bajracharya [43] have investigated the effects of aspect modularisation. Their investigation was conducted by assessing various non-aspect and aspect based evolutionary modularisations of an application. Each modularisation was quantified using design structure matrices to represent the effects of modularisation and net option value analysis is used to measure these effects. By comparing the results of the different modularisations they showed that aspect modularisation adds value to designs when used to reduce dependencies between modules. They also, however, showed that aspects have a negative influence when aspect modularisation adds otherwise avoidable dependencies into a design.

Garcia et al. compare AO and OO implementations of twenty three gang-of-four [44] design patterns. The comparison is based on metrics drawn from AO and OO implementations. These metrics are used to measure separation of concerns, scalability, coupling, cohesion and size. Significant findings of the study are that the implementation of all patterns is not improved when aspects are applied. While some patterns show improvements in the metrics when implemented using aspects, others are better as OO implementations.

In general, additional work needs to be done to provide evidence that aspects are beneficial and to provide clearer guidance to identify when and how aspects should and should not be used at early stages of the development process. The method presented in this paper begins to provide a structured approach to ensure that aspects are used at the right time. Only when more empirical evaluations of early aspects methods are conducted can we begin to fully quantify the benefits that can be derived

[5] The complete results of the empirical experiments can be found in http://www.lancs.ac.uk/postgrad/figueire/co_metrics/

from the integration of these methods and provide more in-depth guidance to the AOSD practitioner early in development lifecycle.

7.2 Benefits of Traceability

In this work we provide a traceability mechanism for recording: design decisions; the rationale for these decisions; and the alternatives considered before reaching a final decision. Recording alternatives and decision rationale supports improved design evolution and the maintenance process. The degree to which improvements can be achieved are based on recording information that will improve maintainability and ensuring benefits of recording are not outweighed by the associated costs. In Sect. 7.2.1 we discuss the factors that need to be considered when balancing the cost and benefits of recording traceability information. Section 7.2.2 puts our traceability schema in context.

7.2.1 Balancing Traceability Costs and Benefits

Ramesh and Dhar [45] are in agreement with Potts and Burns [46] when they argue that the maintenance of software should be done at the level of design specifications early in the development process and propagated to the implementation. This argument is borrowed from Blazer [47], who states that specifications (such as designs) are less complex and more localised than implementation. Both Ramesh and Dhar, and Potts and Burns provide means for recording the rationale in the decision making process. They claim that the benefits of this include:

1. Improved communication of decisions and the rationale behind them;
2. Assurances that over time the rationale behind certain decisions is not confused or lost;
3. Avoidance of critical errors by analysing rationale as well as designs;
4. Definition of contingency alternatives in the face of expected change;
5. Provision of support for change impact analysis.

Although there are advantages to recording traceability information there are also associated costs. Recording traceability information requires an additional effort that implies an extra cost. This means that initially, a development process in which traceability information is recorded incurs additional costs, and as the costs and time are needed to deal with change are not encountered, the benefits of recording traceability information are not immediately evident. However, as a system evolves the costs of recording traceability information remains relatively constant and the costs and time needed to address change are reduced. The reduction in costs and time needed for maintenance effort is due to the availability of the recorded design rationale considered and associated contingencies to be taken in the face of change. This means that the benefits of recording traceability information will increase as the system evolves. Consequently, recording traceability makes sense for systems that are expected to change.

In our work we focus on realising advantages 4 and 5 from the list above. Our traceability mechanism supports addressing expected changes. Our approach allows the specification of alternatives to a decision that is expected to change. These are

contingencies that can be applied when an expected change occurs. Changes that we do not expect are not supported using this mechanism. Our approach ensures that change is considered in the design process. It also ensures the reasoning and alternatives captured during development are not lost and can be analysed and used again during maintenance activities. As Brooks [48] points out, there is "no silver bullet" when attempting to deal with constant change. When recording traceability information, architects and designers must assess their development environment and anticipate what changes are likely. Traceability information will aid in dealing with changes that have been accurately anticipated. The accuracy with which change is anticipated is determinant of the degree to which the benefits of our approach will out weigh the costs.

Ahn and Chong [49] and Heidl and Biffl [50] address the issue of cost with respect to traceability benefits. They provide an approach to gauge the efforts that should be spent on recording traceability information. Their approaches follow the value-based software engineering (VBSE) approach presented by Boehm and Huang [51]. VBSE advocates the adaptation of the development process to ensure that the costs spent on an activity are proportional to the benefits earned. The work from both Ahn and Chong and from Heidl and Biffl define processes that quantify the levels of traceability effort proportional to the benefits of traceability. These processes balance parameters that include project size, value, risks, number of traces, required precision and costs. To provide guidance in the recording and management of traceability information using our traceability mechanism we suggest that a value-based approach should be followed.

7.2.2 Extensible Traceability Schema

The structures that underlie our traceability mechanism are based on those devised by Potts and Burns [46] and Ramesh and Jarke [52]. Potts and Burns outline generic structures "for representing design deliberation and its relation to the derivation of method specific artefacts". They adopt an issue-based model where decisions related to specific issues are recorded. An issue is comparable to a concern that needs to be addressed through design decisions. In their model, Potts and Burns define structures for capturing alternatives, justifications for decisions taken and the artefacts that they affect. Our traceability mechanism conforms to this model but is specialised to refer to concerns and the specific artefacts used in our method.

A conclusion of Ramesh and Jarke in a review of their own empirical studies, and of the state-of-art and state-of-practice, is that the structures needed for the collection of traceability information are based on the particular development context. Based on these reviews, they developed a high-level reference model for traceability that can be used to derive traceability mechanisms tailored for various development contexts. A significant sub-model of this reference model is a rationale model that is used to record alternatives encountered and justifications for decisions taken during development. The description of the traceability mechanism in this paper illustrates a conformant method for recording traceability information. Our traceability mechanism is highly extensible and can be expanded or adapted inline with the Ramesh and Jarke's traceability reference model that caters for different development contexts. As such, our traceability mechanism represents a core that can be extended

to ensure it can be used to record any information that will lead to improvements in maintainability.

8 Summary and Future Work

This paper has presented an integrated approach for deriving aspect-oriented applications from requirements to architecture to design, which is focused specifically on Theme/Doc, CAM and Theme/UML. The aim of this process is to assist developers in handling crosscutting properties at the right time. The basis is the definition of mappings from requirements (Theme/Doc) to architecture (CAM) and from architecture to design (Theme/UML). The modelling languages used in this work are representative of the early aspects community and were selected because of our experience with them. Concepts and ideas that appear in the three approaches can be easily generalised for other AO approaches. We have investigated the different meanings of aspects across the three software lifecycle phases, identifying and explaining which aspect properties should be specified at each stage, and which constructs of Theme/Doc, CAM and Theme/UML reflect these properties. We presented a description of the decisions software engineers must make at each stage.

We illustrated the mappings using an Auction system example. The mappings presented include base (non-aspect) elements of each approach, as these are prerequisite for mapping aspect information. The aspect mapping benefits from a categorisation of aspects into verb-based (defining crosscutting behaviours) and adjective-based (defining crosscutting properties). This categorisation is one of the contributions of this paper. Aspect mapping is considered for each category separately, since their different natures necessitate different mapping heuristics. Finally, we relate this derivation process to traditional processes, like the Unified Process [15], and Aspect-Oriented Methodologies, like [3, 29].

When mapping AO Requirements to AO Architectures and to AO Design, several alternatives may exist. Consequently, software engineers must make some decisions, considering the corresponding advantages and disadvantages. Recording each decision description, together with its advantages, disadvantages, rationale for the considered alternatives and justification for final decision selection is crucial for the management of future changes to a system. This is particularly important when mapping adjective-based aspects, because lots of alternatives are likely, and the justification for each alternative is often not straightforward. An XML-based traceability file (an improved version of the one presented in Jackson et al. [21]) has been used in this paper. The way in which traceability information helps to deal with change is also illustrated.

Ideally, an end goal of this work is to fully automate this process. However, this is only possible when the mapping rules are precisely defined. In this sense, some of the mapping rules of our process can be mechanically applied when the correspondence to target artefacts is clearly identified. For example, constructing aspect themes in Theme/UML from CAM aspect components can be performed without too much effort since a one to one mapping exists. On the other hand, the application of other rules, for example the mapping rules from Theme/Doc to CAM, require a bigger effort. Translating requirements to architecture implies the interpretation of the

stakeholders' aims by the architect, and so it is not always possible to define precise guidelines for a one to one mapping. In this case, expertise is required in the consideration of different alternative mappings or target artefacts. For instance, a Theme/Doc entity might be a CAM component or not, adjective-based themes can map into aspect components, architectural decisions or even be postponed. Some decisions have to be made before applying a specific mapping rule, so similarly to traditional methodologies; it is not always possible to fully automate such application derivation processes.

The lack of automation does not mean we have not achieved our initial goals. Our aim was to provide an integrated process for AO requirements, AO architecture and AO design. To the best of our knowledge, there is no process that integrates these three stages and that also deals with recording traceability information. A systematic and precise mapping, as provided in this paper, can be considered the first step towards the development of automatic model-driven transformations [53, 54]. Initially, the process presented in this paper has to be applied manually, as is the case with most non Model Driven Development processes, like Unified Process [15] or Catalysis [18].

As future work, we plan to develop tool support, in particular, model-driven transformations for automating the mappings as much as possible, and keeping the traceability files updated and synchronised. Model-Driven Development (MDD) approaches [53, 54] could substantially improve the process of automating repetitive, laborious and systematic mappings where the expertise and experience of the software architect is not required. In addition, when assistance from software engineers is required, semi automatic model-driven transformations might be constructed.

As mentioned in Sect. 7.1, further work needs to be done to provide evidence that aspects are beneficial and to provide clearer guidance to identify when and how aspects should and should not be used at early stages of the development process. In our continuing work we will continue to evaluate the application of aspects early in development. We will also refine our method to provide clearer guidance on how to apply aspects at the right time.

References

[1] Filman, R., Elrad, T., Clarke, S., Aksit, M. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Reading (2004)

[2] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. AOSD-Europe Deliverable D11. Lancaster University (May 2005), http://www.aosd-europe.net/deliverables/d11.pdf

[3] Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley, Reading (2005)

[4] Pinto, M., Fuentes, L., Troya, J.M.: A Dynamic Component and Aspect-Oriented Platform. The Computer Journal 48(4), 401–442 (2005)

[5] Sendall, S., Strohmeier, A.: Specifying Concurrent System Behaviour and Timing Constraints using OCL and UML. In: Gogolla, M., Kobryn, C. (eds.) «UML» 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools. LNCS, vol. 2185, pp. 391–405. Springer, Heidelberg (2001)

[6] Rashid, A., Moreira, A., Araújo, J.: Modularisation and Composition of Aspectual Requirements. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, pp. 11–20 (March 2003)
[7] Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach, 1st edn. Addison-Wesley, Reading (1992)
[8] Soeiro, E., Brito, I., Moreira, A.: An XML-Based Language for Specification and Composition of Aspectual Concerns. In: Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS), Paphos, Cyprus, December 2006, pp. 410–419 (2006)
[9] Brinksma, E. (ed.): Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. ISO 8807 (1988)
[10] Lamsweerde, A.V., Dardenne, A., Delcourt, B., Dubisy, F.: The KAOS Project: Knowledge Acquisition in Automated Specification of Software. In: American Association for Artificial Intelligence (AAAI). Spring Symposium Series, Stanford University, USA, pp. 59–62 (March 1991)
[11] Garcia, A., Chavez, C., Batista, T., Sant'anna, C., Kulesza, U., Rashid, A., Lucena, C.: On the Modular Representation of Architectural Aspects. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 82–97. Springer, Heidelberg (2006)
[12] Pessemier, N., Seinturier, L., Duchien, L.: Components, ADL and AOP: Towards a Common Approach. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, Springer, Heidelberg (2004)
[13] Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E.: PRISMA: Towards Quality, Aspect-Oriented and Dynamic Software Architectures. In: Proceedings of the 3rd International Conference on Quality Software (QSIC), Dallas, USA, pp. 59–66 (November 2003)
[14] Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Reading (2002)
[15] Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process, 1st edn. Addison-Wesley, Reading (1999)
[16] Rumbaugh, J.R, Blaha, M.R, Lorensen, W., Eddy, F., Premerlani, W.: Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs (1990)
[17] Shaller, D., Mellor, S.J.: Object Oriented Systems Analysis: Modeling the World in Data. Prentice Hall, Englewood Cliffs (1988)
[18] D'Souza, D.F, Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading (1998)
[19] Chitchyan, R., Pinto, M., Fuentes, L., Rashid, A.: Relating AO Requirements to AO Architecture. In: Proceedings of the 7th Workshop on Early Aspects (EA), at the 20th International Conference on Objects-Oriented Programming, Systems and Applications (OOPSLA), San Diego, USA (October 2005)
[20] Moreira, A., Rashid, A., Araújo, J.: Multi-Dimensional Separation of Concerns in Requirements Engineering. In: Proceedings of 13th International Conference on Requirements Engineering (RE), Paris, France, August-September 2005, pp. 285–296 (2005)
[21] Jackson, A., Sánchez, P., Fuentes, L., Clarke, S.: Towards Traceability from AO Architecture Design to AO Design. In: Proceedings of the 8th Workshop on Early Aspects: Traceability of Aspects in the Early Life Cycle (EA), at the 5th International Conference on Aspect-Oriented Software Development, Bonn, Germany (March 2006)

[22] Pinto, M., Fuentes, L., Troya, J.M.: Supporting the Development of CAM/DAOP Applications: An Integrated Development Process. Software: Practice and Experience 37(1), 21–64 (2007)
[23] Jones, C.: Software Assessments, Benchmarks, and Best Practices, 1st edn. Addison-Wesley, Reading (2000)
[24] Han, Y., Kniesel, G., Cremers, A.B.: A Meta Model and Modelling Notation for AspectJ. In: Proceedings of the 5th Workshop on Aspect-oriented Modelling with UML (AOM), at the 7th International Conference on Unified Modelling Language - the Language and its applications (UML), Lisbon, Portugal (October 2004)
[25] Herrmann, S.: Composable Designs with UFA. In: Proceedings of the 1st Workshop on Aspect-oriented Modelling (AOM), at the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands (April 2002)
[26] Barais, O., Duchien, L., Cariou, E., Pessemier, N., Seinturier, L.: TranSAT: A Framework for the Specifcation of Software Architecture Evolution. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, Springer, Heidelberg (2004)
[27] Baros, J.P., Gomes, L.: Activities as Behaviour Aspects. In: Proceedings of the 2nd Workshop on Aspect-oriented Modelling (AOM), at the 2nd International Conference on Unified Modelling Language - the Language and its Applications (UML), Dresden, Germany (September-October 2002)
[28] Grundy, J.: Multi-Perspective Specification, Design And Implementation Of Software Components Using Aspects. International Journal of Software Engineering and Knowledge Engineering 10(6), 713–734 (2000)
[29] Jacobson, I., Ng, P.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley, Reading (2004)
[30] Magno, J., Moreira, A.: Concern Interactions and Tradeoffs: Preparing Requirements to Architecture. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, Springer, Heidelberg (2006)
[31] Sánchez, P., Magno, J., Fuentes, L., Moreira, A., Araújo, J.: Towards MDD Transformations from AO requirements to AO architecture. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 159–174. Springer, Heidelberg (2006)
[32] Parnas, D.L.: On the Criteria to be used in Decomposing Systems into Modules. Communications of the ACM 15(12), 1053–1058 (1972)
[33] Katara, M., Katz, S.: Architectural Views of Aspects. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, pp. 1–10 (March 2003)
[34] Maletic, J.I., Collard, M.L., Simoes, B.: An XML-Based Approach to Support the Evolution of Model-to-Model Traceability Links. In: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), at the 20th International Conference on Automated Software Engineering (ASE), Long Beach, USA, November 2005, pp. 67–72 (2005)
[35] Sabetzadeh, M., Easterbrook, S.: Traceability in Viewpoint Merging: A Model Management Perspective. In: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), at the 20th International Conference on Automated Software Engineering (ASE), Long Beach, USA, November 2005, pp. 44–49 (2005)
[36] Van den Berg, K., Tekinerdogan, B., Nguyen, H.: Analysis of Crosscutting in Model Transformations. In: Proceedings of the 2nd ECMDA Workshop on Traceability (ECMDA-TW), at the 2nd European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA), SINTEF Report (A219), Bilbao, Spain, July 2006, pp. 51–54 (2006)

[37] Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley, Reading (1997)
[38] Van Landuyt, D., Jackson, A., Op de beeck, S., Grégoire, J., Scandariato, R., Joosen, W., Clarke, S.: Aspectual vs. Component-based Decomposition: A Quantitative Study. In: Proceedings of the 1st Workshop on Aspects in Architectural Description (AARCH), at the 6th International Conference on Aspect-Oriented Software Development (AOSD), Vancouver, Canada (March 2007)
[39] Van Landuyt, D., Jackson, A., Op de beeck, S., Grégoire, J., Scandariato, R., Joosen, W., Clarke, S.: Aspectual vs. Component-based Decomposition: Comparing the maintainability of AO and CB design: A Quantitative Study. In: Proceedings of the 1st Workshop on Aspects in Architectural Description (AARCH), at the 6th International Conference on Aspect-Oriented Software Development (AOSD), Vancouver, Canada (March 2007)
[40] Sant'Anna, C., Lobato, C., Kulesza, U., García, A., Chavez, C., Lucena, C.: On the Quantitative Assessment of Modular Multi-Agent System Architectures. In: Proceedings of the Special Track on Multiagent Systems and Software Architecture (MASSA), at the 7th Net.ObjectDays Conference, Erfurt, Germany, September 2006, pp. 111–135 (2006)
[41] Sant'Anna, C., Figueiredo, E., Garcia, A., Lucena, C.: On the Modularity Assessment of Software Architectures: Do my architectural concerns count? In: Proceedings of the 1st Workshop on Aspects in Architectural Description (AARCH), at the International Conference on Aspect-Oriented Software Development (AOSD), Vancouver, Canada (March 2007)
[42] Tekinerdogan, B., Sant'Anna, C., Garcia, A., Figueiredo, E., Pinto, M., Fuentes L.: General Architectural Evaluation Process Metrics, AOSD-Europe Deliverable D85, University of Twente (May 2007), http://www.aosd-europe.net/deliverables/d85.pdf
[43] Lopes, C.V., Bajracharya, S.K.: Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 1–35. Springer, Heidelberg (2006)
[44] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: Modularizing Design Patterns with Aspects: A Quantitative Study. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 36–74. Springer, Heidelberg (2006)
[45] Ramesh, B., Dhar, V.: Supporting Systems Development by Capturing Deliberations During Requirements Engineering. IEEE Transactions on Software Engineering 18(6), 498–510 (1992)
[46] Potts, C., Burns, G.: Recording the reasons for design decisions. In: Proceedings of the 10th International Conference on Software Engineering (ICSE), Singapore, April 1998, pp. 418–427 (1988)
[47] Blazer, R., Cheatham, T., Green, C.: Software technology in the 90s: using a new paradigm. IEEE Computer 16, 39–45 (1983)
[48] Brooks, F.P.: No Silver Bullet: Essence and Accidents of Software Engineering. Computer 20(4), 10–19 (1987)
[49] Ahn, S., Chong, K.: A: Feature-Oriented Requirements Tracing Method: A Study of Cost-benefit Analysis. In: Proceedings of International Conference on Hybrid Information Technology (ICHIT), Cheju Island, Korea, November 2006, pp. 611–616 (2006)
[50] Heindl, M., Biffl, S.: A case study on value-based requirements tracing. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 60–69. Springer, Heidelberg (2005)

[51] Boehm, B., Huang, L.G.: Value-Based Software Engineering: A Case Study. Computer 36(3), 33–41 (2003)
[52] Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Transactions on Software Engineering 27(1), 58–93 (2001)
[53] Beydeda, S., Book, M., Gruhn, V.(eds.): Model-driven Software Development. Springer, Heidelberg (2005)
[54] Stahl, T., Voelter, M.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Chichester (2006)
[55] Lieberherr, K., Lorenz, D., Mezini, M.: Programming with Aspectual Components, Technical Report NU-CCS99 -01, Northeastern University (March 1999)

Guest Editors' Introduction: Aspects and Software Evolution

Walter Cazzola[1], Shigeru Chiba[2], and Gunter Saake[3]

[1] DICO, Università degli Studi di Milano, Italy
cazzola@disi.unige.it
[2] Tokyo Institute of Technology, Japan
chiba@is.titech.ac.jp
[3] Otto-von-Guericke-Universität Magdeburg, Germany
saake@iti.cs.uni-magdeburg.de

Software evolution and adaptation is a research area, as also the name states, in continuous evolution, that offers stimulating challenges for both academic and industrial researchers. The evolution of software systems, to face unexpected situations or just for improving their features, relies on software engineering techniques and methodologies, that often imply re-designing, refactoring and re-coding part of or the whole system. Nowadays, similar approaches are not applicable in all situations (e.g., for evolving non-stopping systems or systems whose code is not available) and different approaches are necessary.

Aspect-oriented programming is a quite young discipline that is steadily attracting attention within the community of object-oriented researchers and practitioners. This discipline provides the developers with powerful techniques to better modularize object-oriented programs by introducing crosscutting concerns in a safe and non-invasive way.

Evolution is, clearly, a functionality that crosscuts the code of many objects in the system and therefore could benefit from a better modularization. What is still unclear is how and to what extent software evolution can benefit from this better modularization and whether the aspect-oriented programming is suitable for this goal as it is or it has to be evolved as well.

Software evolution may benefit from a cross-fertilization with aspect-oriented programming in several ways. Features such as obliviousness, separation of concerns, and extensibility are likely to be of increasing relevance in the modern software evolution scenario, where the trend is towards systems that exhibit sophisticated functional and non-functional requirements; that are built from independently developed and evolved commercial off-the-shelf (COTS) components; that support plug-and-play, end-user directed reconfigurability; that make extensive use of networking and internetworking; that can be automatically upgraded through the Internet; that are open; and so on. Several of these issues bring forth the need for a system to manage itself to some extent, to inspect components' interfaces dynamically, to augment its application-specific functionality with additional properties, and so on.

From a pragmatic point of view, several aspect-oriented techniques and technologies lend themselves to be employed in addressing these issues. On a more

A. Rashid and M. Aksit (Eds.): Transactions on AOSD IV, LNCS 4640, pp. 114–116, 2007.

conceptual level, several key aspect-oriented principles could play an interesting role as general software design and evolution principles. Even more fundamentally, aspect-oriented programming may provide a cleaner conceptual framework than that underlying the rather 'ad-hoc' solutions embedded in most commercial platforms and technologies. Obliviousness of aspect-oriented programming makes it well suited to address problems such as evolution of legacy systems, customizable software, product families, and more. The scope of application of aspect-oriented concepts in software evolution conceptually spans activities related to all the phases of software life-cycle, from analysis and architectural design to development, reuse, maintenance, and, therefore also evolution.

It is our belief that current trends in ongoing research in aspect-oriented programming and software evolution clearly indicate that an inter-disciplinary approach would be of utmost relevance for both. Therefore, we felt the necessity of investigating the benefits that the use of aspect-oriented techniques on the evolution of object-oriented software systems could bring. On the other hand, we are interested in exploring the limits of these techniques that hinder their use in this context and how to extend (evolve) them to completely exploit their potentialities.

From these considerations raised the need for a venue to support circulation of ideas between these disciplines. The RAM-SE (Reflection, AOP and Meta-data for SW Evolution) workshop series, we organized, represented a good meeting point for people working in the software evolution area, and an occasion to present reflective, aspect-oriented, and meta-data based solutions to evolutionary problems, and new ideas straddling these areas, to provide a discussion forum, and to allow new collaboration projects to be established.

This special issue of the transaction on aspect-oriented software development devoted to cross-fertilization of aspect-oriented software development and software evolution is a further step in the amalgam of these two disciplines. The growing interest on the topic has been proved by the high number of submissions we have received and by their high quality. After a long reviewing process we have selected three of them that well illustrate some of the efforts currently done to simplify the current state of the art in the field:

- Vander Alves, Pedro Matos Jr, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho: Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming.
- Andy Kellens, Kim Mens, and Paolo Tonella: A Survey of Automated Code- Level Aspect Mining Techniques.
- Chunjian Robin Liu, Celina Gibbs, and Yvonne Coady: Safe and Sound Evolution with SONAR: Sustainable Optimization and Navigation with Aspects for System-Wide Reconciliation.

In the first paper, aspects are used to get product lines adaptability to different contexts. The second paper faces the problem of migrating the legacy object-oriented

code to an aspect-oriented representation, in particular how to automatically mine crosscutting concerns and refactoring the system to aspects. Finally, the third paper presents SONAR, a framework to explore and support system-wide evolution through aspect-oriented programming.

Of course, these works just represent *the tip of the iceberg* and still a lot of work has to be done to exploit the aspect-oriented potentials at the best to deal with the evolution of crosscutting concerns. Stay tuned, we have a feeling that we will see amazing things in the near future.

Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming

Vander Alves, Pedro Matos Jr., Leonardo Cole,
Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho

Informatics Center, Federal University of Pernambuco
P.O. Box 7851 - 50.732-970 Recife PE, Brazil
{vra,poamj,lcn,atv,phmb,glr}@cin.ufpe.br

Abstract. For some organizations, the proactive approach to product lines may be inadequate due to prohibitively high investment and risks. As an alternative, the extractive and the reactive approaches are incremental, offering moderate costs and risks, and therefore sometimes may be more appropriate. However, combining these two approaches demands a more detailed process at the implementation level. This paper presents a method and a tool for extracting a product line and evolving it, relying on a strategy that uses refactorings expressed in terms of simpler programming laws. The approach is evaluated with a case study in the domain of games for mobile devices, where variations are handled with aspect-oriented constructs.

1 Introduction

There are several approaches for developing software Product Lines (PL) [10]: proactive, reactive, and extractive [18]. In the proactive approach, the organization analyzes, designs, and implements a *fresh* PL to support the full scope of products needed on the foreseeable horizon. In the reactive approach, the organization incrementally grows *an existing* PL when the demand arises for new products or new requirements on existing products. In the extractive approach, the organization extracts *existing products* into a single PL.

Since the proactive approach demands a high upfront investment and offers more risks, it may be unsuitable for some organizations, particularly for small to medium-sized software development companies with projects under tight schedules. In contrast, the other two approaches have reduced scope, require a lower investment, and thus can be more suitable for such organizations. Although the extractive and the reactive approaches are inherently incremental, it should be pointed out that the proactive approach can be incremental as well. In this case, products are simply derived based on whatever assets are in the core asset base at the time (a *core asset* is an artifact used in the production of more than one product in a PL). However, there still needs to be a potentially high investment for this first increment and, although we do not need to have all core assets in hand before starting to build products, *all* such assets need to be designed and

A. Rashid and M. Aksit (Eds.): Transactions on AOSD IV, LNCS 4640, pp. 117–142, 2007.

planned. An interesting possibility is to combine the extractive and the reactive approaches. But, to our knowledge, this alternative has not been addressed systematically at the architectural and at the implementation levels.

In all approaches, variability management must be addressed in the domain: while focusing on exploiting the commonality within the products, adequate support must be available for composing PL core assets with product-specific artifacts in order to derive a particular PL instance. The more diverse the domain, the harder it is to accomplish this composition task, which in some cases may outweigh the cost of developing the PL core asset themselves.

This paper addresses the issues of structuring and evolving the code of product lines in variant domains. In particular, we present a method that relies on the combination of the extractive and the reactive approaches, by initially extracting variation from an existing application and then reactively adapting the newly created PL to encompass other variant products. The method systematically supports both the extractive and the reactive tasks by defining refactorings. Additionally, we show that such transformations are derived from simple Aspect-Oriented Programming (AOP) laws [11]. Further, we evaluate our approach in the context of an mobile game product line. Finally, we provide tool support for the method.

Indeed, there are a number of techniques for managing variability from requirements to code level. Most techniques rely on object-oriented concepts. These techniques, however, are well-known for failing to capture crosscutting concerns, which often appear in variant domains. Mobile games, in particular, must comply with strict portability requirements that are considerably crosscutting, thereby suggesting AOP to handle variation [1], which is explored in our method.

The next section describes our approach, including its strategy and both extractive and reactive refactorings. Section 3 then analyzes how some refactorings from our method can be derived from existing elementary programming laws. In Sect. 4, the case study evaluating the approach is presented. Tool support for the method is described in Sect. 5. We discuss related work in Sect. 6 and offer concluding remarks in Sect. 7.

2 Method

Contrary to the proactive approach, we rely here on a combination of the extractive and the reactive approaches. Our method [4] first bootstraps the PL and then evolves it with a reactive approach. Initially, there may be one or more independent products, which are refactored in order to expose variations to bootstrap the PL. Next, the PL scope is extended to encompass another product: the PL reacts to accommodate the new variant. During this step, refactorings are performed to maintain the existing product, and a PL extension is used to add a new variant. The PL may react to further extension or refactoring.

The method is systematic because it relies on a collection of provided refactorings. Such refactorings are described in terms of templates, which are a concise and declarative way to specify program transformations. In addition, refactoring

preconditions (a frequently subtle issue) are more clearly organized and not tangled with the transformation itself. Furthermore, the refactorings can be systematically derived from more elementary and simpler programming laws [11,17]. These laws are appropriate because they are considerably simpler than most refactorings, involving only localized program changes, with each one focusing on a specific language construct.

In the remaining of this section, we detail the steps of our strategy, explaining the extractive and the reactive processes, and their associated refactorings. In Sect. 3, we explain these refactorings in terms of more elementary program transformations.

2.1 Extraction

The first step of our method is to extract the PL: from one or more existing product variants, strategies based on refactorings (detailed in Sect. 2.3) extract core assets and corresponding product-specific adaptation constructs. These constructs correspond to aspects and possibly supporting classes (classes only appearing in one product). Figure 1 depicts this approach. In this case, only one core asset is shown, but in general there could be more. Additionally, during evolution of the PL, a product-specific asset could become a core asset, in which case it be used to derive at least two PL members.

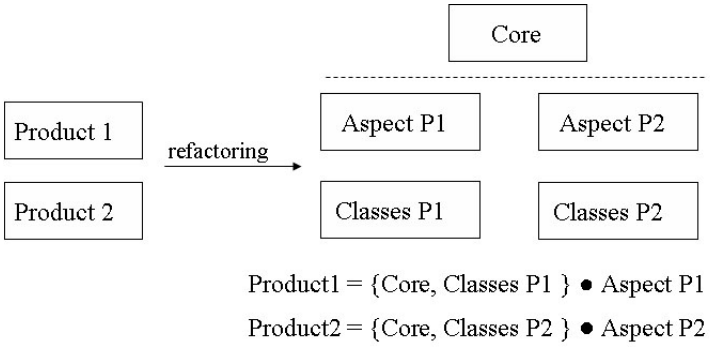


Fig. 1. Bootstrapping the Product Line. Core assets appear above the dashed line.

Product 1 and *Product 2* are existing applications in the same domain (for example, versions of a J2ME game for two platforms). *Core* represents commonality within these applications; it is usually a partially specialized OO framework, but can also contain aspects, in which case such aspects modularize the implementation of crosscutting concerns shared by at least two PL instances. The core is composed either with *Aspect P1* and its supporting classes (*Classes P1*), if any, or *Aspect P2* and its supporting classes (*Classes P2*), if any, in order to instantiate the original *specific* products. The • operator represents aspect composition (weaving). These aspects and their supporting classes thus encapsulate product-specific code.

In order to extract the variation within *Product 1* and *Product 2*, thus defining *Aspect P1* and *Aspect P2*, we must first identify it in the existing code base. When more than one variant exists, diff-like tools provide an alternative. In either case, however, such a view is too detailed at this point. Indeed, the developer first needs to determine the general concerns involved. This could be described more concisely and abstractly with concern graphs, whose construction is supported by a specific tool [24]. Concern graphs localize an abstracted representation of the program elements contributing to the implementation of a concern, making the dependencies between the contributing elements explicit. Therefore, the actual first step in identifying these variations is to build a concern graph corresponding to known variability issues.

Once the variability is identified, the developer should analyze the variability pattern within that concern. Depending on the pattern, a refactoring may be applied in order to extract it from the core (Sect. 2.3). Indeed, refactorings can be used to create product lines in an extractive approach, by extracting product-specific variations into aspects, which can then customize the common core [1,4].

Although our method focuses on code assets, it is important to describe its interaction with configurability-level artifacts, such as feature models [13]. Indeed, the method requires feature modelling and a configuration knowledge, which are essential for effectively describing the PL variability and product derivation. It is outside the scope of this paper to describe in detail the transformation at the feature model level; we specifically address this issue elsewhere [3].

The mapping between features and aspects needs to be specified by a configuration knowledge mechanism [13], which imposes constraints on features and aspect combinations like dependencies, illegal combinations, and default combinations. Constraints involving only feature combinations are also specified in the feature model. Throughout the paper, we use a configuration knowledge mapping groups of features to aspects and classes: the set of features common to both products map to PL core assets; the set of product-specific features map to product-specific aspects and supporting classes.

However, our method does not bind a particular configuration knowledge. Other mappings are possible, depending on the granularity of the features and aspects. For example, with finer-grained variability, single features-rather than feature subsets-could be mapped directly to aspects. Other examples of configuration knowledge can be found elsewhere [20].

2.2 Evolution

Once the product line has been bootstrapped, it can evolve to encompass additional products. In this process, a new aspect is created to adapt the core to the new variant. Moreover, a new feature is added to the feature diagram in order to represent the new product, and the configuration knowledge is updated to map the new feature to the new aspect (Fig. 2).

The refactorings in Table 1 (Sect. 2.3) can also be used for evolution. As Fig. 2 also indicates, the core itself may evolve because *some* of the commonality

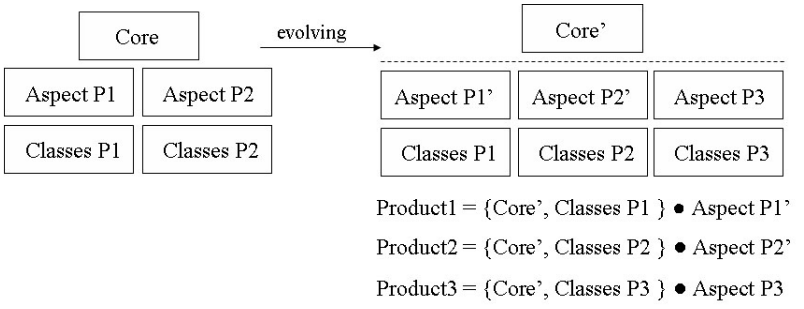


Fig. 2. Evolving the Product Line. Core assets appear above the dashed line.

between *Product 1* and *Product 2* might not be shared *completely* by *Product 3*. That is, *Product 3* has *different* commonality with *Product 1* and *Product 2* than these latter have with each other; therefore, a *slightly* different core is necessary. This may trigger further adaptation of the previously existing aspects, too. However, AspectJ tools can identify parts of the core on which these previous aspects depend, and some refactorings could also be aspect-aware according to the definition of Hanenberg et al. [15]: evolving the core may change some join points within it, so the aspect-aware refactorings accordingly adjust aspects' pointcuts to refer to the new join points, thereby minimizing the need to revisit such previous aspects. The end of Sect. 2.3 discusses how our refactorings could be extended to be aspect-aware according to this definition.

Another evolution scenario (Fig. 3) involves restructuring the product line to explore commonality within aspects. Such commonality (`AspectFlip` aspect) then becomes a core asset, since it is now explicitly shared by at least two PL instances.

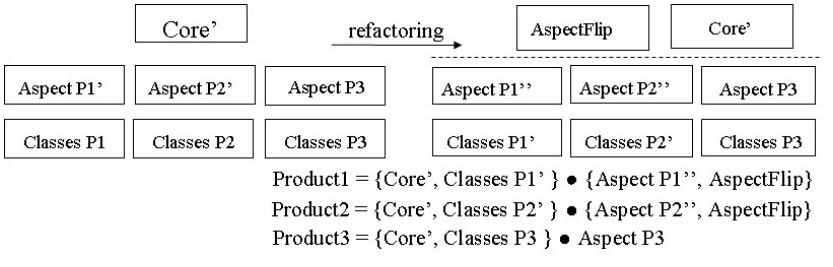


Fig. 3. Refactoring the Product Line. Core assets appear above the dashed line.

Figure 3 can become more complex with the addition of new platforms and identification of reusable aspects. However, constraints in the feature model as well as the configuration knowledge (the mapping of features to aspects) limit aspect combinations, thereby providing support for scalability.

2.3 Refactoring Catalog

This subsection defines a refactoring catalog, which is a set of refactorings supporting the extractive and the evolutive activities described in the previous subsections. Section 4 shows some strategies (sequence of applications of refactorings from this catalog) that manage to handle the implementation of variable features [13]. We have developed this catalog empirically by analyzing variability in a number of mobile games [1,4]. It has allowed us to address most variabilities in this domain, but we have not proved this catalog to be complete. We first specify the AspectJ subset necessary for applying these refactorings. Next, we motivate some refactorings by considering an example of feature extraction. Finally, we list the remaining refactorings.

AspectJ Subset. We consider a subset of AspectJ [11]. This simplifies the definition of transformations and does not compromise our results. However, this may limit the number of refactorings we are able to derive with our laws. For example, the use of `this` to access class members is mandatory. Also, the `return` statement can appear at most once inside a method body and has to be the last command. Additionally, we consider only the following pointcut designators: `call`, `execution`, `args`, `this`, `target`, `within` and `withincode`.

Restricting the use of `this` simplifies the preconditions defined for the laws. This can be seen as a global precondition instead of a restriction to the language. Most of the laws dealing with advice require this restriction. This restriction allows an easy mapping from the executing object referenced from `this` to the executing object exposed inside advice with the pointcut designator `this`.

We only support the mentioned pointcut designators because we think they may represent the core designators of this aspect-oriented language: they have sufficed for us capture join point in four different application domains in previous work [11] and in this work. Extending the set of laws to include other AspectJ constructs would be time demanding but not difficult. Besides, it would not affect the already defined laws. This is regarded as future work.

An Example. In the context of the mobile device game domain, we consider the optional figures concern of a game. We examine the code declaring and using the `dragonRight` image. First, we consider class `Resources`.

```
class Resources {...
   Image dragonRight;...
   void loadImages() { ...
      dragonRight = Image.createImage("dragonRight.png");...
  } ...
}
```

where such field is not used anywhere else in the class. The developer may decide that `dragonRight` is an optional feature specific to Platform 1 (P_1) and thus could extract it into an aspect with inter-type declaration and advice constructs. We would thus have

```
class Resources { ...
   void loadImages() {...}
}

Aspect AP1 {
 Image Resources.dragonRight;
 after() returning(): execution(Resources.loadImages()) {
   dragonRight = Image.createImage("dragonRight.png");
 } ...
}
```

where `Resources` now represents a construct in the game core being built and `AP1` denotes an aspect adapting it for a specific platform, namely P_1. The fact that the field is not used anywhere else in the class allowed us to move the attribution towards the method border (end of method in this case), which allows the variation to be described by a single after advice.

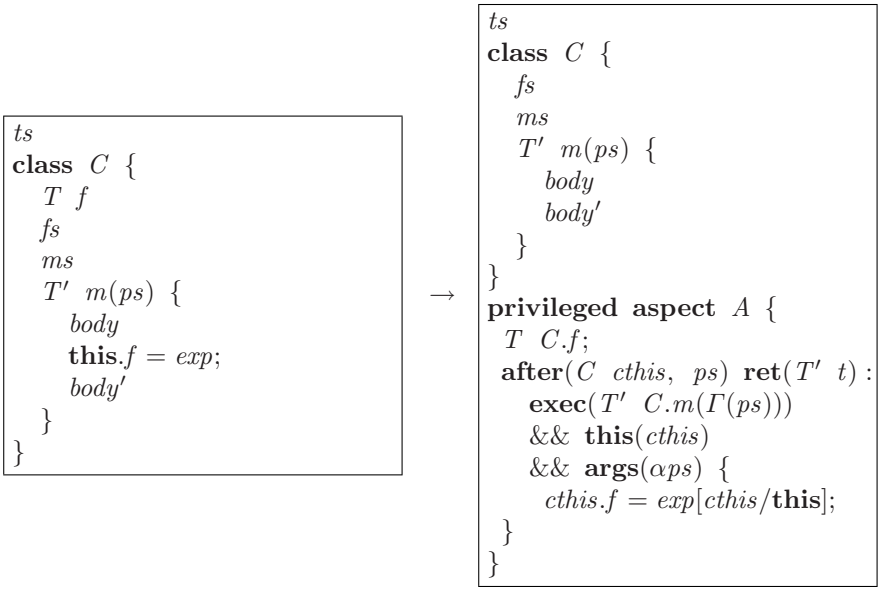
Refactorings like these occur frequently and we thus generalize them using a notation that follows the representation of programming laws [11,17]. Refactoring *Extract Resource to Aspect — after*, whose transformation template is shown shortly ahead, generalizes this transformation and has the purpose of extracting a single variant field, along with part of its usage, into an aspect.

On the left-hand side of Refactoring 1's transformation template, the `f` field and the `this.f=exp;` command (*exp* is an arbitrary expression) denote the variability pattern to be extracted. On the right-hand side, such variability is extracted into aspect `A`. Aspect `A` uses an inter-type declaration construct to introduce field `f` of type T (in the transformation template, T also encompasses the access modifier) into class `C` and an advice construct to add the extracted command to method `m`.

We denote the set of type declarations (classes and aspects) by *ts*. Also, *fs*, *ms* and *pcs* denote field declarations, method declarations and pointcut declarations, respectively. $\sigma(C.m)$ is used to denote the signature of method m of class C, including its return type and the list of formal parameters. $\Gamma(ps)$ denotes the type list from parameters *ps*, and αps denotes the parameter names from *ps*. For brevity, we write `exec` and `ret` instead of `execution` and `returning`, respectively.

Each refactoring provides preconditions to ensure that the program is *syntactically valid* (not necessarily syntactically equivalent) and *semantically equivalent* (behavior preserving) after the transformation. The first and second preconditions are necessary to ensure that the code still compiles after applying the transformation, whereas the last three preserve behavior. In particular, although the right-hand side of the refactoring template is not *syntactically* equivalent to the left-hand side, both sides are *semantically* equivalent, since the third refactoring precondition (shown shortly ahead) guarantees that the `this.f=exp` command can be the last one or in the middle of method `m`.

Refactoring 1 ⟨Extract Resource to Aspect — after⟩



provided

- `A` does not appear in *ts*;
- if the field `f` of class `C` is private, such field does not appear in *ts* nor in *ms*;
- `f` does not appear in *body'*; `exp` does not interfere with *body'*;
- `A` has the highest precedence on the join points involving the signature $\sigma(C.m)$;
- there is no designator `within` or `withincode` capturing join points inside `this.f=exp;`

In the preconditions above, we require that, if the field `f` of class `C` is private, such field does not appear in *ts* nor in *ms* because, when moved to the aspect, the field would be private with respect to the aspect and not with the class, hence a reference to `f` in *ts* or *ms* would not compile (according to AspectJ semantics, visibility modifiers of inter-type declarations are related to the aspect and not to the affected class).

The preconditions on the third bullet are necessary to allow moving the command `this.f=exp;` to the end of method `m`, which is done as an intermediate step during refactoring. Section 3.2 and Fig. 4 explain the refactoring in terms of consecutive applications of elementary fine-grained transformations. The precondition requiring `exp` not to interfere with *body'* is specified at a semantic level, but it can also be specified syntactically if we have further information about the structure of *exp*, which happens frequently, including in our example above and in our case study. In such cases, *exp* is a static method call on third-party API to load image attributes, thus not interfering with *body'*.

Despite its syntactic form, the semantic intent of the higher precedence precondition is the following: the newly created after *advice* has the highest precedence on the join points involving the signature $\sigma(C.m)$. However, the only way AspectJ allows specifying precedence among advice of different aspects is by specifying precedence on *aspects* containing these advice, thus implying that *all* advice of a certain aspect `A` have precedence over *all* advice of another aspect `B`, which is a too coarse-grained way to do so. In fact, we may want some advice of `A` to have precedence over some advice of `B` and some advice of `B` to have precedence over advice of `A`, which would lead to an unsolvable constraint among the precedence of such aspects.

Therefore, applying the same refactoring twice works if the code is extracted into the same aspect (advice precedence within the aspects is addressed as shown shortly ahead); otherwise, it will depend on whether there is already a precedence constraint on the existing aspects. If so, the refactoring could not be applied; otherwise, the refactoring can be applied and the new aspect `A` will have the highest precedence. This is a limitation of AspectJ's expressiveness. An AspectJ extension could be accomplished to define advice precedence on a fine-grained approach, by using the ABC compiler [6], for example. In this case, the semantic intent of the refactoring could be expressed syntactically.

The fifth precondition means that there are no `within` or `withincode` pointcut designators in any aspect in the PL that could match join points in the `this.f=exp;` statement. This precondition is necessary because moving such statement may break those pointcuts. Despite declarative, this precondition is verifiable by examining the PL aspects in the IDE using AJDT's API.

The refactoring described creates aspect `A`. A slight variation of this refactoring assumes `A` already exists. In this case, such aspect would have a particular form after applying the transformation:

```
privileged aspect A {
T C.f;
pcs
bars
afs
after(C cthis, ps) ret(T' t):
   exec(T' C.m(Γ(ps)))
   && this(cthis)
   && args(αps) {
     cthis.f = exp[cthis/this];
 }
}
```

Note that, in this case, the advice can not be considered as a set, since order of declaration dictates precedence of advice. According to the AspectJ semantics, if two advice declared in the same aspect are `after`, the one declared later has precedence; in every other case, the advice declared first has precedence. Thus, we divide the list of advice in two. The first part (*bars*) contains the list of all `before` and `around` advice, while the second part contains only `after` advice (*afs*). This

separation ensures that `after` advice always appear at the end of the aspect. It also allows us to define exactly the point where the new advice should be placed to execute in the same order in both sides of the refactoring. Additionally, for advice declared in different aspects, precedence depends on their hierarchy or their order in a `declare precedence` construct (this is addressed by the fourth precondition of the refactoring). Similar considerations apply to the remaining refactorings. For brevity, we will assume the aspect is created in each case.

Remaining Refactorings: Table 1 summarizes all refactorings from our catalog.

Table 1. Summary of refactorings

Refactoring	Name
1	Extract Resource to Aspect — after
2	Extract Method to Aspect
3	Extract Context
4	Extract Before Block
5	Extract After Block
6	Extract Argument Function
7	Change Class Hierarchy
8	Extract Aspect Commonality

Some of the refactorings in Table 1, such as *Change Class Hierarchy*, are coarse-grained; others, such as *Extract Argument Function*, are fine-grained; some, such as *Extract Method to Aspect*, have medium granularity. Part of their names refers to an AspectJ construct that encapsulates the variation. For example, the *Extract Resource to Aspect — after* we described previously extracts the variant part of a concern, appearing as a field and its uses in the class, into AspectJ's `after` construct. Finally, the refactorings we present are not aspect-aware according to the definition of Hannenberg et al. [15], but these could be adapted to be so by relaxing some preconditions such as the fifth of the *Extract Resource to Aspect — after* refactoring and accordingly changing the `within` and `withincode` pointcuts involved following the guidelines presented elsewhere [15]. In a broad sense, however, our refactorings are aspect-aware, since they can be used in the presence of aspects and manipulate aspects constructs in transformation templates and preconditions.

3 Formal Reasoning for AspectJ Refactorings

This section analyzes how some aspect-oriented extractive refactorings can be decomposed into or derived from existing elementary programming laws [11], which are simpler and easier to reason about than the refactorings, thereby increasing correctness confidence in such extractive transformations. This is specially relevant because it reduces the burden on testing, which is extremely expensive in the PL scenario. Section 3.1 reviews some existing fine-grained aspect-oriented

programming laws [11]. Then, in Sect. 3.2, we relate such refactorings and laws by showing how the former can be described in terms in of the latter. This is illustrated by decomposing refactoring *Extract Resource to Aspect — after* of Section 2.3 into a set of programming laws.

3.1 Programming Laws

Programming laws [11,17], like refactorings, are transformation structures which preserve program consistence and behavior. In contrast, they are much simpler than most refactorings: they involve only localized program changes, and each one focuses on a specific language construct.

Differently from refactorings, laws can be applied not only from the left to right side, but also in the opposite direction. Therefore, there are different preconditions depending on the direction the law is used. This is represented by arrows, where the symbol (←) indicates that this precondition must hold when applying the law from right to left. Similarly, the symbol (→) indicates that this precondition must hold when applying the law from left to right. Finally, the symbol (↔) indicates that the precondition must hold in both directions.

For example, the following law has the purpose of adding an after advice. On the left-hand side of the law, *body′* is the last block of code to execute in method `m`. Thus, we can extract it to an after advice. On the right-hand side, *body′* is not present in method `m`, although it is executed after the execution of method `m` by an after advice declared in aspect `A`. In this aspect, the symbols used in the advice construct have the same meaning as in Refactoring 1.

Law 1 ⟨Add After-Execution Returning Successfully⟩

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
    body′
  }
}
privileged aspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
privileged aspect A {
  pcs
  bars
  afs
  after(C cthis, ps) ret(T′ t) :
    exec(T′ C.m(Γ(ps)))
    && this(cthis)
    && args(αps) {
      body′[cthis/this]
  }
}
```

provided

(→) *body′* does not use local variables declared in *body*; *body′* does not call `super`;
(↔) `A` has the highest precedence on the join points involving the signature $\sigma(C.m)$;
(↔) there is no designator `within` or `withincode` capturing join points inside *body′*;

The highest precedence precondition of this law is analogous to the highest precedence precondition of Refactoring 1, which was discussed in Sect. 2.3. Likewise, the last precondition of this law corresponds to the fifth precondition of Refactoring 1. Therefore, the constraint refers to any aspect.

Examining the left-hand side of this refactoring, we see that *body′* executes before all after advice possibly declared for this join point. This means that the new advice on the right-hand side of the law should be the first one to execute, preserving the order in which the code is executed in both sides of the law. Thus, the after advice should be placed at the end of the after list (*afs*). Moreover, in order to ensure that the new advice created with this law is the first one to execute, we have a precondition stating that aspect `A` has the highest precedence over other aspects defined in *ts*. This precondition must hold in both directions.

The next law represents the language construct which introduces a field into a class. Analyzing this transformation from the left to the right, we can see that `field` declaration is removed from class `C`. However, we introduce `field` in this class by using an inter-type declaration construct declared in aspect `A`.

Law 2 ⟨Move Field to Aspect⟩

ts
class *C* {
fs
T field
ms
}
privileged aspect *A* {
pcs
bars
afs
}
	=	*ts*
		class *C* {
		fs
		ms
		}
		privileged aspect *A* {
		T C.field
		pcs
		bars
		afs
		}

provided

(→) The field *field* of class *C* does not appear in *ts* and *ms*.

This precondition is necessary for the same reason that the second precondition of Refactoring *Extract Resource to Aspect — after* is necessary, which was explained in Sect. 2.3.

3.2 Deriving Refactorings

In this section we use aspect-oriented programming laws [11] to show that the refactorings previously discussed in Sect. 2.3 are behavior preserving transformations. Although we do not conduct a strictly formal proof, the derivation is still useful for understanding refactorings in terms of simpler transformations. Additionally, representing the refactorings as a composition of programming laws helps to better define the preconditions under which those refactorings are valid. For their simplicity, programming laws [17] are suitable for this. A complete formal proof requires establishing the validity of the laws with respect to a formal semantics, which is still on going work [12].

The laws we use (defined elsewhere [11]) consider the entire context, and therefore apply to *closed* programs. Nevertheless, their associated side conditions are purely syntactic. Furthermore, although the context is captured for each particular law application, this is by no means a requirement that the context be fixed for successive transformations. If, eventually, a modified context no longer satisfies the conditions of a law previously applied, this does not invalidate the effected transformation; it just means that in the current context the application of the law would not be valid. Accordingly, the laws compose in the sense that their consecutive application is equivalent to a coarse-grained transformation (refactoring). Indeed, such composition is not as flexible as in Hoare's laws [17]—which can be applied to *open* programs—, but has sufficed to derive the refactorings.

For example, Refactoring 1 (*Extract Resource to Aspect — after*), presented in Sect. 2.3, can be represented as a sequence of object-oriented transformations and aspect-oriented programming laws (Fig. 4). In this case, starting from the left-hand side template of this refactoring, we first need to rearrange the source code manipulating field `f` because AspectJ does not provide any mechanism to

Table 2. Summary of Refactorings Derivations. Consecutive application of laws is represented by →.

Refactoring	Derivation of refactoring in terms of laws
Extract Method to Aspect	Extract Method → Move Method to Aspect
Extract Resource to Aspect — after	OO Refactoring → Add After-Execution Returning Successfully → Move Field to Aspect
Extract Context	Add Around-Execution
Extract Before Block	Add Before-Execution
Extract After Block	Add After-Execution Returning Successfully
Extract Argument Function	Add Around-Call
Class Hierarchy	Extend From Super Type
Extract Aspect Commonality	Change Advice Order → Move Advice[a] → Merge Advice

[a] Repeated application of a law

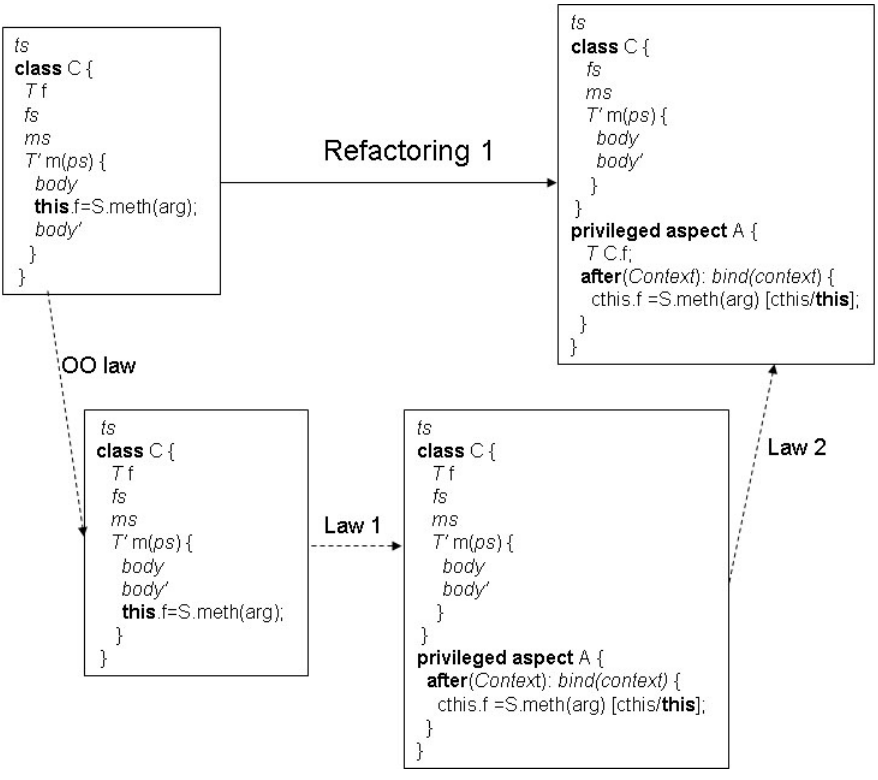


Fig. 4. Derivation of Refactoring *Extract Resource to Aspect — after*. The dashed lines denote application of programming laws (fine-grained transformations); the continuous line denote the application of the refactoring (coarse-grained transformation).

introduce crosscutting behavior in the middle of a method. In order to move the crosscutting code to an aspect, we first need to move the such code to the beginning or end of the method; this allows the creation of a `before` or `after` advice, respectively. In this refactoring, the crosscutting code was moved to end of the method (we name such transformation by OO law in Fig. 4). The OO law holds if the code to be moved is independent of the remaining method code, which is guaranteed by the third precondition of the Refactoring 1 (Sect. 2.3). Once the crosscutting code is at the end of the method, we can use Law 1 (*Add after-execution returning successfully*), mentioned in Sect. 3.1, to create a new advice that is triggered after the method's successful execution. At this point, Law 2 (*Move Field to Aspect*) can be applied to extract field `f` into the aspect. The summary of transformations necessary to accomplish this refactoring is shown in Fig. 4.

The remaining refactorings can be similarly derived from programming laws. In Table 2, each row summarizes the *derivation* of a refactoring whose name is on the first column (this matches the refactorings from Table 1) in terms of the

consecutive applications of aspect-oriented laws (defined elsewhere [11]) in the second column. In the table, consecutive application of laws is represented by $\rightarrow$, and repeated application of the same law is denoted with a superscript^a.

We notice that refactorings can have different levels of complexity when compared to laws. Some refactorings, like *Extract Aspect Commonality*, can be considerably coarse-grained, representing a combination of some laws. On the other hand, some refactorings, like *Extract Before Block*, can be mapped directly into a single law.

4 Case Study: Rain of Fire

Rain of Fire (RoF) is a classic arcade-style game where the player protects a village from different kinds of dragons with catapults. The game is a *commercial product* currently offered by service carriers in South America and Asia. Although it is less than 5K LOC, LOC is neither a necessary nor sufficient condition for complexity. In fact, complexity in the mobile game domain arises mostly due to variability. In general, the mobile game domain is highly variant due to a strong portability constraint: applications have to run in numerous platforms, giving rise to many variant products [2], which are under a tight development cycle, where proactive planning is often unfeasible to achieve.

Although the PL that actually exists in our industrial partner encompasses 12 members, in this case study we investigated how RoF was adapted to run in 3 platforms (P_1, P_2, and P_3), which encompass most variability issues in this PL. P_1 relies solely on MIDP 1.0, whereas P_2 and P_3 rely on MIDP 1.0 and a proprietary API. Some of the variability issues within these products are as follows: optional images, proprietary API for flipping images, screen sizes, and image loading policy. After applying our approach (details shortly ahead), the resulting PL has the feature model of Fig. 5, and the following instances, as described by the selection of features.

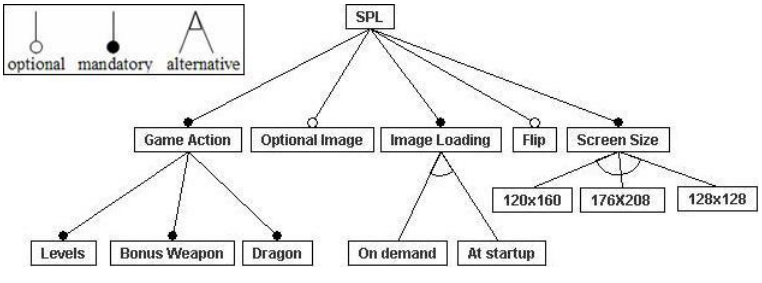


Fig. 5. Variability within Game Product Line

```
P1 = {Dragon, Bonus Weapon, Levels, Optional Image, At startup, Flip, 176x208};
P2 = {Dragon, Bonus Weapon, Levels, On demand, Flip, 120x160};
P3 = {Dragon, Bonus Weapon, Levels, Optional Image, At startup, 128x128};
```

Although this case study has focused only on 3 instances, the feature model shows that other configurations are also possible: the feature model has a total of 24 configurations. Future work is underway to address more configurations.

In order to evaluate our approach, we created a PL implementation of the three products and then compared the PL version with the original implementation of these products. To create and evolve the PL, we first identified the variabilities (such as optional images) with concern graphs and then moved their definition to aspects using the *Extract Resource to Aspect* refactoring. In another step, we addressed method body variability within the platforms. Accordingly, we made extensive use of the *Extract Method to Aspect* refactoring. The *Extract After Block* and *Extract Before Block* refactorings were used when the variant code appeared at the end or beginning of the method body. On the other hand, the *Extract Context* refactoring was used when the variation surrounded common code, representing a context to it. The *Extract Argument Function* refactoring was used when variation appeared as an argument for a method call. Finally, we used the *Change Class Hierarchy* refactoring to deal with class hierarchy variability.

During the evolution of the PL to include P_3, we had to deal with the *load images on demand* concern. This concern was specific to this platform, as it had constrained memory and processing power. To implement this concern, we had to define a method for each screen that could be loaded. Before a screen was loaded, the corresponding method was called. In contrast, in P_1 and P_2 implementations, the images were loaded only once, during game start-up. In this case, there was only one method that loaded all the images into memory. This situation illustrates the scenario in Fig. 2.

We addressed this by applying a sequence of *Extract Method* refactorings in the core to break the single method loading all images into finer-grained methods loading images for each screen; the call of this single method was then moved from the core to P_1's and P_2's aspects, and the calls to such smaller methods were moved to P_3's aspect by the *Extract Before Block* refactoring.

Another evolution scenario took place when we realized that some commonality existed between P_1 and P_2 with respect to the *Flip* feature (proprietary graphic API allowing an image object to be drawn in the reverse direction, without the need for an additional image): these two platforms are from the same vendor and share this feature, which is not shared by P_3, from another vendor. Therefore, the *Flip* feature is isolated in the corresponding aspects of P_1 and P_2, but it would be useful to extract this commonality into a single module. In fact, we were able to factor this out into a single generic aspect (`AspectFlip`) with the *Extract Aspect Commonality* refactoring, thus illustrating the scenario in Fig. 3.

Table 3 reports the occurrence of each refactoring for achieving the resulting PL. *Extract Method to Aspect* was the most frequently employed, since variability within method body was common for extracting most features. As the PL evolves, we expect to employ *Extract Aspect Commonality* more frequently.

For the resulting the PL, we also employed the *Move Field to Aspect* programming law from Sect. 3.1. This law was used 28 times. This is consistent with the results of Table 3, since we do not claim these to be complete (the argument in Sect. 3 is on soundness). Additionally, if we had only used the programming laws themselves instead of the refactorings (composition of the programming laws), we would have to apply approximately twice as many programming laws. In general, the method can combine the refactorings and the programming laws themselves. As the set of refactorings evolve closer to completeness, the direct use of the fine-grained programming laws is expected to decrease and the proportional use of the coarse-grained refactorings is expected to increase.

Table 3. Occurrence of each refactoring

Refactoring	Name	Occurrence
1	Extract Resource to Aspect — after	5
2	Extract Method to Aspect	41
3	Extract Context	1
4	Extract Before Block	2
5	Extract After Block	10
6	Extract Argument Function	1
7	Change Class Hierarchy	1
8	Extract Aspect Commonality	1

The resulting configuration knowledge maps sets of features to implementation artifacts:

```
{Levels, Dragon, Bonus Weapon} -> CoreClasses
{Flip} -> AspectFlip
{176x208, Optional Image, At startup} -> AspectP1
{120x160, On demand} -> AspectP2
{128x128, Optional Image, At startup} -> AspectP3
```

where *CoreClasses* is a set of core assets comprised of classes common to all products; `AspectFlip` is a core asset aspect dealing with the *Flip* feature; `AspectP1`, `AspectP2`, and `AspectP3` deal partially with specific products features of products P1, P2, and P3, respectively. The arrow notation means that the set of features to its left, which are from the feature model represented in Figure 5, map to the aspects or classes to its right. According to this configuration knowledge and to the configuration of each product presented previously, the PL instances are synthesized by

```
P1 = CoreClasses • {AspectP1, AspectFlip};
P2 = CoreClasses • {AspectP2, AspectFlip};
P3 = CoreClasses • {AspectP3};
```

where • denotes composition. According to this derivation of the PL members, the PL core assets consist of the following: 1) eighteen classes in *CoreClasses*; 2)

one core aspect (`AspectFlip`). P1 and P2 are each comprised of *CoreClasses* and two product-specific aspects; P3 is comprised of *CoreClasses* and one product-specific aspect.

Indeed, the configuration knowledge is coarse-grained: there are few reusable aspects across different PL instances. In fact, `AspectFlip` is the only reusable aspect. Current work is underway to explore finer-grained mappings (i.e., more reusable aspects across the PL) and more configurations. Additionally, some scalability issues include having to reduce the granularity of the aspects, so that these become reusable across more instances.

After creation and evolution of the PL, we analyzed code metrics. Table 4 shows the number of Lines of Code (LOC) for each product in the original implementation, in contrast with the PL implementation. We calculate the LOC of a PL instance as the sum of the core's LOC and the LOC of all aspects necessary to instantiate this specific product.

Table 4. LOC in original and PL implementations

Original implementation				**PL implementation**					
P_1	P_2	P_3	Total	**Core assets**		P_1	P_2	P_3	Total
				Core classes	Core aspects				
2,965	2,968	3,143	9,076	2,477	72	3,042	3,047	3,210	4,405

According to Table 4, LOC is slightly higher when comparing each PL instance with the corresponding product in the original implementation. This is caused by the extraction of methods and aspects, which increase code size due to new declarations. On the other hand, there is a 48% reduction in the total LOC of the PL implementation, when compared to the sum of LOCs of the single original versions. This was possible because of the core assets, which represent 57% of the PL LOC. Although, we have a considerable commonality between the three original products source code, it is worth to consider it as different code, because it is repeated for each product and tightly coupled with it. This code repetition increases the effort of program reasoning and maintenance. A reduction due to the avoidance of code repetition could also be obtained using a different product line approach or some modularization techniques, like componentization. Another factor that contributes to the reduction in PL LOC is the existence of reusable aspects.

Table 5 shows the sizes of the PL aspects. The only reusable aspect is considerably smaller than the product-specific ones. The small size of this aspect is convenient for it to be reusable across different PL instances. With a more fine-grained configuration knowledge, we expect that there would be a higher number of reusable aspects and the relative size of the product-specific ones would decrease. Eventually, it could happen that, for some PL instances, no product-specific aspect would be necessary, in which case such instance would be derived solely by reusing core aspects.

Analyzing Table 5 in conjunction with the configuration knowledge presented previously, we can infer that the relative size of aspect code in the PL members ranges from 16% for P1 and P2 to 20% for P3.

Table 5. LOC of aspects in the PL

Aspect	LOC
AspectP1	421
AspectP2	426
AspectP3	661
AspectFlip	72

Another analyzed metric was the packaged application (jar files) sizes of the original and of PL implementations (Table 6). Jar files, that are released to final users, include not only the bytecode files, but also every resource necessary to execute the application, such as images and sound files In the case study products, additional resources represents, on average, 45% of the total jar file size. To measure the impact of our approach on bytecode size, we are considering, in Table 4, the jar files containing only the class files, excluding other resources. The jar file size is a very important factor in games for mobile devices, due to memory constraints.

Table 6. Jar size (kbytes) in original and PL implementations

	Original implementation		**PL implementation**	
	size	reduced size	size	reduced size
P_1	32.4	29.0	67.5	38.4
P_2	33.2	28.8	69.1	33.3
P_3	56.1	52.4	93.5	56.7
Total	98.1	86.6	206.6	104.8

We can notice a jar size increase from original versions to PL instances. The reason for this is the overhead generated by the AspectJ weaver on the bytecode files. We also noticed that very general pointcuts intercepting many join points can lead to greater increases in bytecode file sizes. This considerably influenced us in the definition and use of the refactorings. Moreover, we can gain a significant reduction in the jar size when using a bytecode optimization tool [27]. The reduced size of each original version and PL instance are shown in Table 6.

Although in this case study the PL implementation offers to the user of our approach the same functionality but with a higher application size, our approach is useful mostly because of the benefits that the PL approach brings to the development process: reuse and maintenance are improved, code replication is minimized, and derivation of new products is faster and less costly. Further, the increase on bytecode size can be minimized by further advances in optimization

tools. Our initial results show that, in cases where pointcuts matches few join points, by inlining the body of the advice in the base code, we can already reduce bycode size.

5 Tool Support

We have designed and implemented a prototype of a tool for supporting variability management in the PL context. Currently, the tool aims at extracting variations from existing products, by isolating such variations into aspects, which in turn customize the incrementally emerging PL core. The tool currently implements a subset of the refactorings discussed in Sect. 2. The subset comprises all but the *Extract Aspect Commonality* and the *Extract Argument Function* refactorings.

For example, in order to perform the *Extract Before Block Refactoring*, the user first selects a piece of code to extract (in this refactoring such piece must be at the beginning of a method) and then clicks on the button to perform the refactoring. Next, a dialog box pops up, where the user specifies to which aspect the variability is to be moved. It can be moved to either an existing aspect or to

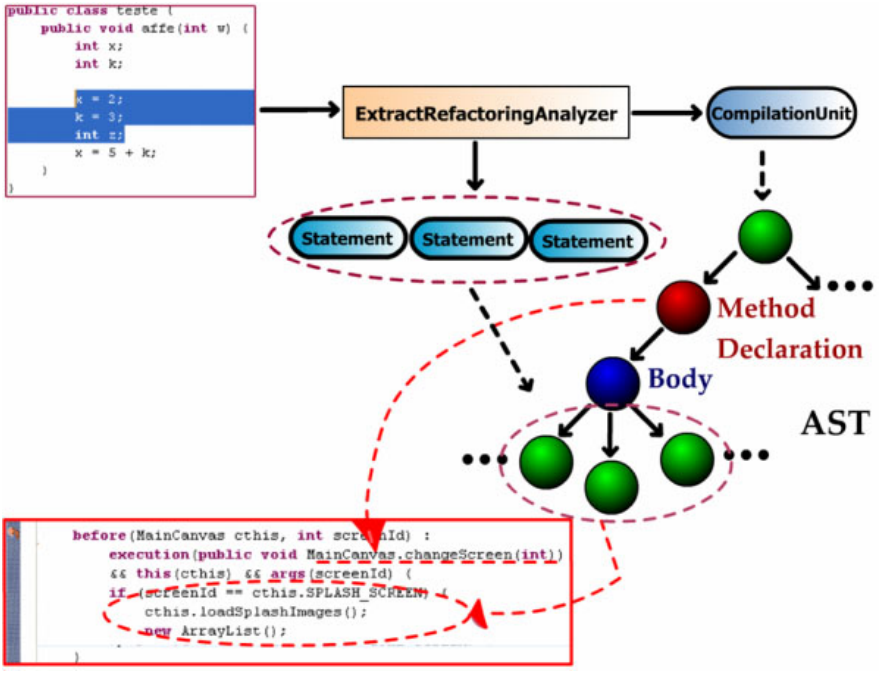


Fig. 6. Basic workings of the tool. User selection is parsed and matched in the program AST. Then the refactorings transformations are applied to classes and aspects, and the output generated.

a new one available from the combo box. After confirming the dialog, the tool first checks the refactoring preconditions and, only if these are met, applies the refactoring transformation, resulting in a new version of the emerging core and in a new version of the aspect isolating the selected variability. For performing the other refactorings, the user interacts likewise with the tool.

The prototype has been implemented as an Eclipse plug-in [26]. It defines the *Product Line Perspective*, comprising some buttons in the tool bar corresponding to the refactorings.

Figure 6 illustrates the basic workings of the tool. At first, the user selection from the editor is captured as text via the Workbench API which is used by the *Extract Refactoring Analyzer* to look up for the equivalent statements in Abstract Syntax Tree (AST). Then, the refactoring preconditions are checked in the AST. In the sequence, the AST pieces necessary to perform the refactoring transformation are determined. Finally, such the transformation is performed in the original AST and also in the aspect. In the latter, the code is generated as text buffer without manipulating AspectJ's AST API, since such API itself is not currently available. In contrast, the AST API Java is available at JDT's `jdt.core.dom package` [26].

6 Related Work

Our research is in the convergence of a number of areas involving PLs, AOP, refactoring, programming laws, and model refactoring. In the next sections, we compare our work to research in recurring combinations of these areas.

6.1 AOP, PLs, and Refactoring

As in our previous research [4], the current work addresses PL refactoring in the extractive and reactive contexts. However, the current work has three major improvements over our previous research.

Formalization. Section 3 of the current work shows how refactorings (coarse-grained program transformations) in the product line context can be understood in terms of programming laws (finer-grained program transformations). The programming laws themselves (not the refactorings) have been defined by the third author in previously [11]. But the novelty in the current work is to decompose the product line refactorings in terms of such laws. Figure 4 shows how Refactoring 1 (Sect. 2.3) is derived from consecutive applications of these laws, and the second column of Table 2 summarizes how each product line refactoring in the first column of the same table is decomposed into a sequence of application of such laws. Indeed, only the first column of the Table 2 is in previous work, but not the second column (the derivation), a conceptually substantial difference. In this way, the new work presented in Section 3 lays a more formal foundation for the work presented previously [4]. By expressing refactorings in terms of

programming laws, we can increase the confidence that the refactorings are indeed correct. This is feasible because the laws are simpler and easier to reason about than the refactorings themselves [12]. Further, this is relevant because it reduces the burden on testing, which is extremely expensive in the product line scenario.

Extension/Refinement of Method: Refactoring preconditions are a subtle issue in object-orientation [7,8] and in aspect-orientation [12]. Accordingly, the preconditions of some refactorings we present have improved slightly since our previous work, and the current work has been updated accordingly. This is also a consequence of having a more formal foundation in the current work.

Tool Support: The scientific contribution of the prototype itself is rather limited, but it helps to show that our approach can benefit from effective tool support, which is essential in the product line scenarios we focus on (extractive and reactive). Additionally, this is a step towards showing the viability of AOSD in industry.

Prior research by other researches also evaluated the use of AOP for building J2ME product lines [5]. We complement this work by considering the implementation of more features in an real application, explicitly specifying the refactorings to build and evolve the PL, and raising issues in AspectJ that need to be addressed in order to foster widespread application in this domain. Additionally, we rely on concern graphs [24] to identify variant features. Concern graphs provide a more concise and abstract description of concerns than source code. Once the concern is identified, we extract it into an aspect and may further revisit it during PL evolution.

AOP refactorings have also been described elsewhere [16,23]. The former proposes a catalog for object-to-aspect and aspect-to-aspect refactorings, whereas the latter provides an abstract representation of object-to-aspect refactorings as roles. However, their use in the PL setting is not explored, and the refactorings format follows the imperative style [14]; in contrast, our approach is template-oriented, abstract, concise, and thus does not bind a specific implementation, which could be done, for instance, with a transformation systems receiving as input refactoring templates.

In another approach, a language-independent way to represent variability is provided, and it is shown how it can be used to build J2ME game PLs product lines [29]. Our approach differs from such work because, although ours relies on language-specific constructs, it has the advantage of not having to specify join points in the base. Moreover, their approach, despite language-independent, considerably complicates understanding the source code due to the tags introduced to represent variability.

A recent work [28] reports the AOP refactoring of a middleware system to modularize features such as client-side invocation, portable interceptors, and dynamic types. Nevertheless, such work does not describe the refactorings abstractly and does not attempt to express them in terms of simpler programming laws as a way to guarantee behave preservation, as we do.

6.2 Programming Laws, Model Refactoring, and Optimization

Previous work [11] presented 30 aspect-oriented programming laws and showed how these could derive some aspect-oriented refactorings. In our work, we have explored the usefulness of such approach in validating extractive and evolutive refactorings for building product lines in the mobile game domain. Additionally, this task prompted not only an extension of the number of laws initially proposed, but also a more careful description of some subtle issues of these laws, such as handling AspectJ's precedence semantics, which were skipped in the original work. Finally, the experience in using the laws during derivation suggested that these be organized in a more concise notation, which could lead to the implementation of a generative library.

The process of defining programming laws and showing how these can be used to derive refactorings has also been addressed for for object-oriented languages [7]. Such research additionally formally proves not only the completeness of such set of laws, but also the correctness of each law, by relying on a weakest precondition semantics [9]. Our work, despite not formally proving the laws, still benefits from understanding coarse-grained transformations in terms of simpler ones.

If high-level algebraic specification of products are available, as described in [21], an efficient optimization algorithm could be applied in order to extract the product line core from these specifications with the Shared Class Extractor operator. However, the hypothesis of having this high-level specification may not be met in practice, in such a way that the domain engineer would need to address handling legacy software directly at the design or at the implementation level. Our approach addresses building a product line from existing design/implementation artifacts. Additionally, the variations handled by our approach are considerably crosscutting and may have fine-granularity, which is not the focus of the work mentioned.

6.3 Aspect Composition

Section 3.2 shows that the laws compose in the sense that their consecutive application is equivalent to a coarse-grained transformation. Note, however, that the application of each law assumes a fixed context. During composition, this context can change from the application of one law to another, but this shows the laws assume a closed-world approach. Therefore, they are not appropriate for working exclusively with libraries, for example, but are suitable for PLs, since the assets are available in the PL scope and can be considered as the context. Moreover, these compositions of laws have shown to be useful for the case study in this work and in case studies in four other domains [11]. However, more general and functional composition among aspects has shown to be limited within AspectJ [22]. Nevertheless, we believe this is not a strong disadvantage because handling variability in PL does not necessarily need to be addressed by using aspect composition. In fact, feature interaction and interaction between the core and the extension—common PL phenomena—may prevent functional

composition from being applied. Current work on XPI [25] and EJP [19] suggest that such interactions are not rare.

7 Conclusion

We present a method and a tool for creating and evolving product lines combining the reactive and extractive approaches. Our method uses a set of refactorings, which can be extended when necessary. We show that these refactorings can be derived from a combination of programming laws, allowing us to better understand these refactorings and increase the confidence that they are correct. This is specially relevant because it reduces the burden on testing, which is extremely expensive in the PL scenario. Our refactorings rely on AOP to modularize crosscutting concerns and to generalize the implementations of these concerns in order to increase code reuse.

Our evaluation with an existing mobile game suggests that we can benefit from extensive code reuse and easily evolve the PL to encompass other products while still maintaining code correctness, since the refactorings are derived from sound elementary programming laws. It also provides some examples of strategies on the applications of refactorings that manage to handle the implementation of variant features.

Although our case study has addressed only a fraction of the configurations from a concrete PL, such fraction exposes most variability issues in the domain. Further, although the evaluation is in the mobile game domain, we argue that the method and the issues addressed here are valid for mobile applications in general, of which mobile games are representative. We also believe that other variant domains could benefit from our method.

Acknowledgments

We would like to thank the anonymous referees for useful suggestions. This research was partially sponsored by CNPq (grants 481575/2004-9, 141247/2003-7), MCT/FINEP/CT-INFO (grant 01/2005 0105089400), and FACEPE.

References

1. Alves, V.: Identifying variations in mobile devices. Journal of Object Technology 4(3), 47–52 (2005)
2. Alves, V., et al.: Comparative analysis of porting strategies in j2me games. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, pp. 123–132. IEEE Computer Society, Los Alamitos (2005)
3. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: GPCE 2006. Proceedings of the 5th International Conference on Generative Programming and Component Engineering, Portland, Oregon, USA, pp. 201–210. ACM Press, New York (2006), http://doi.acm.org/10.1145/1173706.1173737 isbn: 1-59593-237-2

4. Alves, V., Matos Jr., P., Cole, L., Borba, P., Ramalho, G.: Extracting and evolving mobile games product lines. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 70–81. Springer, Heidelberg (2005)
5. Anastasopoulos, M., Muthig, D.: An evaluation of aspect-oriented programming as a product line implementation technology. In: Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, Springer, Heidelberg (2004)
6. Avgustinov, P., et al.: abc: an extensible aspectj compiler. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, pp. 87–98 (2005)
7. Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic reasoning for object-oriented programming. Science of Computer Programming 52, 53–100 (2004)
8. Borba, P., Sampaio, A., Cornélio, M.: A refinement algebra for object-oriented programming. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 457–482. Springer, Heidelberg (2003)
9. Cavalcanti, A., Naumann, D.: A weakest precondition semantics for refinement of object-oriented programs. IEEE Transactions on Software Engineering 26(8), 713–728 (2000)
10. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2002)
11. Cole, L., Borba, P.: Deriving refactorings for AspectJ. In: AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented software development, pp. 123–134. ACM Press, New York (2005)
12. Cole, L., Borba, P., Mota, A.: Proving aspect-oriented programming laws. In: Foundations of Aspect-Oriented Languages Workshop at the 4th International Conference on Aspect-oriented software development, pp. 1–9. Iowa State University Technical Report (2005)
13. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
14. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
15. Oberschulte, C., Hanenberg, S., Unland, R.: Refactoring of aspect-oriented software. In: Net.ObjectDays, Erfurt, Germany (2003)
16. Hannemann, J., Murphy, G.C., Kiczales, G.: Role-based refactoring of crosscutting concerns. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, New York, NY, USA, pp. 135–146. ACM Press, New York (2005)
17. Hoare, C.A.R., Hayes, I.J., Jifeng, H., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. Commun. ACM 30(8), 672–686 (1987)
18. Krueger, C.: Easing the transition to software mass customization. In: Proceedings of the 4th International Workshop on Software Product-Family Engineering, Bilbao, Spain, October 2001, pp. 282–293 (2001)
19. Kulesza, U., Alves, V., Garcia, A., de Lucena, C.J.P., Borba, P.: Improving extensibility of object-oriented frameworks with aspect-oriented programming. In: Morisio, M. (ed.) ICSR 2006. LNCS, vol. 4039, pp. 231–245. Springer, Heidelberg (2006)
20. Liu, J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: International Conference on Software Engineering 2006 (ICSE'06), Shanghai, China (2006)

21. Liu, J., Batory, D.S.: Automatic remodularization and optimized synthesis of product-families. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 379–395. Springer, Heidelberg (2004)
22. Lopez-Herrejon, R., Batory, D., Lengauer, C.: A disciplined approach to aspect composition. In: PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, New York, NY, USA, pp. 68–77. ACM Press, New York (2006)
23. Monteiro, M.P., Fernandes, J.M.: Towards a catalog of aspect-oriented refactorings. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, New York, NY, USA, pp. 111–122. ACM Press, New York (2005)
24. Robillard, M.P., Murphy, G.C.: Concern graphs: Finding and describing concerns using structural program dependencies. In: Proceedings of the 24th International Conference on Software Engineering, May 2002, pp. 406–416 (2002)
25. Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, pp. 166–175. ACM Press, New York (2005)
26. World Wide Web, Eclipse Projetct (2004), `http://www.eclipse.org`
27. World Wide Web, ProGuard (2005), `http://proguard.sourceforge.net/`
28. Zhang, C., Jacobsen, H.-A.: Resolving feature convolution in middleware systems. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, pp. 188–205. ACM Press, New York (2004)
29. Zhang, W., Jarzabek, S.: Reuse without compromising performance: Industrial experience from rpg software product line for mobile devices. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 57–69. Springer, Heidelberg (2005)

A Survey of Automated Code-Level Aspect Mining Techniques

Andy Kellens[1,⋆], Kim Mens[2], and Paolo Tonella[3]

[1] Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, Belgium
akellens@vub.ac.be
[2] Département d'Ingénierie Informatique
Université catholique de Louvain
Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be
[3] ITC-irst, Centro per la Ricerca Scientifica e Tecnologica
Via Sommarive 18, 38050 Trento, Italy
tonella@itc.it

Abstract. This paper offers a first, in-breadth survey and comparison of current aspect mining tools and techniques. It focuses mainly on automated techniques that mine a program's static or dynamic structure for candidate aspects. We present an initial comparative framework for distinguishing aspect mining techniques, and assess known techniques against this framework. The results of this assessment may serve as a roadmap to potential users of aspect mining techniques, to help them in selecting an appropriate technique. It also helps aspect mining researchers to identify remaining open research questions, possible avenues for future research, and interesting combinations of existing techniques.

1 Introduction

Aspect-oriented software development (AOSD) tries to solve the problem of separating the core functionality of a software system from concerns that have a more system-wide behaviour and that cut across the primary decomposition of the software system. This problem is sometimes referred to as the "tyranny of the dominant decomposition" [1]. To overcome this prevalent decomposition [2], the AOSD paradigm provides new language constructs, like advices and pointcuts [3], which allow cross-cutting concerns to be written down in a new kind of module named *aspect*.

Almost ten years after its initial conception, this technology has left the research lab and is starting to be adopted by industry, which poses new interesting research problems. Just like the industrial adoption of the object-oriented paradigm in the early nineties led to a need for migrating legacy software systems to an object-oriented solution — triggering a boost of research on software reverse

⋆ Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

A. Rashid and M. Aksit (Eds.): Transactions on AOSD IV, LNCS 4640, pp. 143–162, 2007.

engineering, reengineering, restructuring and refactoring — the same is currently happening to the aspect-oriented paradigm.

The reasons for wanting to migrate a legacy system to an aspect-oriented solution are multiple. Due to the presence of crosscutting concerns, legacy systems tend to contain many symptoms of duplicated code, *scattering* of concerns throughout the entire system and *tangling* of concern-specific code with that of other concerns. Using aspect-oriented technology, these cross-cutting concerns can be cleanly separated from the base code, which becomes oblivious of them. This is supposed to make the system easier to understand, maintain and evolve.

However, manually applying aspect-oriented techniques to a legacy system is a difficult and error-prone process. Due to the large size of such systems, the complexity of the implementation, the lack of documentation and knowledge about the system, there is a need for tools and techniques that can help software engineers in locating or documenting the cross-cutting concerns in legacy systems or for more automated tools to discover such concerns, as well as for tools and methodologies to refactor the discovered cross-cutting concerns into aspects.

The study and development of such approaches is the objective of the emerging research domains of aspect mining and aspect refactoring. Whereas *aspect mining* is the activity of discovering cross-cutting concerns that potentially could be turned into aspects, *refactoring to aspects* is the activity of actually transforming these potential aspects into real aspects in the software system. (See Fig. 1.)

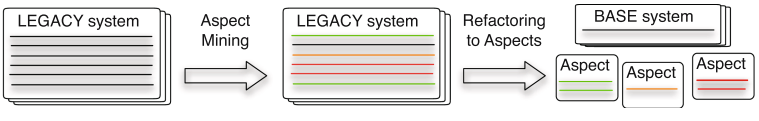


Fig. 1. Migrating a legacy system to an aspect-oriented system

This paper focusses on the activity of aspect mining and conducts a survey of existing code-level techniques, tools and methodologies that have been designed to aid a software engineer in discovering aspect candidates in a legacy system. A multitude of such techniques have recently been proposed, making it hard for potential users to decide which technique is most appropriate for their needs. This survey may serve as a roadmap to them by providing a taxonomy and comparison of currently existing aspect mining techniques, as well as some of their limitations and underlying assumptions.

The survey is also expected to be useful to the aspect mining research community. Although still in its infancy, this research area has recently seen a proliferation of approaches and techniques, inspired by several different research domains. Future research efforts will necessarily be devoted to comparing and combining the alternative approaches. This survey can be seen as a first step in that direction. It provides guidance in determining groups of similar techniques and highlighting their different underlying preconditions and properties.

To the authors' knowledge, this paper is the first published survey of existing aspect mining techniques. Its main contributions to the state of the art are:

- the definition of a set of criteria of comparison;
- the derivation of a taxonomy for the classification of techniques;
- the discussion of the properties of the existing techniques, according to the classification framework;
- the identification of future research directions in the area.

The paper is organized as follows: Sect. 2 defines aspect mining and positions it with respect to other aspect discovery approaches. Section 3 gives an overview of existing aspect mining techniques. The classification criteria are introduced in Sect. 4 and applied to the surveyed techniques in Sect. 5. The outcome of the classification is discussed in Sect. 6. Before concluding the paper in Sect. 8, a description of related research areas is given in Sect. 7.

2 Aspect Discovery

As mentioned above, the process of migrating a legacy system into a system using aspects consists of two steps: the discovery of aspect candidates and the refactoring of (some of) these candidates into aspects. In this survey we investigate techniques and tools that aid a developer in discovering possible aspects. This is not a trivial task, due to the size and complexity of current-day software systems and the lack of explicit documentation on the cross-cutting concerns present in those systems.

Three major kinds of aspect discovery approaches can be distinguished :

Early aspect discovery techniques. Traditionally, AOSD has focussed mostly on the software life-cycle's implementation phase. Research on 'early aspects' tries to discover aspects in earlier phases of the software life-cycle [4], such as requirements and domain analysis [5,6,7] or architecture design [8]. Identifying and managing early aspects not only helps to improve modularity in requirements and architectural design, but many early aspects eventually find their way into the code as implementation aspects.

In the context of legacy systems, where requirements and architecture documents are often outdated, obsolete or no longer available, early aspect discovery techniques cannot be applied and approaches that focus on source code are thus potentially more promising.

Dedicated browsers. A second class of approaches are the advanced special-purpose code browsers that aid a developer in manually navigating the source code of a system to explore cross-cutting concerns. Although the primary goal of these approaches is not to explicitly mine for aspects, but rather to document and localise cross-cutting concerns in order to maintain and evolve a system, these dedicated browsers can be used to identify aspects in a system as well.

Typically, a user of such a browsing approach starts out with a 'seed' of a concern, a starting point in the code, and uses the browser to further explore this concern. To do so the browser may propose other hotspots in the code which are related to the concern or provide the user with a query language to manually traverse the concern. Examples of such approaches are Concern Graphs [9], Intensional Views [10], Aspect Browser [11], (Extended) Aspect Mining Tool [2,12] and Prism [13].

Aspect mining techniques. *automate the process of aspect discovery* and propose their user one or more aspect candidates. To this end, they reason about the source code of the system or about data that is acquired by executing or manipulating the code. All techniques seem to have at least in common that they search for symptoms of cross-cutting concerns, using either techniques from data mining and data analysis like formal concept analysis and cluster analysis, or more classic code analysis techniques like program slicing, software metrics and heuristics, clone detection and pattern matching techniques, dynamic analysis, and so on.

In this survey we focus only on this third category of automated code-level tools and techniques that assist a developer in the activity of mining for cross-cutting concerns in an existing system. We define *aspect mining* as follows:

> *Aspect mining* is the activity of discovering those cross-cutting concerns that potentially could be turned into aspects, from the source code and/or run-time behaviour of a software system. We refer to such concerns as 'aspect candidates'.

3 Overview of Aspect Mining Techniques

This section offers a detailed overview of the different automated code-level aspect mining approaches that have been proposed over the last few years.

3.1 Analysing Recurring Patterns of Execution Traces

Breu and Krinke propose an aspect mining technique named *DynAMiT* (Dynamic Aspect Mining Tool) [14], which analyses program traces reflecting the run-time behaviour of a system in search of recurring execution patterns. To do so, they introduce the notion of *execution relations* between method invocations. Consider the following example of an event trace, where the capitals represent method names:

```
B() {
      C() {
            G() {}
            H() {}
          }
    }
A() {}
```

Breu and Krinke distinguish between four different execution relations: outside-before (e.g., B is called before A), outside-after (e.g. A is called after B), inside-first (e.g., G is the first call in C) and inside-last (e.g., H is the last call in C). Using these execution relations, their mining algorithm discovers aspect candidates based on recurring patterns of method invocations. If an execution relation occurs more than once, and recurs uniformly (for instance, every invocation of method B is followed by an invocation of method A), it is considered to be an aspect candidate. To ensure that the aspect candidates are sufficiently cross-cutting, there is an extra requirement that the recurring relations should appear in different 'calling contexts'. Although this approach is inherently dynamic, the authors have repeated the experiment using control-flow-graphs [15] to calculate the call relations statically. Breu also reports on a hybrid approach [16] where the dynamic information is complemented with static type information in order to remove ambiguities and improve on the results of the technique.

3.2 Formal Concept Analysis

Formal concept analysis (FCA) [17] is a branch of lattice theory which, given a set of objects and attributes describing those objects, creates *concepts*, i.e., maximal groups of objects that have common attributes. These concepts are organised into a lattice, according to the partial order associated with attribute (or equivalently object) set inclusion.

Formal Concept Analysis of Execution Traces. Tonella and Ceccato [18] developed *Dynamo*, an aspect mining tool which applies FCA to execution traces in order to discover possible aspects. When analysing a system using Dynamo, an instrumented version of the system is executed on a number of use cases, manually derived from the software documentation and/or from a high level description of the main functionalities. The output of this execution is a number of execution traces. These traces are then analysed using FCA: the use cases are the objects of the FCA algorithm, while the methods which get invoked during the execution of a use case are the attributes. In the resulting lattice, all concepts are selected which contain traces from exactly one use-case. These are regarded as aspect candidates if the following (automatically verified) conditions hold:

- Scattering: the specific attributes (methods) of the concept belong to more than one class.
- Tangling: different methods from the same class are specific to more than one use-case specific concept.

Formal Concept Analysis of Identifiers. Tourwé and Mens [19] propose an alternative aspect mining technique which relies on FCA. Unlike the Dynamo tool discussed above, Tourwé and Mens's *DelfSTof* tool analyses the source code of a system (experiments have been conducted on Smalltalk code [19] and on Java code [20]). Their approach performs an identifier analysis using the FCA algorithm. The assumption behind this approach is that interesting concerns in

the source code are reflected by the use of naming conventions in the classes and methods of the system. As input to the FCA algorithm, the classes and methods in the system are used as objects. As attributes, the FCA algorithm uses substrings generated from the classes and methods' names. For instance, a class named `QuotedCodeConstant` is split into the strings 'Quoted', 'Code' and 'Constant'. Substrings with little meaning, like 'a', 'with', ... are discarded from the results. The resulting concepts consist out of maximal groups of classes and methods which share a maximal number of substrings. After having filtered out many unimportant concepts automatically, a significant number of concepts remain which need to be inspected manually. Apart from being able to detect a number of programming idioms, design patterns and certain refactoring opportunities [21], the same approach can be used for aspect mining purposes [19] by restricting the concepts to those that are crosscutting (i.e. the involved methods and classes belong to at least two different class hierarchies).

3.3 Natural Language Processing on Source Code

Similar to the previous approach, Shepherd et al. [22] propose a technique that is based on the assumption that cross-cutting concerns are often implemented by the rigorous use of naming and coding conventions. Their approach uses natural language processing (NLP) information as an indicator for possible aspect candidates. They report on an experiment in which they use an NLP technique called *lexical chaining* [23] in order to find groups of related source-code entities which represent a cross-cutting concern. Lexical chaining will output, given a collection of words as input, chains of words which are semantically strongly related. In order to create the chains, the algorithm requires a semantical distance measure between each combination of words. To this end, Shepherd et al. used the WordNet [24] database, in combination with information about the parts of speech of each word, to calculate the semantical path between two words. In order to mine for cross-cutting concerns, they apply the chaining algorithm to the comments, method names, field names and class names of the system they are analysing. A user of their approach needs to manually inspect the resulting chains in order to select likely aspect candidates.

3.4 Detecting Unique Methods

Gybels and Kellens [25,26] propose the use of heuristics to mine for cross-cutting concerns. They observe that, in pre-AOP days, cross-cutting concerns were often implemented in an idiomatic way. Certain of these idioms can be regarded as "symptoms" of aspect candidates. An example of such an idiom is the implementation of a cross-cutting concern by means of a single entity in the system which is called from numerous places in the code (for instance, a 'logging' entity which is called from throughout the code). To detect instances of this pattern, Gybels and Kellens propose the "Unique Methods" heuristic which is defined as:

> "A method without a return value which implements a message implemented by no other method."

After calculating all unique methods in a system, sorting them according to the number of times a method is called, and filtering out irrelevant methods (like for instance *accessor* and *mutator* methods), the user has to manually inspect the resulting methods in order to find suitable aspect candidates. Regardless of the simplicity of this approach, the authors demonstrated the applicability of their technique by detecting typical aspects like tracing, update notification and memory management in the context of a Smalltalk image.

3.5 Clustering of Related Methods

Hierarchical Clustering of Similar Method Names: Shepherd and Pollock [27] report on an experiment in which they used agglomerative hierarchical clustering [28] to group related methods. This technique starts by putting each method in a separate cluster and then recursively merges clusters for which the distance between the methods is smaller than a certain threshold. They implemented this technique as part of an aspect-oriented IDE named *AMAV* (Aspect Miner and Viewer), which allows for easy adaptation of the distance measure used by the algorithm. For an initial experiment they used a simple distance measure opposite proportional to the common substring length of the names of the methods. This mining algorithm is used in combination with the viewing tool of the IDE which not only lists all the clusters which were found, but also consists out of a *cross-cutting pane* which displays the methods related to a cluster as well as an *editor pane*, in which the class context of a particular method is displayed.

Clustering Based on Method Invocations: He and Bai [29] propose another aspect mining technique based on cluster analysis. They start from the assumption that if the same methods are called frequently from within different modules, this may be a good indication that a hidden cross-cutting concern is present. As input for the clustering algorithm, a set of methods is given along with a distance measure based on the static direct invocation relationship (SDIR) between the methods. This distance measure varies between 0 and 1 and represents the dissimilarity of the methods. Methods which are closely related (i.e. which get called frequently together) will have distance approximating 0, while the distance between methods which are never or seldom called together will be close to 1.

3.6 Fan-in Analysis

Marin et al. [30] noticed that many of the well-known cross-cutting concerns exhibit a high fan-in. They propose using a fan-in metric in order to discover cross-cutting concerns in source code. They define the fan-in of a method m as the number of distinct method bodies which can invoke m. Because of polymorphism, a call to a method m contributes to the fan-in of all methods refining m,

as well as method m itself. Their mining algorithm comprises out of the following steps:

- Calculating the fan-in metric for all methods in the system.
- Filtering the results: next to filtering *accessor* and *mutator* methods, as well as utility methods like for instance `toString()`, the number of considered methods is also limited by only considering the methods with a fan-in value higher than a certain threshold.
- Manually analysing the remaining methods.

The authors present an experiment in which cross-cutting concerns were mined with a high precision: one third of all methods with high fan-in were seeds leading to an aspect. Moreover, 60% of the remaining two thirds were removed automatically.

3.7 Clone Detection

The symptom of 'code duplication' may be a good indicator of cross-cutting concerns in the source code of a system: because the cross-cutting concerns could not be cleanly modularised, certain parts of the implementation show high levels of duplicated code. Two techniques rely on this observation to mine for aspect candidates.

Detecting Aspects Using PDG-Based Clone Detection: A first technique, presented by Shepherd et al. [31] and implemented as a tool they call *Ophir*, makes use of *program dependence graphs* (PDG) to detect possible aspects. In a PDG, each statement in the code is represented by a node; the edges of the graph consist of control or data dependence relations between the statements. By comparing PDGs [32,33], this technique is able to recognise code duplication in the beginning of a method (i.e. aspect candidates for a 'before' advice). After filtering and coalescing the resulting PDGs, a number of possible aspect candidates remain.

Using AST- and Token-Based Clone Detection. Bruntink et al. also make use of clone detection techniques to mine for aspects. In [34,35], they compare *token-based* [36] clone detection, which is based on a lexical analysis of the source code, with *AST-based* [37] clone detection, which takes the parse tree of the source code into account. Both techniques output a number of *clone classes*, i.e. groups of code fragments which are considered to be clones of each other. They applied the clone detection techniques to a large C program in which the different cross-cutting concerns were annotated by a developer. In order to measure the effectiveness of the techniques, Bruntink et al. [38] empirically compare the resulting clone classes with the manual documentation of the cross-cutting concerns. Bruntink reports on a refinement of this work, in which a number of metrics for the clone classes are described which can be used to filter the results of the clone detection techniques.

4 Criteria of Comparison

We now present a set of criteria that will allow us, in Sect. 5, to compare the aspect mining techniques listed in Sect. 3. We compiled this set by focussing on the variabilities of the different techniques. As such we intended to obtain a taxonomy that supports potential users of aspect mining techniques to select an adequate technique, and that helps aspect mining researchers to understand the differences between their own and existing techniques. The taxonomy contrasts the different restrictions each technique imposes on the input data, the kinds of analysis which are used, the degree of automation and the scalability of each technique. These criteria form an initial comparative framework that may still evolve, following new developments in the field of aspect mining.

Static versus dynamic data. *What kind of data does the technique analyse?* Does it analyse input data which can be obtained by statically analysing the code, or dynamic information which is obtained by executing the program, or both?

Token-based versus structural/behavioural analysis. *Which kind of analysis does the technique perform?* We distinguish between:

- **Token-based.** Lightweight lexical analysis of the program: sequences of characters, regular expressions, etc.
- **Structural/Behavioural.** Structural and behavioural analysis of the program: parse trees, type information, message sends, etc.

Granularity. *What is the level of granularity of the mined aspect candidates?* While some techniques discover aspects at the level of methods, others work more fine-grained by considering individual statements or code fragments as part of the aspect candidates.

Tangling and Scattering. *What symptoms of aspects does the aspect mining technique look for?* Does it explicitly look for symptoms of scattering, tangling, or both? Cross-cutting concerns are characterised by high tangling and scattering.

User involvement. *What kind of user involvement is required in order to mine for aspects?* What effort does the technique require from its user? Does the user have to manually browse through all results of the technique in order to indicate viable aspect candidates? Is there additional input required from the user during the mining process?

Largest system. *On what size of system has the technique been applied?* Even though a technique might behave well and exhibit interesting properties when applied to small examples, this does not imply that the same holds when the technique is tried on larger software systems. Problems may arise from the computational complexity, the need for user involvement, degradation of accuracy with size, etc. Thus, validation on large software systems may be used as an indicator of scalability.

Empirical validation. *To what degree have existing techniques been validated quantitatively on real-life cases?* For the validations done, has it been reported how many of the known aspects were found and how many of those reported were false positives?

Preconditions. *What (explicit or implicit) conditions must be satisfied by the concerns in the program under investigation in order for a particular mining technique to find suitable aspect candidates?*

5 Assessment

Based on the criteria introduced in Sect. 4, we compare the different aspect mining techniques summarised in Sect. 3. To gain space, we abbreviate the names of the techniques used, as shown in Table 1.

Table 1. List of techniques that were compared

Abbreviated name	Short description of the technique	Sections
Execution patterns	Analysing recurring patterns of execution traces	3.1
Dynamic analysis	Formal concept analysis of execution traces	3.2
Identifier analysis	Formal concept analysis of identifiers	3.2
Language clues	Natural language processing on source code	3.3
Unique methods	Detecting unique methods	3.4
Method clustering	Hierarchical clustering of similar method names	3.5
Call clustering	Clustering based on method invocations	3.5
Fan-in analysis	Fan-in analysis	3.6
Clone Detection (PDG-based)	Detecting aspects using PDG-based clone detection	3.7
Clone Detection (AST/token-based)	Detecting aspects using AST-based and token-based clone detection	3.7

For each of the studied techniques, Table 2 shows the kind of data (static or dynamic) analysed by that technique, as well as the kind of analysis performed (token-based or structural/behavioural).

Table 2. Kind of input data and kind of analysis of each technique

	Kind of input data		**Kind of analysis**	
	static	dynamic	token-based	structural/behavioural
Execution patterns	X	X	–	X
Dynamic analysis	–	X	–	X
Identifier analysis	X	–	X	–
Language clues	X	–	X	–
Unique methods	X	–	–	X
Method clustering	X	–	X	–
Call clustering	X	–	–	X
Fan-in analysis	X	–	–	X
Clone detection (PDG)	X	–	–	X
Clone detection (token)	X	–	X	–
Clone detection (AST)	X	–	–	X

Most techniques work on statically available data. 'Dynamic analysis' reasons about execution traces and thus requires executability of the code under analysis. Only 'Execution patterns' works with *both* kinds of input, since both a static version which uses control-flow graphs, and a dynamic version which uses execution traces, exist. As for the kind of reasoning, four techniques perform a

token-based analysis of the input data. 'Identifier Analysis' and 'Method Clustering' reason about the names of the methods in a system only. The 'Language Clues' approach is token-based because it reasons about individual words which appear in the program's source code. The four token-based techniques all rely on the assumption that cross-cutting concerns are often implemented by the rigorous use of naming conventions. The seven other techniques reason about the input at a structural or behavioural level.

Table 3. Granularity of and symptoms looked for by each technique

	Granularity		**Symptoms**	
	method	code fragment	scattering	tangling
Execution patterns	X	–	X	–
Dynamic analysis	X	–	X	X
Identifier analysis	X	–	X	–
Language clues	X	–	X	–
Unique methods	X	–	X	–
Method clustering	X	–	X	–
Call clustering	X	–	X	–
Fan-in analysis	X	–	X	–
Clone detection	–	X	X	–

Table 3 summarises the finest level of granularity (methods or code fragments) of the different techniques, and whether they look for symptoms of scattering and/or tangling. With a few exceptions, the typical granularity of the techniques surveyed is at method level. Therefore, most techniques output several sets of methods, each representing a potential aspect seed. Only the three 'Clone detection' techniques detect aspect code at the level of code fragments and can therefore provide more fine-grained feedback on the code that needs to be put into the advice of the refactored aspect. All techniques use scattering as the basic indicator of the presence of a cross-cutting concern. Only 'Dynamic analysis' takes *both* scattering and tangling into account, by requiring that the methods which occur in a single use-case scenario are implemented in multiple classes (scattering), but also that these methods occur in multiple use-cases and thus are tangled with other concerns of the system.

Table 4. An assessment of the validation of each technique

Technique	Largest case	Size case	Empirically validated
Execution patterns	Graffiti	3,100 methods/82KLOC	–
Dynamic analysis	JHotDraw	2,800 methods/18KLOC	–
Identifier analysis	JHotDraw	2,800 methods/18KLOC	–
Language clues	PetStore	10KLOC	–
Unique methods	Smalltalk image	3,400 classes/66000 methods	–
Method clustering	JHotDraw	2,800 methods/18KLOC	–
Call clustering	Banking example	12 methods	-
Fan-in analysis	JHotDraw TomCat 5.5 API	2,800 methods/18KLOC 172KLOC	– –
Clone detection (PDG)	TomCat	38KLOC	-
Clone detection (AST/token)	ASML C-Code	20KLOC	X

To provide more insights into the validation of the techniques, Table 4 mentions the largest case on which each technique has been validated, together with the size of that case, and whether the results have been evaluated quantitatively (for example, how many known aspects were actually reported, how many false positives and negatives were reported, and so on). While the size of the largest analysed system is significant for most of the studied techniques (only 'Call Clustering' was applied to a toy example only), empirical validation of the results was almost always neglected. It is also worth noting that 4 out of 9 techniques have been validated on the same case: JHotDraw.

One important criterion to help selecting an appropriate technique to mine a given system for aspects is what implicit or explicit assumptions that technique makes about how the crosscutting concerns are implemented. Table 5 summarises these assumptions in terms of preconditions that a system has to satisfy in order to find suitable aspect candidates with a given technique.

Table 5. What conditions does the implementation of the concerns have to satisfy in order for a technique to find viable aspect candidates?

Technique	Preconditions on crosscutting concerns in the analysed program
Execution patterns	Order of calls in context of crosscutting concern is always the same.
Dynamic analysis	At least one use case exists that exposes the crosscutting concern and another one that does not
Identifier analysis	Names of methods implementing the concern are alike
Language clues	Context of concern contains keywords which are synonyms for the crosscutting concern
Unique methods	Concern is implemented by exactly one method
Method clustering	Names of methods implementing the concern are alike
Call clustering	Concern is implemented by calls to same methods from different modules
Fan-in analysis	Concern is implemented in separate method which is called a high number of times, or many methods implementing the concern call the same method
Clone detection	Concern is implemented by reusing a certain code fragment

'Identifier Analysis', 'Method Clustering', 'Language Clues' and 'Token-based clone detection' all rely on the assumption that developers rigorously made use of naming conventions when implementing the cross-cutting concerns. 'Execution Patterns' and 'Call Clustering' assume that methods which often get called together from within different contexts are candidate aspects. The fan-in technique assumes that crosscutting concerns are implemented by methods which are called many times (large footprint), or by methods calling such methods.

Table 6 summarises the kind of involvement that is required from the user. None of the existing techniques works fully automatic. All techniques require that their users browse through the resulting aspect candidates in order to find suitable aspects. Some require that the users supply appropriate input, like for instance the 'Dynamic Analysis' technique which expects as input also a number of use-cases.

We combined all these tables into a single taxonomy, depicted in Fig. 2, that could serve as an initial roadmap to aspect miners and researchers. Each of the nine considered techniques is represented by a small rectangle. The four larger

Table 6. Which kind of user involvement do the different techniques require?

Technique	User Involvement
Execution patterns	Inspection of the resulting "recurring patterns"
Dynamic analysis	Selection of use cases and manual interpretation of results
Identifier analysis	Browsing of mined aspects using IDE integration
Language clues	Manual interpretation of resulting lexical chains
Unique methods	Inspection of the unique methods; eased by sorting on importance
Method clustering	Browsing of mined aspects using IDE integration
Call clustering	Manual inspection of resulting clusters
Fan-in analysis	Selection of candidates from list of methods, sorted on highest fan-in
Clone detection	Browsing and manual interpretation of the discovered clones

rectangles distinguish 'static' from 'dynamic' techniques, and 'method-level' techniques from techniques that report 'method fragments' as seeds. The two rounded rectangles on the left partition the considered techniques into 'token-based' techniques and "structural/behavioural' ones.

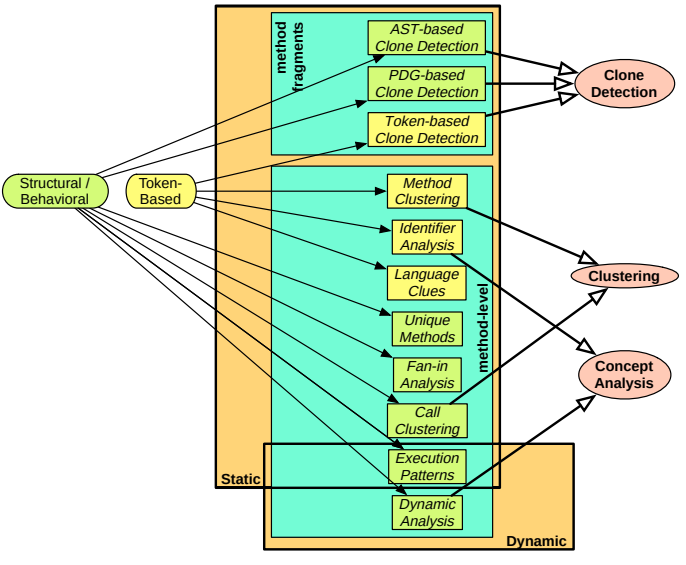


Fig. 2. An initial taxonomy of Automated Code-Level Aspect Mining Techniques

The largest rectangles, separating static from dynamic techniques, represent an extremely relevant criterion for practitioners and researchers, since it entails different requirements on the input (source code vs. executable system) and different interpretations of the output (conservative vs. partial results). This is discussed more thoroughly in the next section.

It is interesting to observe that all known dynamic techniques are structural/behavioral and work at method-level, whereas the static techniques can be divided into 'method-level' and 'method-fragments', as well as based on the kind of information used (i.e., 'token-based' vs. 'structural/behavioural'). This is relevant information for tool developers or practitioners, who might have knowledge

about the best aspect indicators to use (lexical oriented or structure/behaviour based) or who may have certain demands about the granularity of the results.

Each technique listed in Fig. 2 is uniquely classified by each of the three criteria described above (except 'Execution patterns', which relies on both static and dynamic analysis). The ellipses on the right show some additional taxonomic properties that group some (but not all) of the techniques. More specifically, they specify the basic algorithm underlying the technique: 'Concept analysis', 'Clustering' or 'Clone detection'.

6 Discussion

In this section we discuss some of the lessons we have learned from our comparison of automated code-level aspect mining techniques.

Static versus dynamic data. By relying on static information only, most techniques impose little requirements on the program under analysis. Often, the static information required by these techniques can be computed even for software systems that do not form a complete executable or, in some cases, for code that does not even compile or parse. Dynamic techniques impose heavier constraints by requiring both compilation and execution, as well as an appropriate execution environment. For a user of aspect mining techniques, this can have a significant impact on the choice of the used technique.

The only technique which applies both static and dynamic information is 'Execution patterns' [16]. While for this technique the static information is used to perform a more fine-grained analysis than with its purely dynamic variant, for the other techniques we discussed, a combination of static and dynamic analysis does not immediately seem to provide any obvious advantages. Future aspect mining techniques however may combine static information with dynamic analysis. A common problem with dynamic analysis is that, for large systems, the data provided by tracing the execution of the system might be huge. One possible way to limit this enormous amount of data could be to use static information to restrict the number of places in the code which will be instrumented, thus resulting in smaller traces.

Granularity. Most techniques retrieve aspect candidates as sets of methods that pertain to a crosscutting concern. This kind of granularity works reasonably well if the entire method implements the cross-cutting behaviour, and thus the places in the code where this method gets called can be considered as the joinpoints where the refactored aspect needs to intervene. However, for concerns like e.g. parameter checking, the cross-cutting behaviour is not localised into a single method-call, but is instead implemented by a 'pattern' in the code, which is spread throughout multiple statements. In such cases, the user has to provide additional effort in analysing the results of the aspect mining technique in order to highlight the code fragments that are part of the cross-cutting concerns, or use a technique which works at the granularity of method statements.

Tangling and Scattering. Both tangling and scattering have been presented as indicators of cross-cutting concerns. While all techniques take scattering into

account, and try to approximate it by for instance requiring calls to cross-cutting behaviour to originate from different class hierarchies, none of the techniques, with the exception of 'Dynamic analysis', look for symptoms of tangling. This can be explained by the fact that any heuristic for tangling needs high-level information about the different concerns in a system. Since 'Dynamic Analysis' makes use of use-case scenarios, it can take tangling into account by requiring that methods of different classes should be specific to a use-case scenario. As such information seems hard to approximate without external information, it seems unlikely that techniques which analyse source code only can easily take tangling into account. However, when information on artefacts from the earlier phases of the software engineering process are available, these may be exploited to provide a heuristic for tangling.

Empirical validation. In order to perform a quantitative comparison of the studied techniques, empirical validation is of fundamental importance. In most reported studies, however, it was skipped and replaced with a more qualitative result assessment. This is due to the intrinsic difficulty of such a validation: it is hard to define (a priori) the set of relevant aspects to be discovered and to decide (a posteriori) which reported aspects are wrong. Nevertheless, it is impossible for this discipline to make further progress without such an empirical validation. This requires the ability to measure the precision and recall of the results, both in terms of the reported aspects and in terms of the discovered aspect seeds (code entities assigned to the aspect candidates).

In addition to the precision and recall metrics, user studies should be conducted to empirically validate the results. End users of the aspect mining techniques (e.g., the programmers of a system on which the aspect mining is being applied) should be involved in order to evaluate the actual usefulness and usability of each proposed technique. In addition to studies in an industrial context, replication of such studies with students, in classroom settings, would be fruitful too. Such studies might provide important indications of the actual needs that emerge in the execution of a real task of migration toward AOSD. This might in turn steer the research on aspect mining (and refactoring) approaches.

Scalability. Although we have mentioned the largest system on which each technique was validated, it is impossible at this time to make hard claims regarding the scalability of the techniques we studied. While the size of the system can give some indication of whether a technique might scale, and while some techniques were applied to large industrial systems, we cannot make general claims based on the limited data provided by the authors of the different aspect mining techniques. In order to properly assess the scalability of a technique, one would not only have to take into account the time complexity of a technique with respect to the input size, but also the amount of user involvement required for applying that technique. Techniques which require vast amounts of effort to browse through may be less cost-effective. Measuring the amount of required user involvement is strongly related with the need for more empirical validation, as metrics like precision and recall are necessary to quantify this property.

Preconditions. All of the techniques make *different* assumptions about how cross-cutting concerns are implemented in the system, in order for the technique to find viable aspect candidates (see Table 5). Since the assumptions of the techniques we studied seem quite complementary, and each technique thus aims at discovering a different flavour of implementation of cross-cutting concerns, it is advisable for users of aspect mining techniques to apply multiple techniques to the same system. This way, aspects that are missed by one technique because they do not exhibit a particular symptom, may be detected by another technique. The assessment of the current aspect mining techniques we presented in this paper can serve as a roadmap for a developer to select which techniques might be applicable to mine for aspect candidates in a given system.

Common benchmark. There is a strong need to compare the quality of the different aspect mining techniques that have been proposed in literature. Only very few approaches provide a detailed analysis of their effectiveness. Most techniques are presented only as a proof-of-concept in which it is demonstrated that useful aspects are found. And even for those techniques that have presented more detailed results, they cannot necessarily be compared with others either because they were performed on a different case or because they were presented in a non-compatible format, possibly using different metrics. Therefore, in order to obtain better insights into the strengths and weaknesses of known aspect mining approaches, it is advisable to validate the different techniques on a common case-study and using a common set of well-defined metrics.

JHotDraw [39] seems to be a good candidate for becoming a common benchmark for aspect mining techniques. Ceccato et al. [20] already described an experiment in which they used this case to qualitatively compare three different aspect mining techniques: 'Dynamic analysis', 'Fan-in analysis' and 'Identifier analysis'. In addition, JHotDraw is currently being reworked to an aspect-oriented version, AJHotDraw [40], which is supposed to be behaviourally consistent with JHotDraw itself. The results of mining aspects on JHotDraw could be compared with those aspects actually present in AJHotDraw.

7 Related Research Areas

From our description of the different techniques in Section 3, it became clear that the field of aspect mining is strongly related to and inspired by a number of other research fields, of which we recall the most important ones in this section. A closer study of these related fields may provide useful ideas to advance the state of the art in aspect mining.

Data Mining. A number of the techniques we discussed make use of data mining algorithms like *cluster analysis* and *formal concept analysis*. This does not come as a surprise, as in the past *data mining techniques* have already been successfully applied on large-scale data sets to retrieve groups of elements which conceptually belong together. This research domain is quite extensive however, and may contain many other approaches which may be ideal candidates for being used as the basis of aspect mining techniques.

Software Comprehension. Aspect mining is closely related to techniques that aid developers in understanding a piece of software. While the goal of *software comprehension techniques* is more general than discovering cross-cutting concerns in legacy code, the results obtained in this field can lead to interesting insights concerning aspect mining.

Program Analysis. In a similar way, knowing that some *program analysis techniques* (for example, *clone detection*) have already been successfully applied to aspect mining, it might be worthwhile considering other program analysis techniques like *slicing* and *metrics* for the purpose of aspect mining.

Re(verse) Engineering. There has been quite some research on how to re(verse)-engineer an ill-structured software system to one that is better structured (for example, with a nice object-oriented design and well-defined architecture). Variants of some of these re(verse) engineering techniques may prove useful for aspect mining as well.

Concept and Feature Location. Existing approaches to locate high level concepts or features (i.e., user-triggered functional requirements) in the source code may be used for the location of cross-cutting functionalities as well, assuming these are known to the programmers. Thus, they have good potential of combination with the aspect mining techniques surveyed in this paper, as was the case for Eisenbarth et al.'s feature location method [41] and Dynamo [18].

To the authors' knowledge, the present work is the first attempt to provide a survey of currently existing aspect mining techniques. Its main contribution to the aspect mining literature is a tentative framework for the classification of the existing techniques into a coherent taxonomy. Research contributions in the area of aspect mining are being produced at an extremely high rate (further works have appeared since the date of submission of this paper), so we cannot be exhaustive. Rather, we aim at defining a framework that can be reused, possibly with adaptations, whenever a new technique needs to be compared with the existing ones.

8 Conclusion

In this paper we presented a survey of existing automated code-level aspect mining techniques. To compare these techniques we proposed a comparative framework and taxonomy which allowed us to discriminate among the different techniques. From this comparison we learned important lessons that can be used by practitioners when selecting an aspect mining technique and that can serve as input to improve the state of the art in this research area.

Aspect mining users are likely to have some knowledge about the system under analysis. They might know whether a set of well-defined use-cases is available and can be employed to isolate relevant concerns or if it is better to rely on the source code alone. This discriminates dynamic versus static analyses. Moreover, they might have some information about the presence of adhered naming conventions throughout the system, which enables token-oriented techniques.

Additional knowledge, such as the occurrence of code duplication, might be relevant to decide on the granularity of the technique to use and to select the basic algorithm exploited by the mining technique.

The main contribution of this work to the research area of aspect mining is the definition of a (preliminary) classification framework, that can be used to position each new technique proposed in the area, and to compare it with the existing ones along relevant dimensions. Other contributions came out of the discussion of the taxonomy. The main future directions that emerged from this study are the need for empirical, comparative evaluations and the opportunity for developing combined techniques. Indeed, since every technique relies on different assumptions and uses different underlying analysis techniques, the studied techniques are highly complementary, which suggests the possibility of several useful combinations. More specifically, it could be worthwhile to:

- combine techniques that rely on static and dynamic analysis;
- combine token-oriented and structural/behavioural techniques;
- extend techniques that work at the granularity of methods with techniques that work at the level of code fragments;
- improve the results of techniques that only look for symptoms of scattering, by taking into account the phenomenon of tangling too;
- combine techniques that rely on different underlying assumptions, thus enabling the discovery of different kinds of aspects.

A natural follow-up to this study would be a more in-depth comparison of the results obtained by the different techniques, based on empirical validation. To enable a quantitative assessment, we need a common set of benchmark programs against which the techniques can be compared, as well as a common set of metrics for measuring the precision and recall of the produced results. Involvement of end-users of the tools to assess the quality of the produced results is also important, considering some of the mined aspect candidates could be refactored eventually into implementation aspects.

Acknowledgments

The authors are grateful to Mariano Ceccato, Marius Marin and Tom Tourwé, to our anonymous reviewers and to other members of the aspect mining community, for the valuable comments they provided on earlier versions of this paper.

References

1. Tarr, P., Ossher, H., Harrison, W., Stanley, M., Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: International Conference on Software Engineering (ICSE), pp. 107–119. IEEE Computer Society Press, Los Alamitos (1999)
2. Hannemann, J., Kiczales, G.: Overcoming the prevalent decomposition in legacy code. In: Workshop on Advanced Separation of Concerns, International Conference on Software Engineering (ICSE) (2001)

3. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications (2003)
4. Baniassad, E., Clements, P.C., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. IEEE Software 23(1), 61–70 (2006)
5. Baniassad, E., Clarke, S.: Theme: An approach for aspect-oriented analysis and design. In: International Conference on Software Engineering (ICSE), pp. 158–167. IEEE Computer Society Press, Washington, USA (2004)
6. Rashid, A., Sawyer, P., Moreira, A.M.D., Araújo, J.: Early aspects: A model for aspect-oriented requirements engineering. In: Joint International Conference on Requirements Engineering (RE), pp. 199–202. IEEE Computer Society Press, Los Alamitos (2002)
7. Tekinerdogan, B., Aksit, M.: Deriving design aspects from canonical models. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 410–413. Springer, Heidelberg (1998)
8. Bass, L., Klein, M., Northrop, L.: Identifying aspects using architectural reasoning. In: Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design. Workshop of the 3rd International Conference on Aspect-Oriented Software Development (AOSD) (2004)
9. Robillard, M.P., Murphy, G.C.: Concern graphs: Finding and describing concerns using structural program dependencies. In: International Conference on Software Engineering (ICSE 2002), pp. 406–416. ACM Press, New York (2002)
10. Mens, K., Poll, B., González, S.: Using intentional source-code views to aid software maintenance. In: International Conference on Software Maintenance (ICSM'03), pp. 169–178. IEEE Computer Society Press, Los Alamitos (2003)
11. Griswold, W., Kato, Y., Yuan, J.: Aspect browser: Tool support for managing dispersed aspects. In: Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (1999)
12. Zhang, C., Jacobsen, H.: Extended aspect mining tool (2002), `http://www.eecg.utoronto.ca/~czhang/amtex`
13. Zhang, C., Jacobsen, H.A.: Prism is research in aspect mining. In: OOPSLA, ACM, New York (2004)
14. Breu, S., Krinke, J.: Aspect mining using event traces. In: Automated Software Engineering (ASE) (2004)
15. Krinke, J., Breu, S.: Control-flow-graph-based aspect mining. In: 1st Workshop on Aspect Reverse Engineering (2004)
16. Breu, S.: Towards hybrid aspect mining: Static extensions to dynamic aspect mining. In: 1st Workshop on Aspect Reverse Engineering (2004)
17. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, Heidelberg (1999)
18. Tonella, P., Ceccato, M.: Aspect mining through the formal concept analysis of execution traces. In: Working Conference on Reverse Engineering (WCRE) (2004)
19. Tourwé, T., Mens, K.: Mining aspectual views using formal concept analysis. In: Source Code Analysis and Manipulation Workshop (SCAM) (2004)
20. Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonello, P., Tourwé, T.: A qualitative comparison of three aspect mining techniques. In: International Workshop on Program Comprehension (IWPC 2005), pp. 13–22. IEEE Computer Society Press, Los Alamitos (2005)
21. Mens, K., Tourwé, T.: Delving source-code with formal concept analysis. Elsevier Journal on Computer Languages, Systems & Structures (2005) (to appear)
22. Shepherd, D., Tourwé, T., Pollock, L.: Using language clues to discover crosscutting concerns. In: Workshop on the Modeling and Analysis of Concerns (2005)
23. Morris, J., Hirst, G.: Lexical cohesion computed by thesaural relations as an indicator of the structure of text. Computational Linguistics 17(1), 21–48 (1991)
24. Budanitski, A.: Semantic distance in wordnet: an experimental, application-oriented evaluation of five measures (2001)

25. Gybels, K., Kellens, A.: An experiment in using inductive logic programming to uncover pointcuts. In: First European Interactive Workshop on Aspects in Software (2004)
26. Gybels, K., Kellens, A.: Experiences with identifying aspects in Smalltalk using 'unique methods'. In: Workshop on Linking Aspect Technology and Evolution (2005)
27. Shepherd, D., Pollock, L.: Interfaces, aspects and views. In: Linking Aspect Technology and Evolution (LATE) Workshop (2005)
28. Karanjkar, S.: Development of graph clustering algorithms. Master's thesis, University of Minnesota (1998)
29. He, L., Bai, H., Zhang, J., Hu, C.: Amuca algorithm for aspect mining. In: Software Engineering and Knowledge Engineering (SEKE) (2005)
30. Marin, M., van Deursen, A., Moonen, L.: Identifying aspects using fan-in analysis. In: Working Conference on Reverse Engineering (WCRE), pp. 132–141. IEEE Computer Society Press, Los Alamitos (2004)
31. Shepherd, D., Gibson, E., Pollock, L.: Design and evaluation of an automated aspect mining tool. In: International Conference on Software Engineering Research and Practice (2004)
32. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: International Symposium on Static Analysis, pp. 40–56. Springer, Heidelberg (2001)
33. Krinke, J.: Identifying similar code with program dependence graphs. In: Working Conference on Reverse Engineering (WCRE'01), pp. 301–309. IEEE Computer Society Press, Los Alamitos (2001)
34. Bruntink, M., van Deursen, A., van Engelen, R., Tourwé, T.: An evaluation of clone detection techniques for identifying crosscutting concerns. In: International Conference on Software Maintenance (ICSM 2004), IEEE Computer Society Press, Los Alamitos (2004)
35. Bruntink, M., van Deursen, A., van Engelen, R., Tourwè, T.: On the use of clone detection for identifying crosscutting concern code. IEEE Transactions on Software Engineering 31(10), 804–818 (2005)
36. Baker, B.: On finding duplication and near-duplication in large software systems. In: Working Conference on Reverse Engineering (WCRE 1995), pp. 86–95. IEEE Computer Society Press, Los Alamitos (1995)
37. Baxter, I., Yahin, A., Moura, L., Sant Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: International Conference on Software Maintenance (ICSM 1998), IEEE Computer Society Press, Los Alamitos (1998)
38. Bruntink, M.: Aspect mining using clone class metrics. In: 1st Workshop on Aspect Reverse Engineering (2004)
39. Brant, J.: Hotdraw. Master's thesis, University of Illinois (1992)
40. van Deursen, A., Marin, M., Moonen, L.: AJHotDraw: A showcase for refactoring to aspects. In: Workshop on Linking Aspect Technology and Evolution (2005)
41. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Transactions on Software Engineering 29(3), 195–209 (2003)

Safe and Sound Evolution with SONAR

Sustainable Optimization and Navigation with Aspects for System-Wide Reconciliation

Chunjian Robin Liu, Celina Gibbs, and Yvonne Coady

University of Victoria, Canada

Abstract. Traditional diagnostic and optimization techniques typically rely on static instrumentation of a small portion of an overall system. Unfortunately, solely static and localized approaches are simply no longer sustainable in the evolution of today's complex and dynamic systems. Sustainable Optimization and Navigation with Aspects for system-wide Reconciliation[1] is a fluid and unified framework that enables stakeholders to explore and adapt meaningful entities that are otherwise spread across predefined abstraction boundaries. Through a combination of Aspect-Oriented Programming, Extensible Markup Language, and management tools such as Java Management Extensions, SONAR can comprehensively coalesce scattered artifacts—enabling evolution to be more inclusive of system-wide considerations by supporting both iterative and interactive practices. We believe this system-wide approach promotes the application of safe and sound principles in system evolution. This paper presents SONAR's model, examples of its concrete manifestation, and an overview of its associated costs and benefits. Case studies demonstrate how SONAR can be used to accurately identify performance bottlenecks and effectively evolve systems by optimizing behaviour, even at runtime.

1 Introduction

Today's complex systems' behaviours are increasingly difficult to understand and anticipate. One of the contributing factors is the increase in subtle interactions on all fronts—frameworks, middleware, virtual machines, and operating systems are all contributing factors in today's adaptive and autonomic systems. Purely static techniques for evolution are often no longer sustainable in these contexts. Heterogeneity and predefined abstraction boundaries are obstacles to system evolution. Though layering, componentization, and virtualization provide necessary levers for abstraction, behaviour that emerges along execution paths crossing these boundaries ultimately relegates local reasoning to be insufficient. For example, consider the ordeal of trying to find the root-cause of faults in a

[1] In seafaring terms, sonar systems ensure safety by using sound waves to detect underwater obstacles. The name of our tool plays on this word, as be believe it ensures safe and *sound* evolution in software systems.

A. Rashid and M. Aksit (Eds.): Transactions on AOSD IV, LNCS 4640, pp. 163–190, 2007.

system. It is not uncommon for meaningful application-level exceptions to be absorbed by middleware in a distributed system, and thus hidden from view [7]. Or, similarly, lower-level exceptions can sometimes be transformed to a different representation for higher levels to digest, making it more difficult to diagnose the root-cause of failure [8]. These scenarios highlight the need to reconcile issues such as fault analysis in terms of a system as a whole, as it is inadequate to try to diagnose problems when limited to one layer or component. We believe this same principle holds in the context of evolution. That is, it is often difficult to evolve a system in a safe and sound manner without considering the system as a whole.

Understanding and evolving system behaviour thus requires approaches that can flow freely across boundaries and provide comprehensive analysis that can be easily collected, correlated, and ultimately used to adapt applications to new circumstances, sometimes dynamically, as they evolve. Looking at this problem from another angle, complex system architectures must be viewed from multiple perspectives for multiple stakeholders [9]. Furthermore, in order to effectively aggregate data, views may need to be refined iteratively as focus changes during the process of analyzing interests [14]. Ideally, infrastructure to support views should be easily removed once users no longer need it, and hence incur little to no performance penalty. Recent technologies such as those employed by JFluid [27] go a long way to demonstrating that dynamic bytecode instrumentation can be both customized and efficient.

We argue that, for a large class of optimization strategies related to unanticipated external environment conditions, optimizations are becoming an increasingly substantial obstacle to effective evolution. Mixing optimization logic with application logic requires non-local information and makes both of them more difficult to understand, maintain, and evolve, due to idiosyncratic dependencies on external factors. Further, optimization code is context dependent and highly sensitive to dynamic factors such as server load, network traffic, and even order of operation completion. These factors make it particularly inefficient to encode certain kinds of optimizations in the absence of a-priori knowledge about execution contexts in terms of system-wide optimizations. We thus believe tool support to enable system-wide diagnosis and optimization must allow developers to apply and reconcile system monitoring and optimization techniques globally—across operating systems, virtual machines and applications. Further, we believe that at the application level, context-specific optimizations applied at runtime can supply much needed support for highly sensitive dynamic factors.

Sustainable Optimization and Navigation with Aspects for system-wide Reconciliation (*SONAR*) is a fluid and unified framework that allows stakeholders to dynamically explore and adapt meaningful entities that are otherwise spread across predefined abstraction boundaries. This allows for a safe and sound approach to system evolution—safe because it is informed, and sound because it is principled. SONAR is *fluid* in that it can leverage aspects to flow across boundaries in the system including the operating system, virtual machines, and

applications, and it is *unified* in that it provides a language-agnostic, holistic approach to diagnosis and optimization.

Through a combination of Aspect-Oriented Programming (AOP), Extensible Markup Language (XML), and management tools ranging from low-level system calls to high-level features such as Java Management Extensions (JMX), SONAR can comprehensively coalesce scattered artifacts. This enables iterative and interactive system-wide investigation and subsequently safe evolution. SONAR allows a view of the system to easily shift focus between coarser/finer-grained entities along principled points of execution paths that cross abstraction boundaries. At the application level, SONAR's model of deployment includes dynamic application of aspects to address the increasing need for runtime optimizations that can be customized to execution environments. As the deployment of such optimizations presents a new set of management challenges, SONAR further offers centralized support for a dynamic aspect repository to help prevent a disjointed view of the system as it is being altered at runtime.

In an effort to provide a sustainable solution to the problems encountered when trying to comprehend and effectively alter the behaviour of complex systems, SONAR's model of deployment was designed with three key requirements in mind:

- **Principled and system-wide instrumentation:** Instrumentation code must be introduced at principled points in the execution of the system. In order to be system-wide, these points must include all layers of the software stack, such as the operating system, the virtual machine, and the application. Furthermore, in the case of applications, dynamic instrumentation must be supported in a way that lends itself to centralized management. Finally, in the case of low-level infrastructure, instrumentation must have zero impact if disabled or removed. That is, there should be no residual scaffolding left behind.
- **Language/framework-agnostic definition:** To be able to define entities and data of interest across a spectrum of system elements implemented in a variety of programming languages, instrumentation must be language/framework independent.
- **Semantic representation:** To comprehensively maneuver and manage diagnostics and optimizations, data must be available for aggregation into a semantic representation that corresponds to a stakeholder's interest, and visualized/managed through easy to use standard-compliant tools when available. Alternatively, when such tools are not available, the introduction of customized tools or low-level interfaces to system diagnostic techniques must be supported in a way that can be extended to support further comprehensibility or filtering.

Given these requirements, we believe the SONAR model enforces system-wide evolution that is both better informed and principled than an ad hoc strategy. It is safe in that evolutionary changes can be accompanied by a more informed system-wide perspective, and sound because it based upon principled instrumentation strategies that can be replicated over the lifetime of the system.

This paper proceeds as follows. Sections 2 and 3 cover the ways and means SONAR uses to accomplish the goals stated above. Section 4 provides examples of system-wide optimization and navigation, covering operating systems, virtual machines, and application level examples. Section 5 presents a survey of related work in this area. An evaluation of SONAR's implementation based on the costs and benefits of the model in general, along with a more detailed performance and memory footprint evaluation for dynamic aspects in particular, is presented in Sect. 6. Section 7 concludes with a discussion of future work.

2 Background: AOP, XML, and Management Tools

This section briefly introduces the three key technologies used by SONAR to meet the established requirements. Both static and dynamic aspects are supported as a means to provide instrumentation that supports a crosscutting structure; XML is used as a language/framework-agnostic language to fit with multiple AOP frameworks; and finally, management tools such as JMX provide standard-compliant visualization and management.

Aspect-oriented programming modularizes crosscutting concerns—concerns that are present in more than one module, and cannot be better modularized through traditional means [4,20]. Looking at an aspect, a developer can see both the internal structure of a crosscutting concern and its interaction with the rest of the program during execution. Dynamic AOP allows aspects to be introduced/removed to/from a system at runtime. The current SONAR prototype provides support for AspectC [3], AspectJ [4], AspectWerkz [6] and Spring.NET AOP [31] as means of providing both static and dynamic AOP to structure system-wide crosscutting concerns for analysis and optimization.

The XML was originally designed to improve the functionality of the Web by providing more flexible and adaptable information identification [17]. Given that XML is not a fixed format like HTML (a single, predefined markup language), it can be generally used to customize markup languages. As such, SONAR uses an XML representation of aspects to achieve an AOP language-agnostic notation. An interesting technical advantage to this approach is that it allows SONAR to leverage XSL Transformations (XSLT) [33], a language for transforming XML into other documents, and a wide variety of XML-processing tools.

In the SONAR model, management tools must include a broad range of support services such as simple, low-level system calls, application-specific customized servers, and high-level standard-compliant environments. Each of these points in the management spectrum requires different levels of integration to be compatible with the overall SONAR approach. Low-level system calls, such as *vmstat* for monitoring memory usage on a Unix system can be deployed with simple shell scripts, whereas JMX [30] services for application and network management and monitoring require more infrastructure support for dynamic aspect management and state querying through standard management features. This support is detailed further in Sect. 3.2, and examples of each are provided in Sect. 4.

3 SONAR Design and Implementation

SONAR's model is designed to work with a range of AOP support and a range of management tools. Figure 1 shows a high-level perspective of this model, where an XML definition is first transformed to a specific, concrete representation in an AOP language and subsequently applied to the principled points in the execution of the system. This general model allows for aspects to be introduced at any level, from the application to the operating system, and to be available for visualization and management through a range of management tools, from simple system calls to more sophisticated JMX tools. The management interface tools are critical for effective aggregation and filtering of diagnostic data. Through the interface, collections of data should be either consumed or freed effectively, and collection artifacts such as buffer sizes for diagnostics must be easily configured when required. Simple aggregation functions can allow for the data to be processed at the time of collection, such as calculating averages as provided by DTrace [14]. As highlighted in Fig. 1, the differentiation of domain independent versus domain specific API for interfacing with tool support determines the degree to which the tool is portable. We envision most low-level tools to be tied to a domain specific API, whereas higher level tools can leverage standardized interfaces. Examples of each are provided in Sect. 4.

It is important to note here that data collection and communication introduce overhead, and sometimes this overhead can be unacceptable and perturb the system being observed. This consequence is commonly known as the observer effect. But it is the effect that SONAR is actually designed to minimize, by

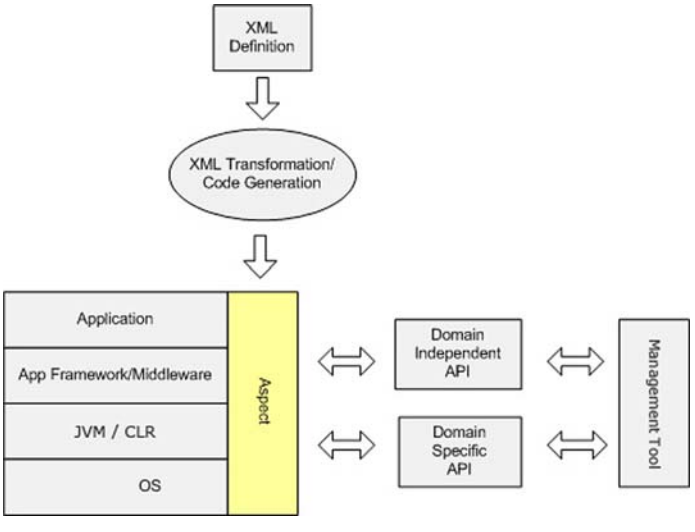


Fig. 1. SONAR architecture, showing a high-level overview of an XML definition of an aspect. The figure outlines how an aspect can be introduced to crosscut a system and interoperate with associated tool support.

allowing developers to more easily customize selected points during the execution of the system to be monitored, and allowing focus to change as information is gathered. This allows for precise collection of data used for system diagnosis, according exactly to the points of interest for the stakeholder.

In one particular manifestation of the SONAR model, dynamic aspects can be generated from XML-based definition files, deployed to applications/frameworks/middleware, and managed through JMX-compatible tools. Management of these dynamic aspects can be further enhanced through a centralized SONAR repository to help prevent a disjointed view of dynamic optimizations in the system. The APIs upon which the JMX tools manage the aspects break down into those that are domain independent, such as deployment/removal of aspects, and those that are domain specific, such as system navigation. From there, JMX can be used to visualize and manage the system. The following subsections provide more detail on this particular configuration for SONAR—that is, the concrete combination of dynamic AOP, XML and JMX, respectively.

3.1 Dynamic AOP Integration

To achieve dynamic instrumentation, we chose dynamic AOP since it provides language-level (code-centric) support for augmenting existing systems for various purposes. The common join point model shared by many existing AOP frameworks provides a solid foundation for implementing instrumentation. This model covers principled execution points in a system written in a number of languages, and thus enables the full range of fine and coarse-grained instrumentation required for meaningful system diagnosis and optimization.

Dynamic AOP further provides a powerful mechanism for runtime aspect manipulation such as runtime deployment/removal. In other words, advice can be dynamically woven into and removed from targets. The current implementation of SONAR uses AspectWerkz to provide dynamic AOP on the Java platform. To mitigate the impact of ad hoc optimizations clouding the overall clarity of the system, SONAR provides a centralized repository of the dynamic aspects

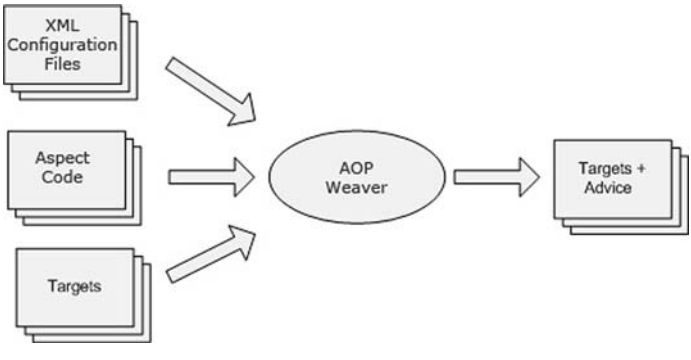


Fig. 2. XML configuration files, aspects, and targets are fed to the AOP weaver to produce targets and advice

currently applied, in order to promote reasoning about their composition in an executing system. Though this is a start, we believe a more comprehensive solution for reasoning about aspect-compositions is required. This is discussed in more detail in Sect. 7 with respect to future work.

Figure 2 depicts how three key ingredients come together to form the instrumented system: an XML configuration file, aspects and the target system to be diagnosed. The following subsections describe details on SONAR's use of XML and JMX, respectively.

3.2 XML Transformation/Code Generation

Aspects in SONAR are defined in AOP framework-independent XML files. Therefore, they can be implemented using different AOP frameworks or even in different programming languages such as Java or C. Examples from each of these languages are further explored in Sect. 4. Figure 3 shows a sample XML definition for an *httpMonitor* aspect. This aspect essentially monitors executions of the *process* method in the *Http11Processor* class.

The core content of every aspect specified in SONAR includes variable/method declarations, pointcut expressions, advice with actions, and parameter definitions. Variable/method declarations and actions are discussed in details in the following subsections. The current schema is mainly based on AspectWerkz' aspect definition schema, which is similar to those of other existing AOP frameworks. Additional elements of the schema are required for transformation and code generation.

More specifically, line 3 specifies the target system name, and that the aspect will be started automatically (auto). Automatically started aspects are enabled when the target systems are loaded, while manually started aspects have to be explicitly manually enabled (through JMX management tools) at the runtime. Line 4 specifies that there is exactly one of these aspects per JVM (other options include *perClass*, *perInstance*, and *perThread* as documented in [6]). Line 5 identifies a pointcut (*methodToMonitor*) associated with the execution of the *process* method, and lines 6-17 define the functionality (around advice, shown here as a special case of before/after with an automated proceed,[2] bound to the pointcut in line 5) to be applied. Line 18 defines the parameters to use for the aspect (details in [6]).

Since the target domain in SONAR is optimization and navigation, variable/method declarations and actions contain code targeting a domain-specific API. This defines the boundary between the aspect code and the domain specific target implementation. For example, the `log()` method used in Fig. 3 (lines 9 and 14) is defined outside of the aspect code and would be specific to the target domain. The aspect merely specifies its invocation. It may be implemented as printing to screen, writing to a log or sending to some management console. As a result, the implementation choice of such a method can be made independently

[2] This is a special feature of SONAR, and can be overridden if the *autoproceed* configuration option is set to *false*.

```
<?xml version="1.0" encoding="UTF-8"?>
<sonar>
  <system name="test" start="auto">
    <aspect name="httpMonitor" class="sonar.aspect.MonitorAspect"
    deployment-model="perJVM" manageable="true">
      <pointcut name="methodToMonitor" expression="execution(*
      org.apache.coyote.http11.Http11Processor.process(..))"/>
      <advice name="monitorTest(JoinPoint)" type="around"
      bind-to="methodToMonitor">
        <action type="before">
          <![CDATA[
            log("...");
          ]]>
        </action>
        <action type="after">
          <![CDATA[
            log("...");
          ]]>
        </action>
      </advice>
      <param name="..." value="..."/>
    </aspect>
  </system>
</sonar>
```

Fig. 3. Sample XML definition file showing around advice

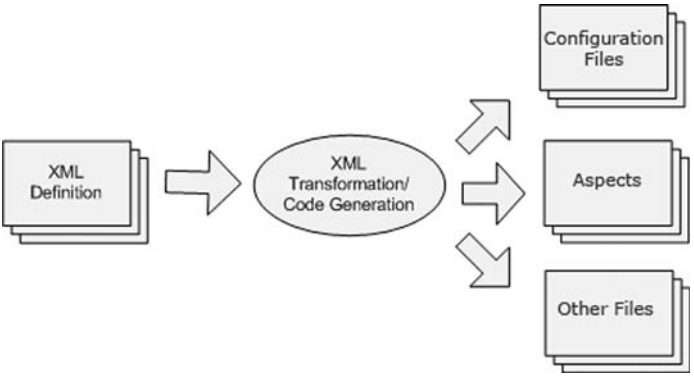


Fig. 4. XML definition files are transformed into configuration files and source code in a target language using XSLT and a domain-specific compiler/code generator

from aspect code and therefore, can be customized based on the target system and the target management tool. Further examples of how the use of domain independent and specific APIs can be used for both optimization and navigation are provided in the examples in Sect. 4.

As shown in Fig. 4, XSLT is used to transform XML definition files into other XML files, such as the aspect definition file for AspectWerkz (Fig. 5), aspect code (Fig. 6) and other required source code such as interfaces and helper

```
<?xml version="1.0" encoding="UTF-8"?>

  <!DOCTYPE aspectwerkz PUBLIC "-//AspectWerkz//DTD//EN"
  "http://aspectwerkz.codehaus.org/dtd/aspectwerkz2.dtd">

  <aspectwerkz>
    <system id="test">
      <aspect name="httpMonitor" class="sonar.aspect.MonitorAspect"
      deployment-model="perJVM">
        <pointcut name="methodToMonitor" expression="execution(*
        org.apache.coyote.httpll.HttpllProcessor.process(..))"/>
        <advice name="monitorTest(JoinPoint)" type="around"
        bind-to="methodToMonitor"/>
        <param name="..." value="..."></param>
    </aspect>
  </system>
</aspectwerkz>
```

Fig. 5. AspectWerkz' *aop.xml*, generated by transforming the definition file in Fig. 3

```
package sonar.aspect;

import org.codehaus.aspectwerkz.*;
import org.codehaus.aspectwerkz.definition.*;
import org.codehaus.aspectwerkz.joinpoint.*;
import org.codehaus.aspectwerkz.transform.inlining.deployer.*;

import sonar.util.*;

import java.lang.management.*;
import javax.management.*;
import javax.management.openmbean.*;

public class MonitorAspect implements MonitorAspectMBean {
  private final AspectContext aspectContext;

  public MonitorAspect(final AspectContext aspectContext) {
    this.aspectContext = aspectContext;
  }


  public Object monitorTest(final JoinPoint joinPoint) throws Throwable {
    log("...");

    final Object result = joinPoint.proceed();

    log("...");

    return result;
  }
}
```

Fig. 6. Java source code containing AspectWerkz-specific code, as prescribed in Fig. 5

classes for management purposes. If variable/method declarations and actions in the aspect definition are written in domain-specific languages, a domain specific compiler must be used to compile the code into the language used in the target system. SONAR accommodates this kind of heterogeneity by supplying multiple *transformers*, generating the language-specific aspects, one per target language.

3.3 JMX Management

Originally known as Sun's JMAPI, JMX [30] is gaining momentum as an underlying architecture for J2EE servers. It defines the architecture, design patterns, interfaces, and services for application and network management and monitoring. Managed beans (MBeans) act as wrappers, providing localized management for applications, components, or resources in a distributed setting. MBean servers are a registry for MBeans, exposing interfaces for local/remote management. An MBean server is lightweight, and parts of a server infrastructure are implemented as MBeans. SONAR uses JMX's support for dynamic aspect management and state querying to support both optimization and navigation through standard management features.

Java Management Extension is used as a means to comprehensively visualize and manage aspects introduced by SONAR. This includes retrieving data from aspects, invoking operations, and receiving event notification. Through JMX, aspects can be managed by JMX-compatible tools remotely and/or locally. For

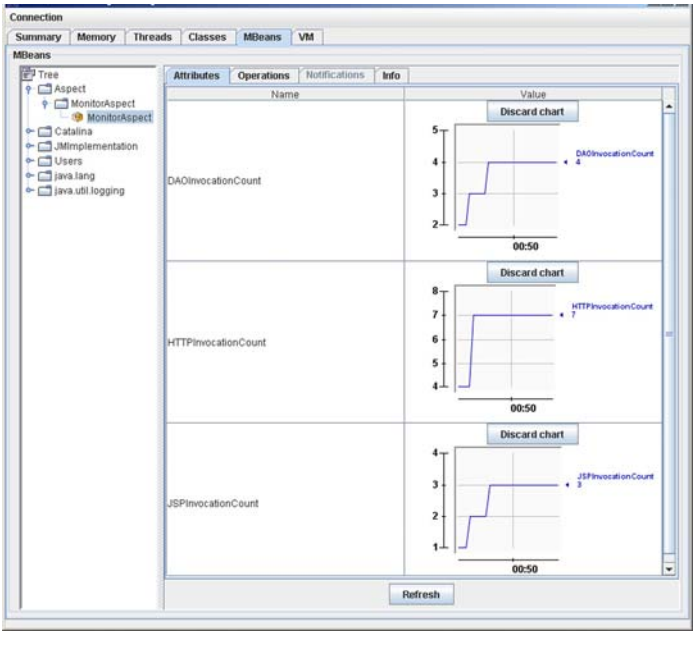


Fig. 7. Data from *MonitorAspect* is comprehensively visualized as data values change over time, and can be updated in JConsole

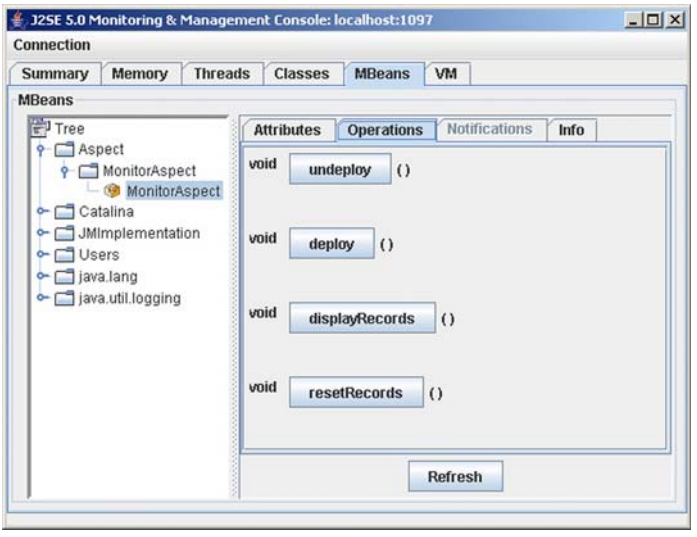


Fig. 8. Operations (both generic and domain specific) of *MonitorAspect* are listed and can be invoked manually in JConsole

example, JConsole, a JMX-compliant graphical tool for monitoring and management built into Sun's JDK distribution. Figures 7 and 8 show how JConsole can be used to manage aspects.

These figures specifically show how the simple *MonitorAspect*, which monitors HTTP requests, database access and JSP service, is visualized and managed in SONAR. Figure 7 illustrates how the statistics from three different invocation points collected by *MonitorAspect* can be visualized as line charts in JConsole. As these data points are updated in the running system, the charts are automatically updated as well, and accessed through the *attributes* tab of the JConsole interface (top left in the figure).

Figure 8 shows the operations supported by *MonitorAspect*. The deploy()/undeploy() buttons are used to manage the aspect at runtime. After being undeployed, advice defined in *MonitorAspect* are removed from all targets. The *MonitorAspect* still can be accessed by JConsole, and can be redeployed by invoking deploy() from this management interface (*operations* tab, shown at the top of the figure).

4 Case Studies: Optimization and Navigation with SONAR

This section demonstrates concrete manifestations of the SONAR model as it applies to three different levels of a system: the operating system, the virtual machine, and the application level. In the operating system, we consider a performance optimization for amortizing access costs to disk. The concrete

configuration for SONAR with this example consists of an XML transformer for AspectC and simple, rudimentary system interfaces for memory and disk management (Sect. 4.1). In the virtual machine, we consider the navigation of a memory management subsystem. This example uses SONAR's transformer for AspectJ, and a third-party customized management interface developed specifically for this navigation system as detailed in [29] (Sect. 4.2). At the application level, dynamic profiling and caching support are introduced and managed in the context of sample banking and reservation system applications, respectively. This final example uses the AspectWerkz transformer in SONAR, and couples these dynamic aspects with standardized JMX management tools (Sect. 4.3).

4.1 Operating System Optimization

We have studied and presented the impact of aspects on operating system code in previous work [10]. Here we demonstrate how these same operating system aspects, refactored from the original source code, can be incorporated into the SONAR model. We consider an optimization for accessing files that have been mapped into main memory. This *prefetching* aspect performs a read-ahead from disk in order to reduce access latencies. We use SONAR's AspectC transformer to realize the aspect, which is then statically incorporated into the build of the system. We rely only on the most simple, low-level, pre-existing system diagnostic tools in order to loosely monitor impact of the functionality introduced. As the AspectC prototype does not support dynamic aspects, the instrumentation cannot be modified at runtime. We believe that, at this level, these management tools at least provide a starting point for cost-effective system monitoring and management. In future work we plan to explore facilities for aggregating and filtering data provided by these pre-existing tools.

An effective system optimization found in several versions of FreeBSD [18] concerns *prefetching*, a heuristic used to reduce disk latencies by attempting to bring pages into memory in advance of explicit requests for them. For example, if sequential access to a file is detected, a page fault will result in a disk request not only for the missing page, but also for several pages in advance of this page, in anticipation of future accesses. Here we consider an aspect associated with this functionality for FreeBSD v3.3. SONAR's XML specification for the prefetching aspect is shown in Fig. 9, and its subsequent manifestation in AspectC is shown in Fig. 10.

This static aspect specifies an allocation of virtual pages for prefetched pages in the first advice, and their subsequent de-allocation in the event it is not cost-effective to retrieve them in the next three advice. This functionality is a refactoring of an optimization commonly found within FreeBSD, but in this form it is an optimization that can be added/removed easily at compile-time.

Rustic management tools exist for understanding behaviour associated with this and other virtual memory optimizations. For example, many Unix operating systems such as FreeBSD support system calls such as *vmstat*, shown in Fig. 11. For example, *vmstat* pauses a given number of seconds between each display, then reports statistics for kernel thread, virtual memory, disk, trap, and CPU

```
<?xml version="1.0" encoding="UTF-8"?>
<sonar>
  <system name="file">
    <aspect name="filePrefetch" class="sequential_mapped_file_prefetching" language="AspectC">
      <import>...</import>
      <pointcut name="fault_path" params="vm_map_t map"
      expression="cflow(execution(int vm_fault(map, vm_offset_t, vm_prot_t, int)))"/>

      <pointcut name="ffs_read_path" params="struct vnode* vp, struct uio* io_info, int size, struct buff** bpp"
      expression="cflow(execution(int ffs_read(vp, io_info, size, bpp)))"/>

      <advice type="before">
        <pointcut params="vm_map_t map, vm_object_t object, vm_page_t* pagelist, int length, int faulted_page">
          <expression>
            <![CDATA[
              execution(int vnode_pager_getpages(object, pagelist, length, faulted_page))
              && fault_path(map)
            ]]>
          </expression>
        </pointcut>
        <action>
          <![CDATA[
           if (object->declared_behaviour == SEQUENTIAL) {
             vm_map_lock(map);
             plan_and_alloc_sequential_prefetch_pages(object, pagelist, length, faulted_page);
             vm_map_unlock(map);
           }
          ]]>
        </action>
      </advice>

      <advice type="around" autoProceed="false">
        <pointcut params="vm_object_t object, vm_page_t* pagelist, int length, int faulted_page"
        expression="execution(int ffs_getpages(object, pagelist, length, faulted_page))"/>
        <action>
          <![CDATA[
            if (object->behaviour == SEQUENTIAL) {
               struct vnode* vp = object->handle;
               struct uio* io_info = io_prep(pagelist[faulted_page]->pindex,MAXBSIZE, curproc);
               int error = ffs_read(vp, io_info, MAXBSIZE, curproc->p_ucred);
               return cleanup_after_read(error, object, pagelist, length, faulted_page);
            } else
               proceed(object, pagelist, length, faulted_page);
          ]]>
        </action>
      </advice>

      <advice type="after">
        <pointcut params="struct uio* io_info, int size, struct buf** bpp">
          <expression>
            <![CDATA[
              execution(int breadn(struct vnode *, daddr_t, int, daddr_t*, int*,int, struct ucred*, struct buf**))
              && fault_path(vm_map_t, vm_offset_t, vm_prot_t, int)
              && ffs_read_path(struct vnode*, io_info, size, bpp)
            ]]>
          </expression>
        </pointcut>
        <action>
          <![CDATA[
            flip_buffer_pages_to_allocated_vm_pages((char *)bpp->b_data, size, io_info);
          ]]>
        </action>
      </advice>

    </aspect>
  </system>
</sonar>
```

Fig. 9. XML specification of a prefetching aspect for the AspectC transformer

activity. On multiprocessor machines, however, it is important to use *mpstat*, shown in Fig. 12, for per processor statistics. We believe that coupling these optimizations with the right management tools is one of the benefits associated with SONAR's model.

Prefetching is a common feature in operating system code. In SONAR, not only is its structure improved through modularization as an aspect, but in this form the concern is better coupled with the tools that monitor resources it

```
#include ...

aspect sequential_mapped_file_prefetching {

  pointcut fault_path(vm_map_t map):
  cflow(execution(int vm_fault(map, vm_offset_t, vm_prot_t, int)));

  pointcut ffs_read_path(struct vnode* vp, struct uio* io_info, int size, struct buff** bpp):
  cflow(execution(int ffs_read(vp, io_info, size, bpp)));

  before(vm_map_t map, vm_object_t object, vm_page_t* pagelist, int length, int faulted_page):
  execution(int vnode_pager_getpages(object, pagelist, length, faulted_page))
  && fault_path(map) {
     if (object->declared_behaviour == SEQUENTIAL) {
       vm_map_lock(map);
       plan_and_alloc_sequential_prefetch_pages(object, pagelist, length, faulted_page);
       vm_map_unlock(map);
     }
  }

  around(vm_object_t object, vm_page_t* pagelist, int length, int faulted_page):
  execution(int ffs_getpages(object, pagelist, length, faulted_page)) {
    if (object->behaviour == SEQUENTIAL) {
       struct vnode* vp = object->handle;
       struct uio* io_info = io_prep(pagelist[faulted_page]->pindex,MAXBSIZE, curproc);
       int error = ffs_read(vp, io_info, MAXBSIZE, curproc->p_ucred);
       return cleanup_after_read(error, object, pagelist, length, faulted_page);
    } else
        proceed(object, pagelist, length, faulted_page);
  }

  after(struct uio* io_info, int size, struct buf** bpp):
  execution(int breadn(struct vnode *, daddr_t, int, daddr_t*, int*,int, struct ucred*, struct buf**))
  && fault_path(vm_map_t, vm_offset_t, vm_prot_t, int)
  && ffs_read_path(struct vnode*, io_info, size, bpp) {
     flip_buffer_pages_to_allocated_vm_pages((char *)bpp->b_data, size, io_info);
  }

}
```

Fig. 10. Prefetching aspect generated in AspectC

```
cascade<5> vmstat 5
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr s6 sd sd --   in   sy   cs us sy id
 0 0 0 1639616 1820976 24 52 107 0 0  0  0  0  1  0  0  254  244   35 22  6 72
 0 0 0 1356288 1376872 0  1  0  0  0  0  0  0  0  0  0  624 5253  890  0  0 99
 0 0 0 1357128 1377504 4 20  2  0  0  0  0  0  0  0  0  836 5575  993  0  3 97
…
```

Fig. 11. Rudimentary data aggregation with *vmstat*

```
cascade<18> mpstat 5
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   25   1  162   150  132   22   33   81   36    0   136   27   6   0  67
  1   27   2  122   106    6   14  100   81   48    0   116   17   6   0  77
<<State change>>
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0   25   1  162   150  132   22   33   81   36    0   136   27   6   0  67
  1   27   2  122   106    6   14  100   81   48    0   116   17   6   0  77
```

Fig. 12. Rudimentary data aggregation with *mpstat*

is designed to optimize. In future work, we consider the ability to tune this optimization further, according to application specific needs and across the kernel boundary.

4.2 Virtual Machine Navigation

GCSpy is general purpose heap visualization framework that allows developers to perform dynamic analysis of memory consumption [29]. It is designed to visualize a wide variety of memory management systems, offering dynamic visualization of workloads for either garbage collected or manually controlled systems.

GCSpy is based on a client–server architecture where the system that is being visualized is the server and the visualization GUI is the client, communicating through standard sockets. A screenshot of the client is shown in Fig. 13, highlighting used versus unused space in memory, and supplying an interface for the full suite of GCSpy functionality. The advantages of this client–server design are that a minimal amount of code is added to the system being visualized, as the client can be run on another machine to affect the operation of the server as little as possible, and the client can be connected to and disconnected from the server anytime. Only the server side of GCSpy needs to be customized for a particular system, and the client is a portable GUI, making it suitable as a management tool within SONAR.

Here we consider the application of the SONAR model for introducing a *GCSpy* aspect within the Jikes RVM [1,19]. The XML specification of a small portion of the *GCSpy* aspect for AspectJ is shown in Fig. 14, and the corresponding aspect is shown in Fig. 15. The management tool is unchanged from that shown in Fig. 13. This code shows two simple advice, and highlights a key issue associated with the SONAR model. The issue highlighted here deals with effective refactoring for SONAR.

With respect to the refactoring issue, the incarnation of the *GCSpy* aspect that is appropriate for SONAR is different from the first refactoring of the *GCSpy* aspect presented originally in [19]. In SONAR, raw code in the XML template should be minimized, as it will only be exposed to a rudimentary XML editor

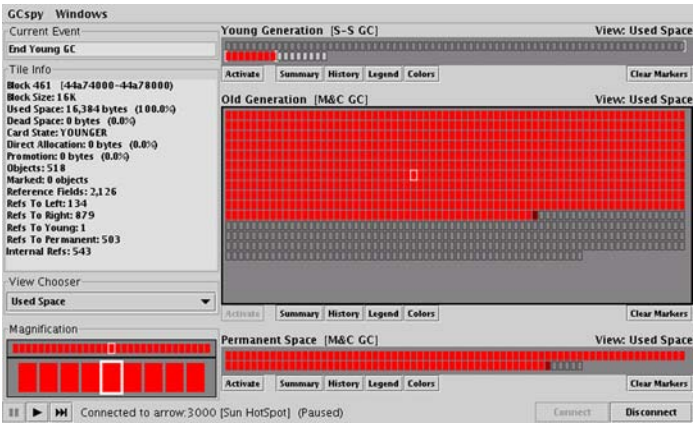


Fig. 13. This screenshot shows GCspy visualizing Sun's Java HotSpot virtual machine running the SPECjvm98_213_javac benchmark

```
<?xml version="1.0" encoding="UTF-8"?>
<sonar>
  <system name="gcspy">
    <aspect name="gcSpy" class="org.mmtk.plan.GCSpy" baseClass="org.mmtk.plan.GCSpyBase"
    deployment-model="perJVM" language="AspectJ">
      <import>org.mmtk.vm.gcspy.*</import>

      <pointcut name="planBoot" expression="execution(* Plan.boot())"/>
      <advice name="planBoot" type="before" bind-to="planBoot">
        <action>
        <![CDATA[
          planBoot();
        ]]>
        </action>
      </advice>

      <pointcut name="planPostAlloc" params="VM_Address ref, Object[] o, int bytes, boolean b, int allocator"
      throws="VM_PragmaUninterruptible">
          <expression>
            <![CDATA[
              execution(* Plan.postAlloc(VM_Address, Object[], int, boolean, int))
              && args(ref, o, bytes, b, allocator)
            ]]>
          </expression>
      </pointcut>
      <advice name="planPostAlloc" type="around" bind-to="planPostAlloc" autoProceed="false">
        <action>
        <![CDATA[
          if(! planPostAlloc(ref, o, bytes, b, allocator))
          {
            proceed(ref, o, bytes, b, allocator);
          }
        ]]>
        </action>
      </advice>

    </aspect>
  </system>
</sonar>
```

Fig. 14. XML specification of GCSpy aspect for AspectJ transformer

```
package org.mmtk.plan;

import org.mmtk.vm.gcspy.*;

privileged aspect GCSpy extends GCSpyBase {
  before():
  execution(* Plan.boot()) {
    planBoot();
  }

  void around(VM_Address ref, Object[] o, int bytes, boolean b, int allocator) throws VM_PragmaUninterruptible:
  execution(* Plan.postAlloc(VM_Address, Object[], int, boolean, int))
  && args(ref, o, bytes, b, allocator) {
    if(! planPostAlloc(ref, o, bytes, b, allocator)) {
      proceed(ref, o, bytes, b, allocator);
    }
  }
}
```

Fig. 15. GCSpy aspect generated in AspectJ

and not full IDE support. Hence, the form of the aspect presented here leverages a helper class to include the bulk of the implementation, reducing the XML definition to the dependency that exists between the pointcut/method signatures. This way the core functionality of GCSpy can be still edited in the rich context of tool support present in the IDE, instead of a plain XML editor.

The *GCSpy* aspect demonstrates a simple application of the SONAR model applied at the VM level. In this example, the original implementation is coupled with a predefined monitoring tool. Coordinating this example with other memory

management concerns controlled by SONAR in the operating system (such as the previous example), and other monitoring tools with feedback capabilities, provides a viable means for a system-wide approach to memory management.

4.3 Application Level Optimization and Navigation

This section offers examples from two different sample applications. The first is based on a modified version of the J2EE sample banking application, *Duke's Bank*, and makes use of SONAR's transformer for AspectWerkz (Sect. 4.3.1). The second is based on a modified version of Spring.NET's sample airline reservation system, *SpringAir*, and makes use of the Spring.NET AOP transformer (Sect. 4.3.2). The associated costs of the deployment of dynamic aspects such as these are further evaluated in Sect. 6.

4.3.1 Optimization and Navigation of a Simple Banking Application

The following example, *ProfileAspect*, demonstrates SONAR's ability to navigate a sample system. It is based on a modified version of the J2EE sample banking application, *Duke's Bank*. Given that the scenarios considered in the example are request-centric, the modification to the code was to provide a tag for each request. That is, a unique identification (ReqID) is assigned to every request, be it generated from a servlet or JSP access, and sent to the server. The ReqID is retrieved when a request reaches Tomcat's HTTP adapter, and saved to a thread-local variable in order to be accessed by subsequent operations.

This profiling aspect is not enabled (i.e., its deployment strategy is *manual*) when the system is started. The aspect itself is registered as a standard MBean (management bean) to the JBoss JMX server. The stakeholder can thus enable profiling through domain-independent API (deploy/undeploy) shown in Fig. 8.

The *ProfileAspect* reflects a request-centric view of the system, recording key data points as requests are serviced. As a result, it exposes several key configurable optimization and navigation options and operations through JConsole, such as the ability to:

- enable/disable tracing to specified classes and/or methods, including those in JBoss, Tomcat, and Duke's Bank,
- apply a filter, consisting of a ReqID pattern, to filter out unwanted data,
- configure tracing details (stack trace, timestamps, duration),
- manipulate buffer operations (change the size, clear the buffer).

All of the above options and operations are accessible through this aspect's JMX interface. All data is stored on the server side and can be retrieved and viewed through the JMX management tool.

Figure 16 visually depicts the information collected by SONAR regarding a stack trace of serving a request to retrieve customer accounts. Each blue bar indicates the processing time (in ms) of a method, the summation of the time spent in processing its method body and subsequent method calls. The top level is the Tomcat adapter—the entry point of serving an HTTP request. Access to

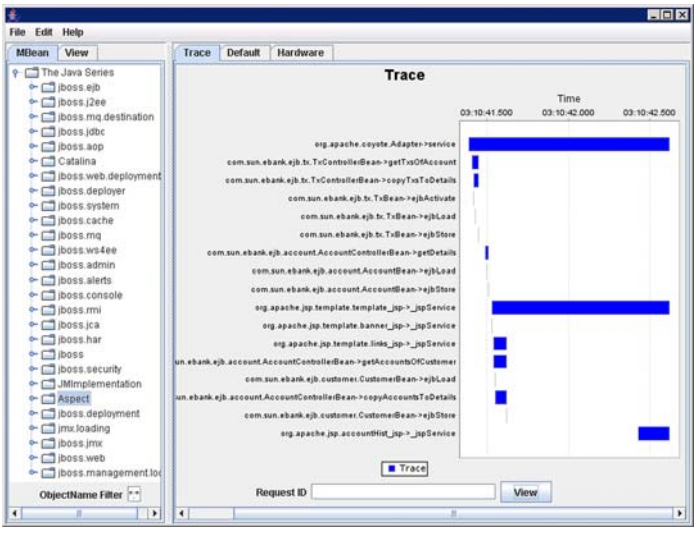


Fig. 16. The stack trace of serving a request that retrieves a list of customer accounts and their corresponding balances

SessionBeans, EntityBeans and JSPs are traced to clearly show the processing time in each layer according to J2EE architecture.

This banking application demonstrates the role of dynamic aspects within the SONAR model. The JMX management tool provides a means of tuning operations for tracing and buffering within the application. Though the tools from the previous examples have been far more rudimentary, we envision this kind of management facility to be applied system-wide—across application, VM and operating system boundaries.

4.3.2 Cross-Platform Optimization

We chose *SpringAir*, a web application built upon Spring.NET framework, as our second case study. Specifically, this study demonstrates SONAR's cross-platform support for optimization. The basic idea is to retrieve some cached data from remote systems and build a local copy in order to save the time spent on network communication. Optimizations that improve locality can dramatically impact the evolvability of many of today's web-service based applications.

We developed an *AirportInfoCacheAspect* which provides caching for airport information. As shown in Fig. 17, the *airportInfoCacheAdvice* is applied to the *DefaultBookingAgent*'s service methods with names matching the *GetAirport.** pattern Fig. 18.

Spring.NET AOP is quite different from other AOP frameworks. Therefore, the XML specification schema is extended in order to address such differences. The most noticeable changes are made to the *pointcut* tag, since Spring.NET AOP does not define an AspectJ-like pointcut expression language. Furthermore, in Spring.NET AOP, advice must implement one of the built-in advice interfaces

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sonar>
3    <system name="springAir">
4      <aspect name="airportInfoCacheAspect" objectID="bookingAgent"
5      deployment-model="perClass" language="Spring.NET">
6
7        <pointcut name="airportInfo" type="RegexpMethod">
8          <target-type>SpringAir.Service.DefaultBookingAgent</target-type>
9          <target-method-pattern>GetAirport.*</target-method-pattern>
10       </pointcut>
11
12       <advice name="airportInfoCacheAdvice" class="SpringAir.Cache.AirportInfoCacheAdvice"
13       type="around" bind-to="airportInfo" autoProceed="false">
14         <import>SpringAir.Domain;Sonar.Cache</import>
15         <action>
16           <![CDATA[
17             string airportInfoName = invocation.Method.Name.Substring(3);
18
19             object airportInfo = GlobalCache.get(airportInfoName);
20
21             if(airportInfo != null)
22             {
23               return airportInfo;
24             }
25
26             airportInfo = invocation.Proceed();
27
28             GlobalCache.insert(airportInfoName, airportInfo);
29
30             return airportInfo;
31           ]]>
32         </action>
33       </advice>
34
35     </aspect>
36   </system>
37 </sonar>
```

Fig. 17. XML specification of cache aspect for Spring.NET transformer

(i.e. *IMethodInterceptor*). As a result, a class cannot define two advice of the same type. Therefore, as in lines 12 and 13, the *class* attribute and *import* tag are added to the *advice* tag.

Spring.NET AOP supports applying advice programmatically or declaratively using XML configuration. Figure 19 shows the generated XML segment.

Since there is no direct support for JMX in either Spring.NET or .NET framework, we decided to expose the cached information through a web service by using .NET's web service support. In this way, the cached data is accessible by any programs with web services support. Our Java client uses the Java web service API to access the cached airport information in XML, and parses the data to build a local cache.

In addition to crossing application, VM, and operating system boundaries within a single node, this final example demonstrates how SONAR can apply across multiple systems. This opens the possibility of monitoring and tuning cross-platform concerns, and even further coordinating them with lower-level resource managment conforming to the SONAR model.

```
using SpringAir.Domain;
using Sonar.Cache;

using AopAlliance.Intercept;
using Spring.Aop;

using System;

namespace SpringAir.Cache {
  public class AirportInfoCacheAdvice : IMethodInterceptor {
    public object Invoke(IMethodInvocation invocation) {
      string airportInfoName = invocation.Method.Name.Substring(3);

      object airportInfo = GlobalCache.get(airportInfoName);

      if(airportInfo != null) {
        return airportInfo;
      }

      airportInfo = invocation.Proceed();

      GlobalCache.insert(airportInfoName, airportInfo);

      return airportInfo;
    }
  }
}
```

Fig. 18. Cache aspect generated in Spring.NET

```
<object id="airportInfoCacheAdvice"
  type="Spring.Aop.Support.RegexpMethodPointcutAdvisor">
  <property name="pattern" value="GetAirport.*"/>
  <property name="advice">
    <object type="SpringAir.Cache.AirportInfoCacheAdvice"/>
  </property>
</object>

<object id="bookingAgent" type="Spring.Aop.Framework.ProxyFactoryObject">
  <property name="target">
    <object type="SpringAir.Service.DefaultBookingAgent"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>airportInfoCacheAdvice</value>
    </list>
  </property>
</object>
```

Fig. 19. Configuration XML generated in Spring.NET

5 Related Work

Given the growing need for more holistic, system-wide, diagnostic tools, it is no surprise that SONAR is one of many projects working to address this and related challenges. Within this spectrum, SONAR sits as a lightweight dynamic approach, which could effectively be used in concert with several more heavyweight approaches.

Pinpoint [7,8] is a dynamic analysis methodology that automates problem determination in complex systems by coupling coarse-grained tagging of client requests with data mining techniques. Data mining correlates failures and successes of client requests as they pass through the system. This combined approach is used to determine which component(s) are most likely to be at fault, and has been applied to the problem of root-cause analysis on the J2EE platform with impressive results. In terms of tracing client requests, Pinpoint and SONAR are very similar, but Pinpoint provides data mining where SONAR simply offers an iterative and interactive interface for human control. In terms of failure detection, Pinpoint uses traffic sniffing and manually provided middleware instrumentation. SONAR again relies on human interaction to navigate to points of interest, and dynamically deploys/removes aspects for instrumentation. We believe a future merger of these two approaches could provide the best spectrum of support for complex system diagnosis.

Magpie's [24] goal is to provide synthesis of runtime data into concise models of system performance. In Magpie, online performance modeling is an operating system service. Magpie's modeling service collates detailed traces from multiple machines, extracts request-specific information, and constructs probabilistic models of request behaviour. It would be possible to adopt some of the strategies used by Magpie for distribution and performance debugging within SONAR with the intention of applying it to further environments, in particular operating system services. As SONAR uses language-agnostic definitions for instrumentation points, it could be used to deploy dynamic aspects for C [13,16] into operating system services.

DTrace [14] is a unified tracing toolkit for both system and application levels. DTrace can be used to observe, debug and tune systems using the D programming language which is designed specifically for tracing. As a result, it is a comprehensive dynamic tracing framework, applicable within the Solaris Operating Environment, FreeBSD, and Mac OS X. DTrace attains many of the goals shared by SONAR, to monitor, debug and tune systems and runtime from multiple perspectives, but within a proprietary environment.

JFluid [27] is a profiler built into the NetBeans IDE [26]. Profiling functions, such as monitoring CPU, memory and threads, aids performance-related diagnostics. JFluid uses highly efficient dynamic bytecode instrumentation, making it possible to use the tool on the fly. A mechanism in the JVM called HotSwap [12] allows users to dynamically turn profiling on/off and to profile just a selected subset of the code. The subset and target of profiling (CPU, memory, etc.) can be changed at run time. Dynamic bytecode instrumentation is guaranteed not to alter program semantics, as it only impacts well-defined events, such as method

entry/exit, and object allocation. JFluid demonstrates the ability to provide highly efficient bytecode instrumentation, which bodes well for costs associated with dynamic AOP. We believe that this analysis of JFluid technology indicates that the implementation of navigation in SONAR could be very low cost and localizable, though we expect that the ability to adapt or optimize will come at a cost.

PEM/K42 [15] uses an approach called *vertical profiling* to correlate performance and behaviour information across all layers of a system (hardware, operating system, virtual machine, application server, and application) to identify causes of performance problems. The Performance and Environment Monitoring (PEM) and K42 Operating System groups [11,22] at IBM Research are getting promising results using this and a set of other approaches to develop effective system diagnosis tools. Though SONAR in comparison is a much more lightweight approach, we believe it could be an early prototype of a tool that would fit with this family. In particular, the use of JMX in SONAR was inspired by the comprehensive interfaces used by PEM to provide visualization and management of system diagnostics.

PROgrammable extenSions of sErvices (PROSE) [28] is an adaptive middleware platform for dynamic AOP which allows aspects to be woven, unwoven or replaced at runtime. It provides middleware tools that allow runtime monitoring of remote aspects, a creation wizard, and transaction-based aspect insertion/withdrawal. PROSE aspects are represented as plain Java objects and can be sent to computers on a network. Once an aspect has been inserted into a JVM it will execute advice as expected until it is removed. Though dynamic AOP is something common to both PROSE and SONAR, SONAR simply leverages it as means for dynamic instrumentation at an application level, whereas PROSE goes much deeper to leverage it for pervasive and coordinated access to the system's resources. We believe that, in the long term, these lightweight/heavyweight approaches will converge, allowing high-level management of core system components supplied by aspects.

In terms of convergence, we also see a great number of other approaches to AOP that we would like to incorporate as an option within the SONAR model. CaesarJ [2] is a new Java based programming language where components are collaborations of classes. CaesarJ is used to better modularize crosscutting features or non-functional concerns by providing explicit support to implement, abstract and integrate such components. More recently, Aspects with Explicit Distribution (AWED) [25] has proposed an approach for the implementation of crosscutting functionalities in distributed applications with several distinct features specific to a distributed programming domain. These features include remote pointcuts, distributed advice, and distributed aspects, and each support natural notions of state sharing and aspect instance deployment among a group of hosts.

In terms of dynamic AOP and profiling, other important related work includes TOSKANA [16] and Glassbox [21]. TOSKANA provides a toolkit for deploying

dynamic aspects into an operating system kernel. It provides before, after and around advice for kernel functions and supports the specification of pointcuts and the implementation of aspects as dynamically exchangeable kernel modules. Glassbox uses aspect libraries for profiling and troubleshooting Java applications, automatically diagnosing common problems. These services allow developers to spend less time dealing with logging and debugging, and provide an efficient means of performance tuning.

The language-agnostic goal of SONAR is most closely related to Lafferty's work [23] which supports the AspectJ notion of AOP which is also consistent with the component model of the .NET Framework. In this work, aspect-based properties are implemented as Common Language Infrastructure (CLI) components with XML-based crosscutting specifications, and load-time weaving is used. Though the aspect-component bindings are written in terms of attribute types, additional support for custom crosscutting can be specified in terms of CLI metadata, enabling further language-independence. We envision this to be a complementary approach to SONAR's current approach to transformations, and seek to include this as a further option for transformation in future incarnations of SONAR.

6 Analysis: Costs and Benefits of SONAR

Here we consider costs in terms of performance and memory, and benefits in terms of the strength of the overall programming model of SONAR. In terms of costs, with any approach to instrumentation there are overheads, but the overheads associated with the static aspects (Sect. 4.1, 4.2) are less substantial than those associated with dynamic AOP (Sect. 4.3). In Sect. 6.1 we focus on quantification of these costs associated with the dynamic aspects such as those introduced in Sect. 4. With respect to benefits, Sect. 6.2 highlights the way we believe the SONAR model to be useful, and the extensibility of this approach within other contexts.

6.1 Costs: Performance and Memory Utilization

All tests reported in this section were conducted on a Pentium 4 2.4 GHz, 512 MB machine, running Windows XP SP1, JDK 1.5.0 (1.5.0_02).

Since SONAR is designed not only for development systems but also for production systems, performance is crucial for its applicability. Moreover, system navigation tasks usually require introspection of system state, but the process of navigation might adversely impact system state and behaviour [32]. As a result, the performance impact of SONAR should be as minimal as possible. In this section we consider costs associated with SONAR. In order to get an impression of the worst case, we consider the most heavyweight approach supported by the prototype—the dynamic bytecode instrumentation provided by AspectWerkz.

Currently, no JVM supports schema redefinition of any loaded classes. That is, changes (such as add, remove or rename fields or methods, change of method signatures or inheritance) are not allowed at runtime[3] [6]. However, dynamic deployment/undeployment of aspects in AspectWerkz requires schema changes to target classes.

To get around this restriction, AspectWerkz uses a preparation mechanism to enable target classes for later deployment/undeployment at runtime. A special construct called *deployment scope* is used to specify the join points to be prepared by adding a call to a `public static final` method that redirects to the target join point. The added indirection introduces overhead; however, such indirection can be inlined by most modern JVMs [6]. As a result, the impact of the added hook on runtime performance is negligible when aspects are not woven into or removed from target classes. Even for woven aspects, the time needed to instrument an advised method invocation has been optimized in the AspectWerkz 2 implementation [5].

Upon weaving or preparation, hooks are inserted into target classes. Consequently, the size of target classes is increased. Table 1 shows the impact of using AspectWerkz on target class size. The added size to each target class file is around 1,000 bytes. This does not include the size of aspect classes themselves since only the hook code is inserted into target classes. For medium and large classes, this is still relatively small. Table 1 also shows as the original target class size increases from 7,738 to 25,521 bytes, the impact decreases from 12.68 to 3.16%.

Table 1. Impact on target class size

	javax.servlet.http.HttpServlet	org.apache.coyote.http11.Http11Processor
Original (bytes)	7,738	25,521
Post weaving (bytes)	8,719	26,327
Increase (%)	12.68	3.16

Both classes have a single around advice woven to a single method, respectively, using AspectWerkz 2.0

As shown in Fig. 20, the JBoss Application Server's startup time is increased by about three times. The performance is significantly degraded since it is running under AspectWerkz' online mode—aspects are woven into target classes when they are loaded into the JVM. Furthermore, as shown in Fig. 21, the memory footprint is also increased by about 28 Mb (31.30 %).

We should mention here that one limitation of SONAR is that it does not support distribution. As mentioned in future work, however, we would like to explore the incorporation of a development such as AWED into the SONAR

[3] As mentioned in the Java APIs instrumentation section, this restriction might be annulled in the future.

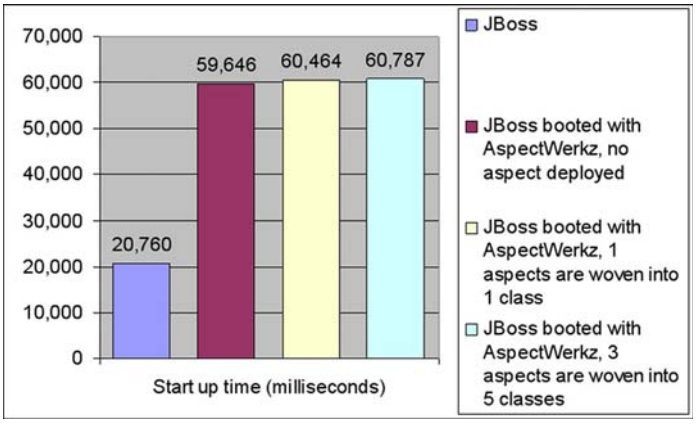


Fig. 20. Impact on startup time. Sampled by running JBoss Application Server 4.0.1 SP1 with embedded Tomcat 5 and AspectWerkz 2.0.

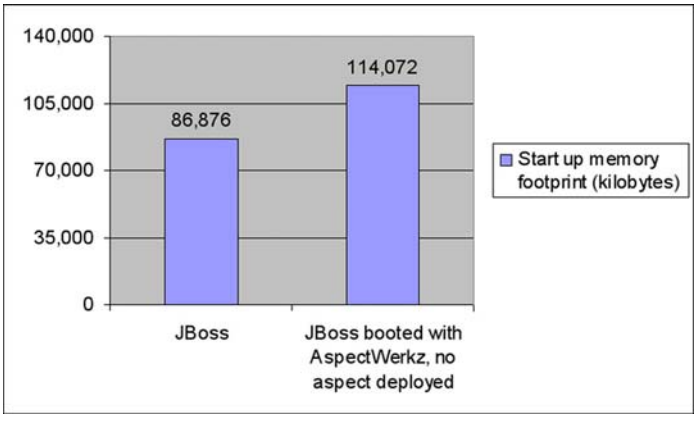


Fig. 21. Impact on memory footprint. Sampled under the same setting as in Fig. 20.

model. However, in SONAR's current prototype, by using JMX, aspects deployed in a distributed fashion could ultimately be accessed and managed through the JMX Remote API.

6.2 Benefits: Programming Model and Unified Framework

We believe the key benefit to SONAR is its model for a unified framework. We have demonstrated concrete manifestations of this model in three different SONAR configurations in Sect. 4: (1) using the AspectC transformer and low-level system interfaces for management (Sect. 4.1), (2) using the AspectJ transformer and a customized client-server management tool (Sect. 4.2), and (3)

using the AspectWerkz transformer and standardized JMX management tools (Sect. 4.3). Each of these configurations promotes an integrated approach for optimization/navigation with corresponding management interfaces.

We believe this establishes the efficacy of a relatively language-agnostic approach in the context of system-wide evolution. Of course, it is not completely language independent, but is extensible in that regard. Though the tradeoffs in terms of specific costs associated with dynamic approaches will need to be visited further in terms of impact within large systems, the fact that SONAR can be used to accurately identify performance bottlenecks at many levels in the system and thus be part of effective evolution strategies in a principled way may indeed outweigh performance compromises in some cases. We believe this model will continue to extend to other contexts, such as embedded and distributed. This future work is discussed further in the following section.

7 Future Work and Conclusions

The current SONAR prototype includes transformers for AspectWerkz, AspectJ, Spring.NET AOP, and AspectC, respectively. As overviewed in the previous section, future work includes exploring transformers for more AOP frameworks. Additionally, we believe it is necessary to focus on creating tighter aspect compositions for optimizations that are more explicitly coordinated between distinct layers in a system—from the operating system, virtual machine, middleware and application software—along critical execution paths. This work would necessarily include further investigation of how to unify SONAR with heavyweight, low-level tool kits in order to provide an efficient means of truly integrating tasks crossing all layers in the software stack.

Another important avenue to future work investigates a high-level aspect composition language specifically for understanding and explicitly coordinating multiple aspects for multi-level optimization. The problem of scale, in particular with dynamic aspects, is critical in a system like SONAR that enables fine-grained modifications to system infrastructure software. Though SONAR's centralized repository allows developers to scan through all the dynamic aspects applied to the system at any given time, we believe we need to leverage semantic representation of system behaviour, and further target more automated comprehensive management of collections of aspects. For example, this semantic analysis could enable developers to more easily recognize when some combination of optimizations may actually interfere with each other and have conflicts, or to determine when the life-time of an optimization has essentially expired due to changes in the external environment.

Finally, the application of the SONAR model within embedded systems poses a new set of challenges in terms of costs and benefits. Changes in memory footprint or performance characteristics would not be acceptable for many embedded real-time systems. Give that the costs for most constructs in AspectC are minimal, we believe these systems can potentially benefit from SONAR's model,

as they sorely require more attention for increased tool support for application development.

This paper has demonstrated the ways in which SONAR's combined use of AOP, XML, and management tools supports a more fluid and continuous approach to interactive software evolution. We believe this model enables safe and principled system-wide evolution. We have shown how XML can be used to support a language-independent notation for the instrumentation of aspects, how these aspects can be used for optimization and navigation across the software stack, and how services such as JMX provide management and visualization tools coupled with these aspects. Our analysis reveals that the costs of SONAR in a dynamic heavyweight scenario are not to be overlooked. However, it is not unreasonable to assume that some of these costs may in fact be offset by the savings incurred when appropriate optimizations such as prefetching, caching, or operation reordering, can be effectively combined as a system evolves. We believe SONAR provides critical tool support for allowing software developers to better explore these tradeoffs.

References

1. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño Virtual Machine. IBM System Journal 39(1) (February 2000)
2. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: Overview of CaesarJ. Transactions on Aspect-Oriented Software Development I 3880, 135–173 (2006)
3. AspectC. The AspectC Project, `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`
4. AspectJ. The AspectJ Project, `http://www.eclipse.org/aspectj/`
5. AspectWerkz. AOP Benchmark, `http://docs.codehaus.org/display/AW/AOP+Benchmark`
6. AspectWerkz 2, `http://aspectwerkz.codehaus.org/`
7. Chen, M., Kiciman, E., Accardi, A., Fox, A., Brewer, E.: Using Runtime Paths for Macroanalysis. In: Symposium on Hot Operating Systems, HotOS IV (2003)
8. Chen, M., Kiciman, E., Fratkin, E., Brewer, E., Fox, A.: Pinpoint: Problem determination in large, dynamic, Internet services. In: Proc. International Conference on Dependable Systems and Networks (IPDS Track), June 2002, pp. 595–604 (2002)
9. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison Wesley Proffessional, Reading (2002)
10. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In: Joint 8 European Software Engineering Conference (ESEC) and 9 ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp. 89–98 (2001)
11. Continuous Program Optimization, `http://www.research.ibm.com/CPO/`
12. Dmitriev, M.: Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In: Proc. Workshop on Engineering Complex Object-Oriented Systems for Evolution held at OOPSLA (2001)

13. Douence, R., Fritz, T., Loriant, N., Menaud, J., Segura-Devillechaise, M., Sudholt, M.: An expressive aspect language for system applications with Arachne. In: Proc. International Conference on Aspect-Oriented Software Development (2005)
14. DTrace. Big Admin: DTrace, http://www.sun.com/bigadmin/content/dtrace/
15. Duesterwarld, E.: Performance and Environment Monitoring for Continuous Program Optimization. In: Proc. of Invitational Workshop on the Future of Virtual Execution Environments (2004)
16. Engel, M., Freisleben, M.: Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. In: Proc. International Conference on Aspect-Oriented Software Development (AOSD), pp. 51–62 (2005)
17. Extensible Markup Language (XML), http://www.w3.org/XML/
18. FreeBSD. The FreeBSD Operating System, http://www.freebsd.org
19. Gibbs, C., Liu, R., Coady, Y.: Sustainable System Infrastructure and Big Bang Evolution: Can Aspects Keep Pace? In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 241–261. Springer, Heidelberg (2005)
20. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, Springer, Heidelberg (1997)
21. Glassbox, http://www.glassbox.com/glassbox/Home.html
22. K42. K42 Operating System, http://www.research.ibm.com/K42/
23. Lafferty, D., Cahill, V.: Language-Independent Aspect-Oriented Programming. In: Proc. Workshop on Languages and Applications held at OOPSLA (2003)
24. Magpie. Magpie: request tracking for performance analysis, http://research.microsoft.com/projects/magpie/
25. Navarro, L., Südholt, M., Vanderperren, W., Fraine, B.D., Suvée, D.: Explicitly distributed AOP using AWED. In: Proc. International Conference on Aspect-Oriented Software Development (AOSD) (March 2006)
26. NetBeans IDE, http://www.netbeans.org/index.html
27. NetBeans Profiler Project, http://profiler.netbeans.org/index.html
28. Nicoara, A., Alonso, G.: Dynamic AOP with Prose. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, Springer, Heidelberg (2005)
29. Printezis, T., Jones, R.: GCspy: An adaptable heap visualisation framework. In: Proc. Conferences on Object-Oriented Programming, Systems, Languages, and Applications, pp. 343–358 (2002)
30. S.D.N. (SDN). Java Management Extensions (JMX), http://java.sun.com/products/JavaManagement/
31. Spring.NET AOP, http://www.springframework.net/doc/reference/html/aop.html
32. Werner, H.: Across the Frontiers. Ox Bow Press (1990)
33. XSL Transformations (XSLT), http://www.w3.org/TR/xslt

Author Index