Awais Rashid   Mehmet Aksit (Eds.)

Transactions on Aspect-Oriented Software Development II

Springer

Volume Editors

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

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

Library of Congress Control Number: 2006921902

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

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

Editorial

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

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

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

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

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

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

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

Organization

Editorial Board

List of Reviewers

Don Batory
Klaas van den Berg
Lodewijk Bergmans
Gordon S. Blair
Lynne Blair
Johan Brichau
Fei Cao
Walter Cazzolla
Shigeru Chiba
Ruzanna Chitchyan
Siobhán Clarke
Thomas Cleenewerk
Yvonne Coady
Pascal Costanza
Geoff Coulson
Krysztof Czarnecki
Cormac Driver
Eric Eide
Eric Ernst
Tzilla Elrad
Robert France
Marc E. Fiuczynski
Lidia Fuentes
Joerg Kienzle
Micheal Kircher
Julia Lawall
Cristina Lopez
David H. Lorenz
Florence Maraninchi
Jean-Marc Menaud
Mira Mezini
Gilles Muller
Juan Manuel Murillo
Gail Murphy
James Noble
Carlos Noguera
Harold Ossher
Klaus Ostermann
Renaud Pawlak
Monica Pinto
Ragghu Reddy
Martin Robillard
Douglas Schmidt
Christa Schwanninger
Marc Segura-Devillechaise
Ian Sommerville

Table of Contents

On Horizontal Specification Architectures and Their Aspect-Oriented Implementations

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

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

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

1 Introduction

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

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

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

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

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

2 Two Dimensions of Software Architecture

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

2.1 Vertical Units

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

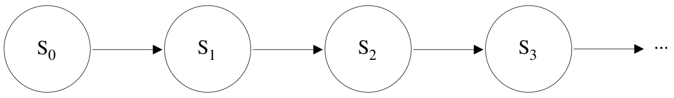


Fig. 1. Execution as a state sequence

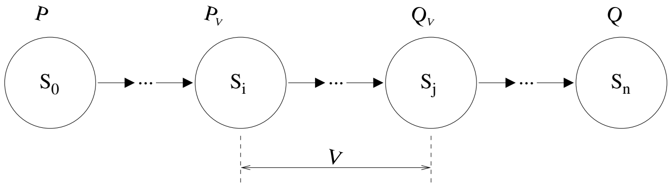


Fig. 2. A vertical slice V in an execution

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

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

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

2.2 Horizontal Units

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

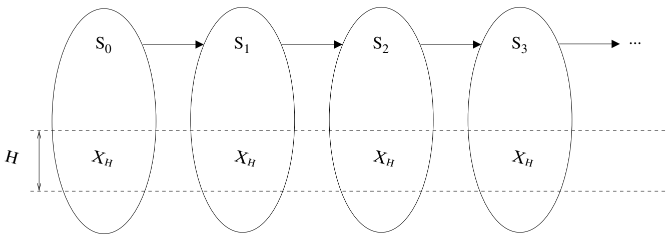


Fig. 3. A horizontal slice H in an execution

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

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

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

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

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

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

2.3 Architecting Horizontal Abstractions

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

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

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

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

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

For comparison, consider how stepwise refinement proceeds with vertical architectural units. In terms of executions, each operational abstraction generates state sequences that lead from an initial state to a final state, so that the required precondition–postcondition pairs are satisfied. At the highest level of abstraction, there may be only one state change, and each refinement step replaces some state changes by state sequences that are generated by more refined architectural units. This leads to a design where the early focus is on interfaces and their use, whereas the "secrets" inside the vertical components may become available only toward the end of the design, when the level of implementable interface operations is achieved.

Due to the above, the abstractions that a vertical architecture provides are not abstractions of the meaning: the complete meaning is assumed available already at the highest level as a precondition–postcondition pair, and it remains the same throughout the design process. Instead, at each level of refinement, a vertical architecture gives an *abstraction of the structure* of an implementable operational model.

3 Aligning Architectural Units in Specification and Implementation

Identification and specification of vertical architectural units have been the primus motor for software architecture research (e.g. [34, 40]). This has led to the adoption of the vertical implementation architecture as the spine of software development, visible in procedural languages, and object orientation in the form of importance of interfaces. The preservation of vertical architectural units in specification and implementation automatically establishes the alignment between the two.

Recent interest in incremental and iterative approaches has resulted in designs where extendibility and flexibility are emphasized, requiring also the option to introduce increments in a horizontal fashion in terms of features, for instance. Building on top of conventional architecture-based practices, software bus architecture, for example, provides enough flexibility for adding features incrementally [38]. However, in that context, the management of different configurations remains a task of the designer, for which no support is provided at the implementation level, apart from methodological one. This methodological support should then be offered by a specification method that guides the designer to focus on features in a stepwise fashion. In a way, the intention is to guide the designer to define coexisting vertical and horizontal architectures in the same component architecture [23].

In contrast to architecture-based approaches, aspect-oriented implementation techniques enable new approaches. In an asymmetric aspect language, for instance, new functions can be added to object-oriented implementations via the process of *weaving* additional operations to existing code. In symmetric languages this corresponds to composition of different "slices". In both cases the augmentation can be based on method call signatures, for instance, which allows the introduction of tangled functions in a centralized fashion. However, several implementation methods have been proposed, and there is no universally accepted way to specify software for an aspect-oriented implementation. In fact, in some cases it seems that aspects are a sophisticated way of solving certain types of problems in an ad hoc fashion. To a certain extent, this can be blamed on our inability to create the types of architectures needed by aspect orientation at an abstract level.

By nature, aspects form horizontal structures, which have not been addressed until recently. Next, some of the related approaches are briefly introduced.

One of the most prominent aspect-oriented design approaches is Theme [5, 8], based on subject-oriented design [9] combined with support for requirements capturing, and management. On the one hand, Theme facilitates finding and managing the design subjects (concerns) of the system. On the other hand, composition patterns define rules for composing (possibly overlapping) design subjects. Also, a mapping to AspectJ and Hyper/J has been investigated [10]. Compared to Theme, we take a more formal approach and promote horizontal architectures as a general-purpose tool for architecting at the specification level. In practice, this enables us to conduct formal proofs and to reason about the relations between different "design subjects".

Problem frames [20] are another representative of horizontal architectures. There, the objective is to decompose problems into smaller subproblems using the horizontal dimension of architecting, whereas we start with an abstract solution and add more details to it during the design.

At design level, horizontal architectures are visible in design fragments [7], which aim at separating individual design decisions at architectural level. This enables revisiting old decisions in favor of newer ones, at least in principle, as advocated in [6]. Approaches where such fragments are used have been based on architectural description languages and informal representation. In our approach, each design fragment would correspond to a horizontal unit (or a set of them), depending on the complexity of the fragment.

Next, we will introduce the DisCo method, which codifies our approach to horizontal architecting. Toward the end of the paper, we then show how horizontal architectures composed at specification level can be aligned with an aspect-oriented implementation.

4 The DisCo Method

The above views have been stimulated by the long-term experiences gained with the DisCo method[2] [22, 49]. DisCo is a formal specification method for reactive system, whose semantics are in TLA [35]. TLA is a linear-time temporal logic where the expressive power of full temporal logic has been reduced to enhance the refinement of specifications. A system meeting its specification and a specification being an implementation of another can be expressed by logical

[2] DisCo is an acronym for Distributed Co-operation.

implication. In the following, we will concentrate on the structural aspects of the DisCo language.

4.1 Horizontal Architectures in DisCo

In DisCo, the horizontal dimension, as discussed above, is used as the primary dimension for modularity. The internal structure of horizontal units consists of partial classes that reflect the vertical dimension. For instance, each of the attributes of a class can be introduced in different horizontal units.

Behavioral modeling is DisCo's bread and butter. The design usually advances so that first the high-level behaviors are included in the model. Based on this abstract behavioral model it is then possible to include more details, even to the level where direct mapping to available implementation techniques becomes an option [33].

In more detail, horizontal components correspond to superposition steps referred to as *layers.*[3] Layers represent behavioral, rather than structural aspects of the system. Thus, they serve in capturing concepts of the problem domain that would otherwise be scattered throughout the vertical architecture. Formally, each layer is a mapping from a more abstract vertical architecture to a more detailed one. As the design decisions are encapsulated inside the layers, they become first-class design elements.

Ideally, each layer contains only those details that pertain to a particular behavioral concept. Thus, better alignment between the requirements and design can be achieved. In the cases where a layered structure is aligned with an aspect-oriented implementation, this results in improved traceability concerning the entire design flow.

Figure 4 illustrates the specification architecture in the simple case of the example outlined above. The ellipses are specifications that correspond to vertical architectures and squares depict layers applied to them. The horizontal architecture consists of three layers. Each layer refines a vertical architecture to a more detailed one by adding a new piece of "meaning" to the system. If different superposition steps are applied to the same vertical architecture, this results in different branches of the same specification. These branches can be composed later in parallel, if needed.

Different concerns can be partially treated in common layers. A situation of this kind arises when a common design decision is made to cater for two different features, for instance. This means that, generally, concerns are treated in one or more layers. Moreover, there can be overlap between the sets of layers treating different concerns, which correspond to different branches having a common root specification. Thus, capturing also nonorthogonal concerns is supported.

4.2 DisCo Primitives

DisCo specifications are built up using layers that consist of classes and *joint actions* operating on instance variables. Formally, classes are considered as sets

[3] Terms *layer* and *superposition step* are used interchangeably in the following.

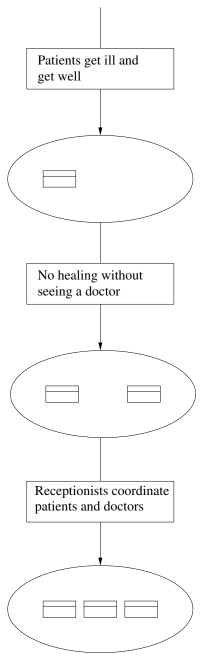

Patients get ill and get well
No healing without seeing a doctor
Receptionists coordinate patients and doctors

Fig. 4. Horizontal architecture consisting of layers

and objects are their elements. Unlike traditional object-oriented methodologies, DisCo omits member functions of objects. Instead, the states of objects are changed by executing multiobject joint actions, which are atomic units of execution. Actions consist of

- a name,
- roles, in which objects can *participate*,
- guard, which must evaluate to *true* for the participating object combination for the action to be executed, and
- a body that may refer only to the participating objects.

Actions model *what* several objects do together hiding the details of communication. An action is said to be *enabled* if such a combination of objects exists that the action can be executed.

A simple DisCo layer `l` shown below consists of class `C`,[4] which has attribute `i` of type integer and assertion `pos` stating that each object of class `C` has a positive value in its attribute `i`. Assertions must be verified to make sure that they are not violated. The specification includes a joint action `swap`, whose roles are `c1` and `c2`. Expression $c1.i >= c2.i$ is the action guard, and the action body

[4] We use capital letters when referring to class names and lower-case letters when referring to objects of those classes or speficications of the same name.

swaps the values of the participants. The double bar in the body means parallel assignment:

```
  layer l is
   class C is
     i: integer;
   end;

   assert pos is ∀ c: C :: c.i > 0;

   action swap(c1, c2: C) is
   when c1.i ≥ c2.i do
     c1.i := c2.i || c2.i := c1.i;
   end;
 end;
```

4.3 Decomposition and Composition

Each layer is allowed to add new data and functionality to an existing specification. Formally, layer `l` in the prior example augments an empty specification denoted as ϵ. The augmentations in DisCo are restricted in such a way that the *safety properties* of the specification being extended cannot be violated. Informally, safety properties can be phrased as "nothing bad ever happens". More formally, safety properties are such properties whose violation can be detected from a finite prefix of the system's behavior expressed as an infinite sequence of states.[5]

The purpose of the restrictions is to guarantee that each layer is, in fact, a strict horizontal component in the resulting system. In particular, a variable is allowed to be assigned to only in the layer introducing it. By this we avoid the problems that would be caused by contradictory assignments to the same variables in different layers. When the specifier introduces a variable, this forces him to think about how and when it can be updated, which promotes concern-oriented thinking. Although this may seem to be too restrictive for the designer, this has proved not to be the case in practice, since assignments can also be introduced in a nondeterministic form that can be refined later.

The layer below extends the specification produced by the earlier layer `l`. It introduces a new class `D` similar to class `l.C` and refines action `swap`, which is renamed as `sum`. The body of the action is augmented with a new assignment:[6]

```
 layer aug is
   import l;
   class D is
     i: integer;
   end;

   refined sum(c1, c2: C; d: D) of swap(c1, c2) is
   when ... do
     ...
```

[5] Possibilities to relax this restriction without compromising the main advantages of the approach have been investigated in [32].

[6] The ellipses in the guard and the body are part of DisCo syntax standing for the original guard and body.

```
      d.i := c1.i + c2.i;
    end;
  end;
```

Composition of different specification branches takes place as follows. If component specifications have a common ancestor in their refinement history, the common parts of the components appear only once. Classes and actions that have a common ancestor are merged. If new subclasses are introduced in the component specifications, merging becomes more complicated, but can still be automatic [32].

The use of atomic actions liberates us from defining the order of executions, thus contributing to the clarity of the composition and superposition. However, this also hardens the alignment of specifications with their implementations when using conventional implementation techniques.

5 From DisCo Specification to Aspect-Oriented Implementation

In this section, we outline the process of mapping DisCo specifications to aspect-oriented implementations. In addition to the generic process, we also introduce some specific practicalities using Hyper/J and AspectJ.

5.1 Generic Approach

DisCo allows specification of reactive behavior at a level of abstraction considerably higher than in programming languages. Thus, whether the implementation is aspect oriented or not, it involves many design decisions. These design decisions are in many cases domain and platform dependent, which makes their codification very hard. The generic parts of the mapping are described in the following.

Computation of a system modeled by a DisCo specification consists of executing actions, each of which is preceded by two phases: Selecting an action to be executed next and binding the participants to the roles. In a DisCo specification, these choices are nondeterministic. However, when implementing the specification the responsibility to initiate an action execution must be assigned to some of the parties. These parties are called *active units*. A design decision of selecting the objects that act as active units must be made at the latest when the system is implemented disregarding the selected implementation technique. In reality, this piece of information is often obvious to the specifier.

Due to the nondeterministic nature of the execution semantics, the implementation involves a scheduler that controls the execution of methods. However, some of the actions interact with the environment of the system, for instance the user. Their initiation should obviously remain nondeterministic. Hence, partitioning between the internal and environment actions is necessary. An interested reader is referred to [32, 33] for more details on the partitioning.

The above design choices do not alter the structure, however. As already discussed, structurally DisCo specifications consist of branches tackling different concerns of the stakeholders, resulting in a symmetric structure in the sense that all the branches introduce similar entities, and no dominant decomposition is required before defining other parts of the system. To preserve the structural alignment, the layer hierarchy should be reflected in the implementation as closely as possible.

The main phases of implementing DisCo specifications are listed below:

1. Assigning the active units. This phase also involves determining the environment actions, which reflect the effect of the environment of the system. In many cases, the active units are participating in such actions.
2. Making the implementation deterministic. Since layers can still include some nondeterministic behavior that does not involve the active units, such parts need to be made deterministic. This means designing local schedulers for the corresponding actions.
3. Creating the implementation level modules that correspond to the layer hierarchy and defining the join points. This is somewhat different depending on whether the implementation technique provides asymmetric or symmetric facilities.

Next, we consider the practical issues when using symmetric and asymmetric implementation languages. For this purpose, Hyper/J and AspectJ are used as examples.

5.2 Symmetric Implementation with Hyper/J

The symmetric approach taken by DisCo resembles Hyperspaces introduced in [39, 50], and in fact the DisCo method can be seen as a realization of Hyperspaces at a specification level, if we consider hyperslices to be formed from superposition steps.

When taking the symmetry of concerns as the starting point for implementing a DisCo specification, one is tempted to use Hyper/J [39], a realization of Hyperspaces in Java, as a concrete vehicle for implementations, as both DisCo and Hyper/J rely on treating concerns in a symmetric fashion. In addition to the issues discussed above, one must consider how the actions are executed once they are composed. In the DisCo model, this is handled by defining the execution of an action as atomic, which is not always true in an implementation-level description. While default treatment in Hyper/J can be satisfactory for a majority of cases, an option to define the order manually must still be offered to programmers.

When using Hyper/J, the layer hierarchy can be mapped almost directly to the implementation. Since the composition rules in DisCo are defined inside layers, and in Hyper/J in separate hypermodules, additional hypermodules are obviously needed. In Hyper/J, the active units do not impose any constraints on the modularity of the implementation. However, they are needed to define the control flow. If the active units are not assigned as discussed above, one could compose a declarative system, where all the functions are present, but the control is missing.

5.3 Asymmetric Implementation with AspectJ

When an asymmetric implementation mechanism like AspectJ [48], for instance, is used for implementing a DisCo specification, the structure of the implementation requires more thorough analysis, although the above issues of initiating executions and the order of operations still remain relevant. The reason for more thorough consideration is that such an implementation is composed by taking an object-oriented implementation as a starting point, and then adding aspects for features whose introduction as aspects is adequate. Obviously, the starting point for composing such an implementation must then be the selection of the right branch of the specification as the baseline object-oriented implementation. In our experiments, we have observed that the branch that defines the active units is often a good candidate for this. Then, all other layers are weaved to this branch using aspects, resulting in the same hierarchy as in the original specification. However, their implementation techniques vary; the concern that introduces active units forms the dominant decomposition of [46], into which additional functionality—even more abstract one—is woven using aspects. Originally, we proposed an approach where intermediate levels of layers were eliminated, if they were no longer relevant from the implementation perspective [1], but currently we feel that also such “intermediate” layers play an important role in documenting how aspects can be weaved to other branches of the specification.

Another difference in the structure of the specification and its implementation is that the design of a specification often begins with abstract concepts, whereas when aiming at an implementation, one is tempted to create something that bears implementation-level significance as such early in the development. This leads us to judge that, while an alignment exists, the incremental methodology used for deriving the specification cannot be easily applied at implementation level. We believe that the value of the described approach lies in being able to address separation of concerns at the specification and design level, and in being able to coordinate where additional functionality may be inserted. This is distinct from lower-level development aspects (tracing, logging, profiling, etc.) that can be dealt with using language facilities at the programming language level.

In more detail, the heuristics we have sketched for implementing DisCo specifications in AspectJ are as follows. For simplicity, we assume that the specification is created only with superposition, i.e. without parallel composition, and the implementation only includes code derived from the specification. Moreover, one of the leaf layers of the layer hierarchy must be identified as the one that includes the active units. This will be referred to as the *active layer*.

The active units are implemented in standard Java and the rest of the layers as aspects in AspectJ. Joint actions described by the active layer are implemented as Java methods in an object-oriented implementation.

The joint actions given in nonleaf layers of the layer hierarchy are refined by superposition when the specification is created. These actions are transformed to aspects, whose join points are tied to the methods in the implementation of the active units. If, for example, action $\mathcal{A}$ from a nonleaf layer has been refined to $\mathcal{A}'$ in an active unit, then the aspect implementation of $\mathcal{A}$ is to be triggered by a method execution in the implementation of $\mathcal{A}'$.

The nonactive leaf layers are implemented as aspects so that their join points are tied to the aspect implementations of the nonleaf layers. Referring to the above example: If $\mathcal{A}$ has been refined to $\mathcal{A}''$ in a nonactive leaf layer, then the join point of the implementation of $\mathcal{A}''$ is tied to the execution of the implementation of $\mathcal{A}$. After this construction, the execution of $\mathcal{A}'$ triggers the execution of $\mathcal{A}$, which, in turn, triggers the execution of $\mathcal{A}''$. This scheme will be illustrated in the sequel by an example.

6 DisCo Specification of a Mobile Switch

The purpose of the following example is to show the types of abstractions that the DisCo approach encourages, together with the types of increments that lead to the final specification. A Hyper/J and an AspectJ implementation of the given specification will be introduced in the next section, together with a discussion on the alignment of the structures of the specification and its implementations.

6.1 Overview

The roots of this example are in research in an industrial setting focusing on specification architectures for a mobile telephone switch [19]. The basic task of such a system is to route "calls" from callers to callees. In addition, a more complicated call model—call forwarding—is included in the behavior. In call forwarding, a call is first routed to some callee, but for some reason (maybe she did not answer the phone) it is routed further to some other callee. The switch comprises also a unit carrying out charging. The unit observes the behavior of the routing and stores information on which the charging can be based.

This section introduces a specification of the switch. The discussion is structured in accordance with the horizontal specification architecture depicted in Fig. 5. The starting point of every DisCo specification is an empty specification ϵ, depicted on the top of the figure. Applying superposition step `legs` results in a specification with the same name. The same applies to steps `charging` and `routing`. Specification `simpleSwitch` at the bottom is a composition of specification branches `charging` and `routing`.

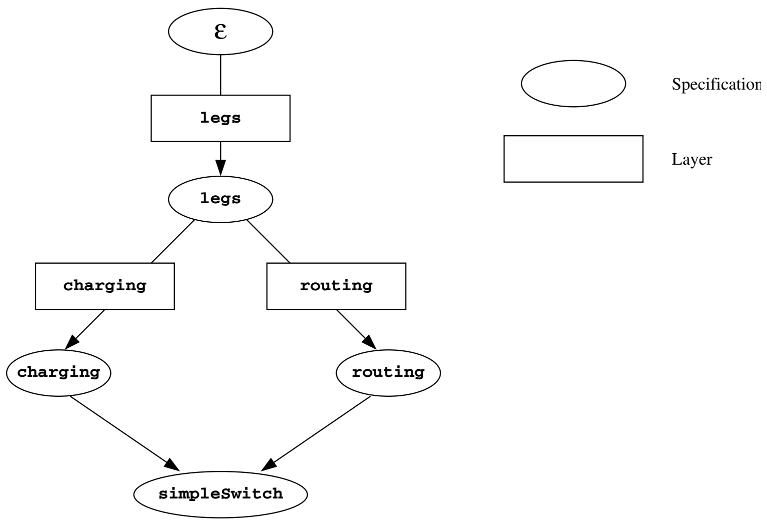


Fig. 5. Specification architecture

6.2 Legs Aspect

Concept *leg* is selected as a fundamental behavioral abstraction of the switch. Informally, a leg is a relation between two mobile terminals modeling that the terminals are connected to each other. In the simplest case, the relation holds between a caller and a callee, whereas more advanced call models like *call forwarding* allow several legs between terminals. Figure 6 depicts a case where terminal `A` has called `B` (`AB-leg`), but the call has been forwarded to terminal `C` (`BC-leg`).

Superposition step `legs` describes the behavior of the switch in terms of classes `Terminal` and `Leg`. The former has no attributes at this level of abstraction. The latter consists of two attributes `a` and `b` that model the two connected parties:

```
class Terminal is end;
class Leg is
  a, b: reference Terminal;
end;
```

Specification `legs` is the common high-level specification for future specification branches. The specification comprises joint actions `connect` and `disconnect`, given in superposition step `legs`, that coordinate the creation and deletion of legs between terminals, respectively. Action `connect` synchronizes two terminals (terminal `a` initiating the call and `b` modeling the terminal the call is being routed to) and a newly created leg `l`. Action `disconnect` simply deletes the participating leg object:

```
action connect(a, b: Terminal; l: new Leg) is
when true do
```

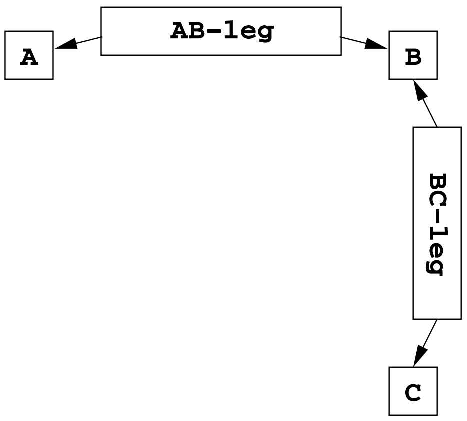


Fig. 6. Example: `A` has called `B` but the call has been forwarded to `C`

```
    l.a := a || l.b := b;
  end;

  action disconnect(l: Leg) is
  when true do
    delete l;
  end;
```

At this level of abstraction the behavior of the system is nondeterministic due to the guards of actions being identically *true*.

6.3 Routing Aspect

Superposition step `routing` defines how the actual connection between terminals is achieved in the switch. In the step, the abstraction `Leg` is implemented by processes `In` and `Out` controlling the connection to incoming and outgoing directions, respectively. When routing, processes are started one after another, and they form a linked list. Figure 7 depicts the result of superimposing the processes on the specification in Fig. 6, and indicates how processes `in1` and `out1` implement `AB-leg`, and `in2` and `out2` implement `BC-leg`.

Superposition step `routing` defines an abstract base class `Process` from which actual processes are derived:

```
  abstract class Process is
    b: reference Terminal;
    next, prev: reference Process;
    implementedLeg: reference Leg;
  end;

  class In is new Process by
    a: reference Terminal;
  end;
  class Out is new Process;
```

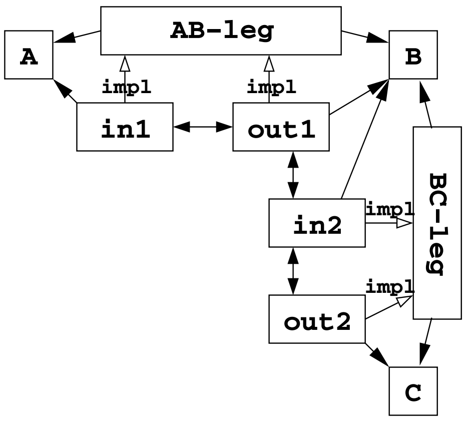


Fig. 7. Refined version of `A` calling `B`, who forwards the call to `C`

Routing a call is modeled with actions `startIn`, `inStartOut`, and `outStartForwarding`. Of these, `inStartOut` is derived from `legs.connect`, and therefore it is combined with refinements of `legs.connect` from other branches of specification. The other two actions perform their obvious tasks, and can only affect variables introduced in the above classes. Therefore, we will only focus on action `inStartOut`. The enabling condition of the action consists of the corresponding parts of the ancestor action plus two conjuncts, which bind the participant in role `a` and ensure that `i` is not yet linked. The modifications the action introduces to the state of the objects are the same as statements of `legs.connect` augmented with assignments to newly introduced variables. This is expressed as follows:

```
refined inStartOut(a: Terminal; i: In; o:new Out; l: new Leg)
of connect(a, i.b, l) is
when ... a = i.a ∧ i.next = null do
  ...
  o.b := i.b || i.next := o ||
  o.implementedLeg := l;
end;
```

In addition to the above actions, superposition step `routing` contains five joint actions (omitted here) to model tearing down a call.

6.4 Charging Aspect

Charging is based on `legs`. When a leg is created charging is started, and when a `leg` is terminated, charging is stopped. Superposition step `charging` does not specify any actual charging policy, but only shows how it could be added to the specification.

Singleton class `Charging` is given to model the unit controlling the billing:

```
class Charging(1) is end;
```

Actions `startCharging` and `stopCharging` are derived from actions `connect` and `disconnect` given in superposition step `legs`, respectively. The only thing the refinements do is the addition of a `Charging` object to synchronize in the actions. For instance, action `startCharging` is given as follows:

```
refined startCharging(a, b: Terminal; l: Leg; c: Charging)
of connect(a, b, l) is
when true do
end;
```

An actual charging policy could be specified for instance by reserving a record for each created leg. The record could be used for storing timestamps indicating the execution moments of the above actions.

6.5 Composition of the Branches Charging and Routing

Specification `simpleSwitch` (the white ellipse in the bottom of Fig. 5) composes two branches by importing specifications `charging` and `routing`. The composite specification is formed by merging extensions to the classes and merging actions that have a common ancestor in their refinement history. For example, actions `routing.inStartOut` and `charging.startCharging` have both refined action `legs.connect`; therefore, they must be merged. The signature of the created composite action *startO&C* contains roles of all component actions. The new enabling condition is the conjunction of enabling conditions of all component actions. Similarly, statements of the action are based on the assignments of the actions being composed. Thus, the unfolded composite action `startO&C` of the example would be given as follows:

```
action startO&C(a: Terminal; i: In; o: new Out; l: new Leg; c: Charging) is
when true ∧
     a = i.a ∧ i.next = null ∧
     true do
  l.a := a || l.b := b ||
  o.b := i.b || i.next := o ||
  o.implementedLeg := l;
end;
```

7 Preserving Alignment in Implementation

In this section, we address implementation of the switch using Hyper/J and AspectJ languages as outlined in Sect. 5. Toward the end of the section, we will also discuss implementations at a general level.

7.1 Hyper/J Implementation

As discussed in Sect. 5, a Hyper/J implementation can be composed using the same underlying structure as the specification. The contents of layers `charging`, `legs`, and `routing` would then give rise to classes corresponding to those given in the specification in a straightforward fashion (Fig. 8). For example, the class structure of the routing implementation corresponds to the class structure given

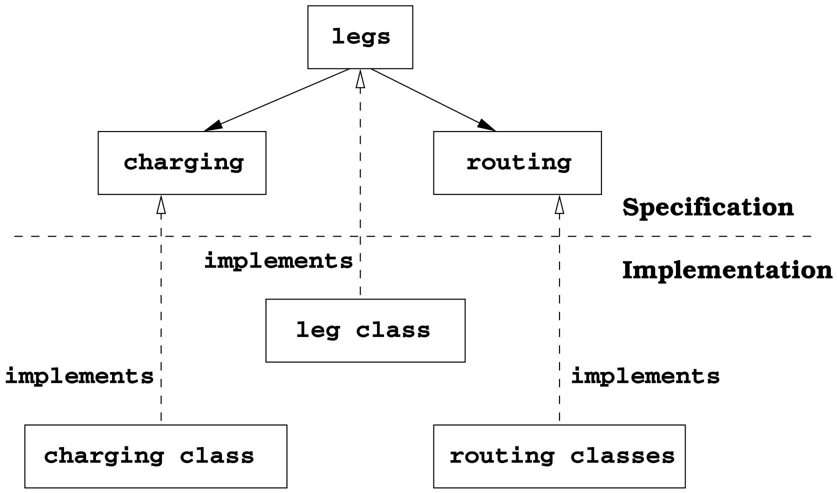


Fig. 8. Implementing superposition steps `legs`, `charging`, and `routing` with Hyper/J

in the specification. Hence, specification-derived classes `Terminal`, `Process`, `InProcess`, and `OutProcess` are introduced.

In the telecommunications domain it is obvious that actions `connect` and `disconnect` are initiated by the environment. When a call is initiated, a new `InProcess` is started that manages signaling toward the caller. When routing reaches terminal B, a new `OutProcess` is born, which implies an execution of the joint action *connect*. If terminal B has set up call forwarding to a terminal C, a new leg should be created as if terminal B had initiated a new call, and the resulting `InProcess` is linked with the proceeding `OutProcess` (see Fig. 7).

Then, a hypermodule is needed that equates the operations that are common in the different branches of the specification (Fig. 9). If necessary, it is also possible to create a declarative combination of `charging` and `legs`. For executions, however, branch `routing` that introduces the actual control is a necessity. Moreover, it would be possible to implement charging-specific functions, provided that the specification would be more detailed in this respect.

In more detail, the given hypermodule should equate operations that have been derived from the same ancestor action. Unfortunately, as object instantiation, parameters, and other implementation details play a crucial role, some parts of the specification do not lend themselves for Hyper/J implementation due to the current tool implementations for practical reasons.

When considering the alignment between the specification and its implementation, the final composite DisCo specification, `simpleSwitch`, can be seen as a counterpart of the hypermodule used in Hyper/J implementation that

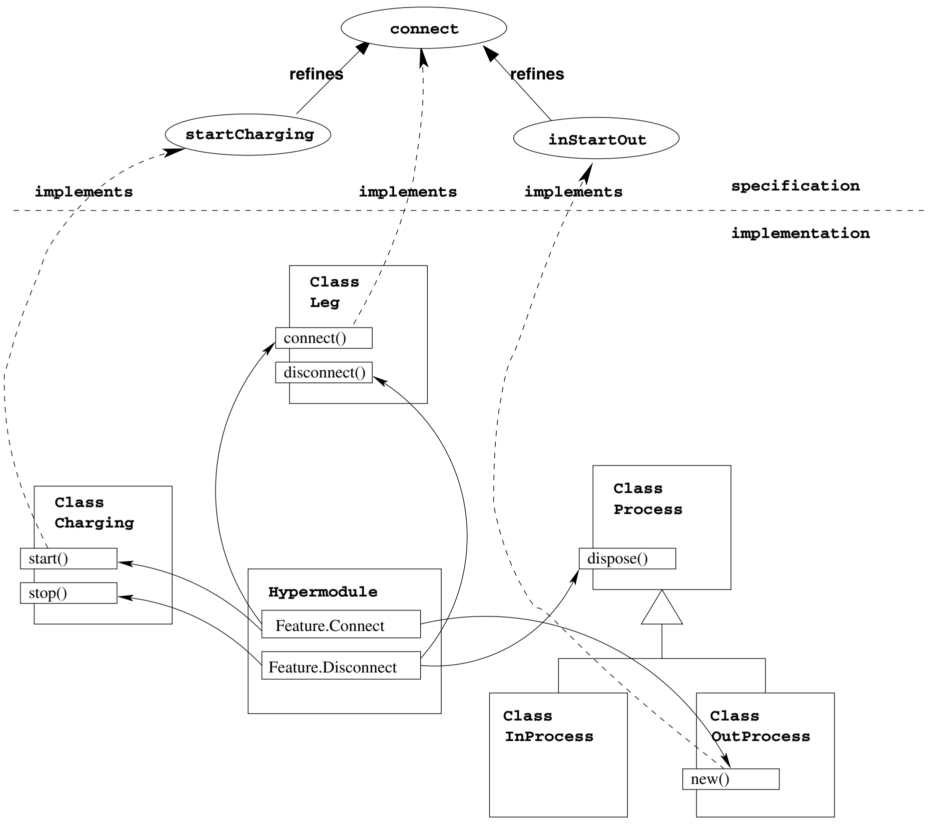


Fig. 9. Hypermodule addressing contracted points

integrates all the concerns given in different branches using the rules defined by DisCo for combinations as the guideline for equations.

7.2 AspectJ Implementation

Similar to the above, layer `routing` was selected as the active layer in AspectJ implementation. The objects introduced in the layer initiate the executions of the actions. `Routing` layer was mapped to Java objects and the other layers to AspectJ aspects as depicted in Fig. 10.

The class structure for `routing` is similar to the one discussed above. Next, we will address other branches of the specification. Aspect implementation of the charging concern is straightforward. Only two advices are needed for starting and stopping charging for a leg. Points where the advices are to be woven into can now be traced from the object implementation to the specification. Actions `connect` and `disconnect` of the superposition step `legs` are implemented with aspect `Leg`, whose pointcuts refer to methods in routing

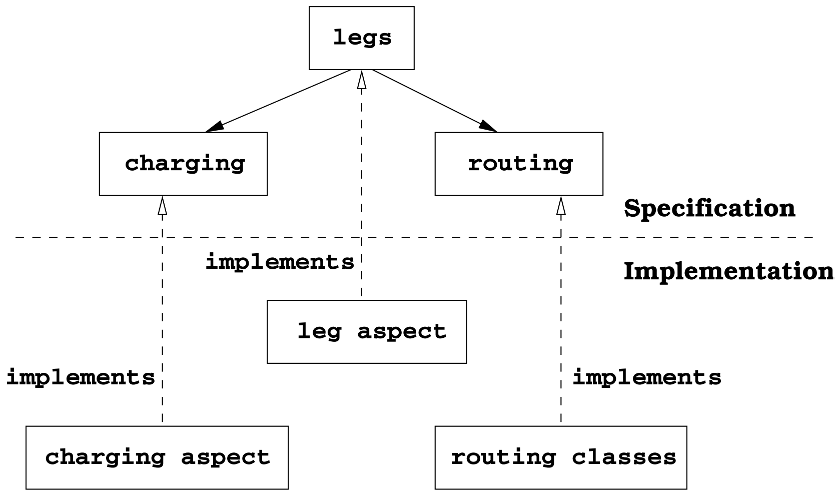


Fig. 10. Implementing superposition steps `legs`, `charging`, and `routing`

classes. Referring to them is justified by the fact that, in our switch, objects of these classes are the only ones that trigger action executions. `Leg` aspect works as an interface contract for the methods triggering the action executions and aspects interested in them. This enables creation of charging without knowledge of how the legs are implemented in routing classes. Instead of making join points in the `Charging` aspect dependent on the implementation of routing object structure, they can now be stated in terms of the `Leg` aspect.

In more detail, methods `connect(..)` and `disconnect(..)` in `Leg`, called by the corresponding advices, are adviced in the `Charging` aspect for starting and stopping charging, respectively. These methods contain no functionality, but they are still invoked as indicated by the high-level design. As a result, all aspects that advice them become independent of changes to the set of join points that make up the pointcuts in `Leg`. This is visualized in Fig. 11.

It should be noted that the selected approach ignores specification `simpleSwitch` when implementing the specification. While a better alignment is gained by mapping the individual layers to the implementation, the composite specification serves in specifying the joint behavior of the total implementation consisting of conventional object code as well as aspects. In this role, it can be used for validation and verification purposes.

7.3 Discussion

The selected mapping approach maintains the independence of different specification branches in implementation in both symmetric and asymmetric cases. Symmetric approach based on Hyper/J relies on a hypermodule that defines the

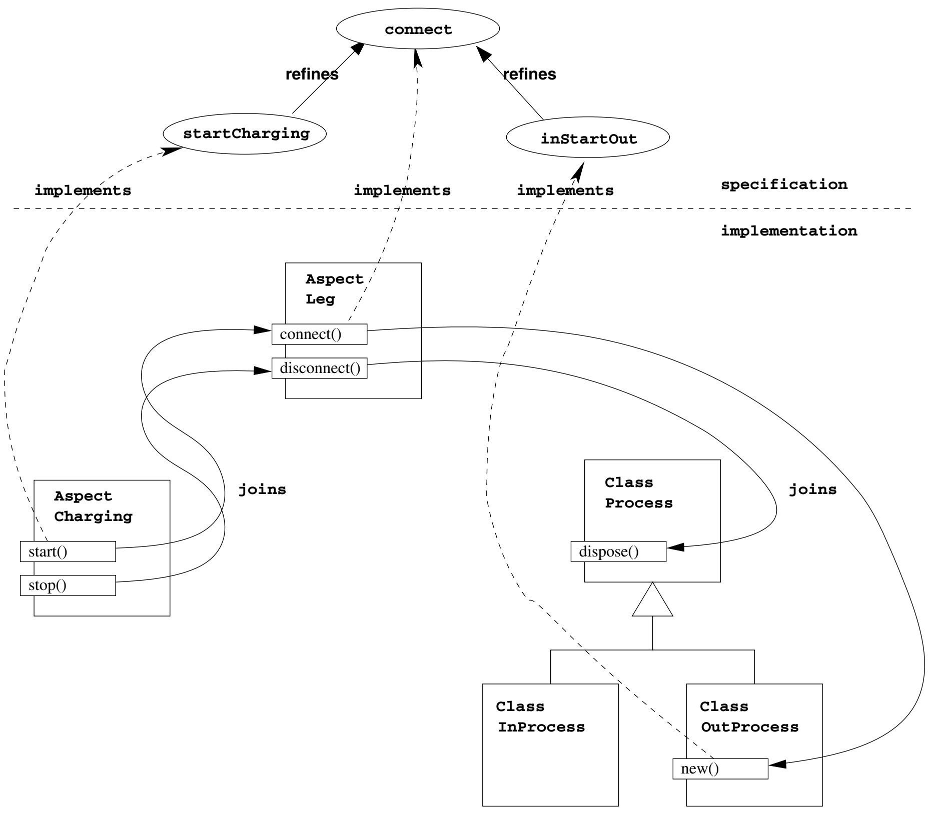


Fig. 11. Aspect joining advices at contracted points

relation between the different branches of the specification in a symmetric fashion. In contrast, asymmetric aspect-based implementations can be linked with object structures of which they need no detailed knowledge.

From the technical point of view, the fundamental task of a mobile switch is routing. Other tasks—like charging—are triggered by the routing module. Therefore, routing gives the actual control flow for the whole system disregarding whether its selection as the dominant decomposition is implicit or explicit. In Hyper/J, when the branch that defines the control flow of the system is not included, the role of the hypermodule becomes declarative, and in AspectJ, join points serve a more declarative than definitive purpose in our approach in any case.

When comparing this implementation to a straightforward one, where a baseline implementation (branch `routing`) would be given first and then extended with aspects needed for charging, the main difference lies in abstraction leg. In the Hyper/J implementation, this abstraction can be used to provide a high-level view to the system, as well as as a basis for attaching new features to it with new hypermodules. In contrast, in the AspectJ implementation, leg abstraction provides a join point for hooking into abstract behaviors, whose concrete realizations are only known by legs. Therefore, the value of the abstraction as a specification-level join point collection is evident: Maintainability is improved, as multiple pointcut declarations do not need to be updated to reflect such changes.

In general, the use of such abstractions that do not have a straightforward counterpart in a conventional implementation has emerged as a pattern of reducing complexity in the DisCo approach. In an aspect-oriented implementation, this appears as equation of multiple operations, potentially including declarative rather than executable definitions (Hyper/J) and as aspects woven to other aspects (AspectJ), i.e., one aspect represents the abstraction and other aspects add more features to it.

8 Related Work

Recent years have witnessed emergence of a number of new approaches to aspect-oriented capturing of requirements, design, and architecting. These approaches, some of which have already been discussed above, are too numerous to be addressed here. However, a relation between our work and the most important and closely matching ones will be drawn below. An interested reader is also referred to recently published extensive surveys in [4, 11].

For distributed systems, a different kind of architectural duality results from the idea of communication-closed layers [14]. By weakening sequential composition to *layer composition* ($\bullet$), in which temporal precedence of the included actions is weakened into logical precedence, the situation is achieved that parallel composition ($\|$) of the form

$$(S_0 \bullet S_1) \parallel (T_0 \bullet T_1)$$

can also be viewed as layer composition of simpler parallel structures $S_i \parallel T_i$,

$$(S_0 \parallel T_0) \bullet (S_1 \parallel T_1).$$

For a design method this duality means that a distributed system can be specified and designed in terms of the sequentially layered view with layers $S_i \parallel T_i$, although the implementation structure will follow the parallel view with more complex processes [21]. Although specification modules (the layered view) then cut across implementation processes (the parallel view), they do not exhibit horizontal modularity in the sense discussed above. As for design methodology, communication-closed layers are, in fact, specified similarly to vertical modules, using pre- and postconditions (of the total system). In other words, the semantics of communication-closed layers are expressed in terms of algorithmic, not reactive meanings, with the objective of using traditional formally based methods also in the presence of concurrency.

Katz et al. [27, 28, 44] have observed the close connection between superposition and crosscutting modularity. In [45] the authors use AspectJ for implementing superposition-based specifications. A framework for combining superposition steps using two operations is proposed. Firstly, superposition steps can be combined sequentially, like in our approach. Secondly, superposition steps can be merged, which roughly corresponds to parallel composition of the different branches of specification architecture in DisCo. The biggest difference is that our approach suggests a modeling language at a more abstract level, supported by tools [2]. Furthermore, we also utilize a more restricted version of superposition to guarantee the preservation of safety properties by construction, thus relying less on the designer's ability to conduct formal proofs.

Other specification methods supporting aspect orientation, such as Theme [5, 8], are usually defined as extensions to UML [43]. In many respects, however, issues of horizontal architecture have been neglected at the specification level. This is also true for QCCS (quality controlled component-based software development) [47] where the focus is on nonfunctional properties, especially on quality of service. Similar to DisCo, the basic problem is how to implement crosscutting designs, in this case including reusable components and contracts.

Another solution based on UML, suggested by Kandé [24], provides a concern-oriented view of software architecture supporting encapsulation of concerns. For this purpose, the approach defines a framework called perspectival concern-spaces, which can be seen as an instantiation of the Hyperspace model in UML with emphasis on architecture. Manual translation to AspectJ is supported as well as mapping to an architectural description language.

One could also see similarities between the Rational Unified Process (RUP) [52] and our approach. On the one hand, both are incremental and architecture centric. In addition, the DisCo layers could be seen to correspond to use cases promoted by RUP. On the other hand, in RUP, the use cases and the architecture are described by different artifacts, whereas in DisCo the vertical architecture emerges incrementally in refinement.

Model-driven architecture (MDA) [16, 51] by Object Management Group (OMG) also includes elements similar to the DisCo approach. In MDA, however, only three predefined levels are used, including *computation-independent*, *platform-independent*, and *platform-specific* models. Furthermore, like in our approach, higher levels of abstraction can be considered as contracts or design restrictions for lower-level implementations. By allowing the individual levels of abstraction of MDA to consist of several DisCo layers, one can create a development approach that fits in the guidelines of MDA without compromising rigorousness [42]. Similar results can also be achieved using tool support for managing specialization of design patterns [37].

9 Conclusions

The background of this work is in earlier research related to the formal method DisCo, which has led us to consider the different dimensions of software architecting. This research, initiated by the ability to express design patterns as syntactic and semantic units of systems exceeding the limits of individual objects [36], has resulted in encouraging results on how to structure a specification in a fashion where the specifier would be able to tackle one concern at a time. Such an approach, based on a solid theory [32], reflects the architecture of different concerns rather than the conventional module structure. So far, it has already allowed us to ease software evolution [3], use predefined design steps for reusing architectural design decisions [29], and provided an option to animate even partial system specification early in the development [2]. Some ideas have also been embedded in pattern-oriented tool support for UML specifications, where patterns form the horizontal architecture [17].

In this paper, based on our earlier work, we have discussed vertical and horizontal architectures and their use for constructing reactive systems. These dimensions are in some sense dual, and they are incompatible with each other in the sense that it does not seem useful to combine them in a single system of modules. Therefore, the horizontal dimension, which is currently visible in design patterns and aspect-oriented programming, for instance, remains as a crosscutting dimension with respect to the conventional vertical units of implementation.

However, the two dimensions can be related in different ways. As discussed, the DisCo approach uses the horizontal dimension as the primary dimension, and provides an incremental specification method for composing specifications. Vertical implementation architecture is anticipated in terms of a high-level view of objects and their cooperation. In contrast, most aspect-oriented approaches, in particular those influenced by AspectJ, have taken a more pragmatic and asymmetric approach. There, the primary architectural dimension is based on conventional vertical modularity, and crosscutting aspects are added to it by an auxiliary mechanism for horizontal modularity. Still, as demonstrated in the example in this paper, even when using an implementation mechanism that allows symmetric representation of concerns, like Hyper/J, the special role of a subsystem providing control flow is evident at the level of programming, because otherwise only declarative contribution can be given. For more abstract representation, other mechanisms that offer improved modeling facilities are therefore needed.

Since horizontal structures provide abstractions of the total system, their explicit use seems natural in a structured approach to specification and in incremental design of systems. At the programming language level it is, however, difficult to develop general-purpose support for horizontal architectures. This means that a well-designed horizontal structure may be lost in an implementation, or entangled in a vertical architecture. To avoid this, we have discussed a mapping that implements DisCo specifications in Hyper/J and AspectJ achieving alignment between a symmetric specification and its aspect-oriented implementation using both symmetric and asymmetric techniques.

Considering future work, a more rigorous mapping would enable automation, needed in practical applications, but this has not been carried out yet. As discussed above, the mapping includes nonobvious design choices that are domain and platform dependent. However, it seems quite obvious that we could provide, at least in some limited domain, basic tool support for the mapping similar to UML tools generating header files and stubs for the methods. For each such domain, we would need to address additional constraints. Moreover, considerations on the importance of declarative properties in pragmatic implementation work should be tested in practice.

Acknowledgments

The authors would like to thank the other members of the DisCo team and the anonymous reviewers for their helpful comments.

References

[1] T. Aaltonen, J. Helin, M. Katara, P. Kellomäki, and T. Mikkonen. Coordinating objects and aspects. *Electronic Notes in Theoretical Computer Science*, 68(3): 248–267, 2003

[2] T. Aaltonen, M. Katara, and R. Pitkänen. DisCo toolset—the new generation. *Journal of Universal Computer Science*, 7(1):3–18, 2001. `http://www.jucs.org`. Cited September 2006

[3] T. Aaltonen and T. Mikkonen. Managing software evolution with a formalized abstraction hierarchy. In: *Proceedings of the Eight IEEE International Conference on Engineering of Complex Computer Systems*, Greenbelt, MD, USA, IEEE Computer Society, Piscataway, pp. 224–231, December 2002

[4] AOSD-Europe. Survey of aspect-oriented analysis and design approaches. `http://www.aosd-europe.net`. Cited September 2006

[5] E. Baniassad and S. Clarke. Theme: an approach for aspect-oriented analysis and design. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, IEEE Computer Society, Piscataway, pp. 158–167, 2004

[6] J. Bosch. Software architecture: the next step. In: F. Oquendo, B. Warboys, and R. Morrison (eds.) *Software Architecture—Proceedings of the First European Workshop on Software Architecture*, Number 3047 in Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York, pp. 194–199, 2004

[7] J.L. Boyd and G.M. Karam. Using design fragments to modularize the design of communications in concurrent systems. In: *7th International Workshop on Software Specification and Design*, IEEE Computer Society, Piscataway, pp. 184–193, 1993

[8] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 44(1):71–100, 2002

[9] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, 1999

[10] S. Clarke and R.J. Walker. Separating crosscutting concerns across the lifecycle: from composition patterns to AspectJ and Hyper/J. Technical Report TCD-CS-2001-15, Department of Computer Science, Trinity College, Dublin, 2001

[11] C.E. Cuesta, M. del Pilar Romay, P. de la Fuente, and M. Barrio-Solórzano. Architectural aspects of architectural aspects. In: R. Morrison and F. Oquendo (eds.) *Software Architecture—Proceedings of the 2nd European Workshop on Software Architecture*, Number 3527 in Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York, pp. 247–262, 2005

[12] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976

[13] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(4):1–4, 1980
[14] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155–173, 1982
[15] R.E. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development*. Addison–Wesley, Reading, 2004
[16] Object Management Group. MDA guide version 1.0.1. OMG Document Number omg/2003-06-01. `http://www.omg.org/mda/specs.htm`. Cited September 2006
[17] I. Hammouda, J. Koskinen, M. Pussinen, M. Katara, and T. Mikkonen. Adaptable concern-based framework specialization in UML. In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, IEEE Computer Society, Linz, Austria, pp. 78–87, September 2004
[18] W. Harrison, H. Ossher, and P. Tarr. Asymmerically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Thomas J. Watson Research Center, December 2002
[19] S. Isojärvi. DisCo and Nokia: experiences of DisCo with modeling real-time system in multiprocessor environment. *FMEIndSem'97*, Otaniemi, Finland, February 20, 1997
[20] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison–Wesley, Reading, 2001
[21] W. Janssen and J. Zwiers. From sequential layers to distributed processes. In: *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, ACM, New York, pp. 215–227, 1992
[22] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In: *Proceedings of the 12th International Conference on Software Engineering (ICSE'90)*, IEEE Computer Society, Piscataway, pp. 63–71, 1990
[23] J. Jokinen, H.-M. Järvinen, and T. Mikkonen. Incremental introduction of behaviors with static software architecture. *Computer Standards and Interfaces*, 25(3):215–222, 2003
[24] M.M. Kandé. *A Concern-Oriented Approach to Software Architecture*. PhD Thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2003
[25] M. Katara and S. Katz. Architectural views of aspects. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, MA, USA, ACM, New York, pp. 1–10, March 2003
[26] M. Katara, R. Kurki-Suonio, and T. Mikkonen. On the horizontal dimension of software architecture in formal specifications of reactive systems. In: C. Clifton, R. Lämmel, and G.T. Leavens (eds.) *FOAL 2004 Proceedings, Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, Lancaster, UK, Technical Report TR #04-04 Department of Computer Science, Iowa State University, Iowa, pp. 37–43, March 2004
[27] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, 1993
[28] S. Katz and J. Gil. Aspects and superimpositions. Position paper in ECOOP 1999 workshop on Aspect-Oriented Programming, Lisbon, Portugal, June 1999
[29] P. Kellomäki and T. Mikkonen. Design templates for collective behavior. In: E. Bertino (ed.) *Proceedings of ECOOP 2000, 14th European Conference on Object-Oriented Programming*, Number 1850 in Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York, pp. 277–295, 2000

[30] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In: M. Akşit and S. Matsuoka (eds.) *ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Vol. 1241* Lecture Notes in Computer Science Springer, Berlin Heidelberg New York, pp. 220–242, June 1997

[31] R. Kurki-Suonio. Modular modeling of temporal behaviors. In: S. Ohsuga, H. Kangassalo, H. Jaakkola, K. Hori, and N. Yonezaki (eds.) *Information Modelling and Knowledge Bases III*, IOS Press, Amsterdam, pp. 283–300, 1992

[32] R. Kurki-Suonio. *A Practical Theory of Reactive Systems—Incremental Modeling of Dynamic Behaviors.* Springer, Berlin Heidelberg New York, 2005

[33] R. Kurki-Suonio and T. Mikkonen. Abstractions of distributed cooperation, their refinement and implementation. In: B. Krämer, N. Uchihira, P. Croll, and S. Russo (eds.) *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, IEEE Computer Society, Piscataway, pp. 94–102, April 1998

[34] P. Laine. The role of SW architecture in solving fundamental problems in object-oriented development of large embedded SW systems. In: *Working IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society, Piscataway, pp. 14–23, 2001

[35] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994

[36] T. Mikkonen. Formalizing design patterns. In: B. Werner (ed.) *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, IEEE Computer Society, Piscataway, pp. 115–124, April 1998

[37] T. Mikkonen, R. Pitkänen, and M. Pussinen. On the role of architectural style in model-driven development. In: F. Oquendo, B. Warboys, and R. Morrison (eds.) *Software Architecture—Proceedings of the First European Workshop on Software Architecture*, Number 3047 in Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York, pp. 74–87, 2004

[38] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. *ACM Operating Systems Review*, 27:58–68, 1993

[39] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. In: M. Akşit (ed.) *Software Architectures and Component Technology.* Kluwer Academic, Dordrecht, 2001

[40] D.L. Parnas. On the criteria to be used in decomposing systems in modules. *Communications of the ACM*, 15(12):1053–1058, 1972

[41] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, 1985

[42] R. Pitkänen. A specification-driven approach for development of enterprise systems. In: K. Koskimies, J. Lilius, I. Porres, and K. Østerbye (eds.) *Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques, Vol. 34* TUCS General Publication, Turku Centre for Computer Science, Turku, pp. 74–87, August 2004

[43] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison–Wesley, Reading, 1998

[44] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, ACM, New York, pp. 28–40, April 2002

[45] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, 2003

[46] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. *N* degrees of separation: multi-dimensional separation of concerns. In: D. Garlan (ed.) *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, USA, ACM, New York, pp. 107–119, May 1999

[47] T. Weis, N. Plouzeau, G. Amoros, P. Donth, K. Geihs, J.-M. Jézéquel, and A.-M. Sassen. *Business Component-Based Software Engineering*, Chapter QCCS: Quality Controlled Component-based Software development. Kluwer, Dordrecht, 2002

[48] AspectJ. `http://aspectj.org`. Cited September 2006

[49] DisCo. `http://disco.cs.tut.fi`. Cited September 2006

[50] Hyperspace. `http://www.research.ibm.com/hyperspace/`. Cited September 2006

[51] MDA. `http://www.omg.org/mda/`. Cited September 2006

[52] Rational Unified Process. `http://www-306.ibm.com/software/awdtools/rup/`. Cited September 2006

A Framework for Policy Driven Auto-adaptive Systems Using Dynamic Framed Aspects

Phil Greenwood and Lynne Blair

Computing Department, InfoLab21, South Drive,
Lancaster University, Lancaster, UK, LA1 4WA
{greenwop, lb}@comp.lancs.ac.uk

Abstract. This paper describes and evaluates a framework that allows adaptive behaviour to be applied to systems by using a combination of dynamic Aspect-Oriented Programming (AOP), parameterisation and policies. Our approach allows the operator to create policies to define adaptive behaviour based on Event-Condition-Action rules. The combination of dynamic AOP with parameterisation aids reuse and allows aspects to be generated to suit the current system conditions; these aspects can then be woven at run time to adapt the application behaviour. This approach is evaluated in two ways; firstly performance measurements are presented to show that such behaviour does not add a substantial overhead to the target system. Secondly, Aspect-Oriented software metrics are applied to the adaptations applied to illustrate their reusability and flexibility.

Keywords: dynamic AOP, frames, dynamic adaptation, auto-adaptive, parameterisation, policies, aspect interaction.

1 Introduction

Many modern applications are required to operate in environments that are susceptible to change. There are numerous properties that could vary during an application's run time, from the available hardware resources to the user requirements. For a system to continue operating optimally, it is vital that the system adapts to suit the current conditions. Aspect-Oriented Programming (AOP) [1] is now a common and recognised approach to implementing system adaptability.

A traditional approach to system adaptation is to stop the target system, apply the changes, and then restart the target system [2]. Critically, any stopping of the system is undesirable for domains that require high levels of availability. More recently, focus has shifted to dynamically adaptable systems. These types of systems [3–7] do not require the target system to be taken off-line for adaptations to be made, thus maintaining high levels of availability.

These dynamically adaptable systems can be split into two categories. The first category of such systems rely on human system operators to decide when and what adaptations should take place; we shall call these *manually-adaptive* systems.

The second category of dynamically adaptable systems are able to apply adaptations automatically, typically by monitoring the target system and selecting the most appropriate adaptation when it is required. This category is what we shall call *auto-adaptive* systems. There are a variety of systems that achieve this in different ways [3–5, 7].

Several benefits can be gained from implementing an auto-adaptive system:

– the system is able to react much quicker to changes in the environment;
– the chances of errors being introduced to the system are lowered due to reduced human contact with the system;
– improved levels of availability.

The majority of previous research has focused on the use of component-based development to implement adaptive systems [8]. The loosely coupled nature of components eases their dynamic replacement and makes them a suitable choice for adaptive systems. However, one fundamental problem with component-based programming is that crosscutting concerns cannot be easily encapsulated; this reduces maintainability and causes large disruption when a concern spread over a number of components requires adapting.

We propose that dynamic AOP (dAOP) [9] is an ideal technique to prevent these problems and provides suitable mechanisms for developing adaptive systems. dAOP has previously been used to adapt system behaviour at run time [3, 4, 7]; we aim to exploit this technique further. Our motivation is to improve the reusability and flexibility of the adaptations and apply them automatically without incurring a substantial overhead to the target system. Improving the reusability will have a variety of benefits associated with the development of adaptations, such as reducing the implementation effort. Additionally, by increasing the flexibility of the adaptations it is possible to customise them to the state of the target system at the time they are applied. This allows the target system to be adapted in a more suitable and precise manner. We hypothesize that by combining policies [5, 10], parameterisation techniques [11] and reflection [12] with dAOP we can achieve these aims.

Our goals in this work are to develop a flexible framework capable of adding auto-adaptive behaviour retrospectively to existing systems. More specifically this framework should:

– require no change to the underlying target system;
– be able to extend/modify adaptations;
– create flexible and reusable adaptations;
– ensure correctness and compatibility between the aspects and the system;
– not add a substantial overhead to the target systems performance.

The evaluation presented will show that these aims have been met without incurring a substantial overhead. Performance measurements are presented to illustrate the overhead from performing a variety of operations to implement this auto-adaptive behaviour. To illustrate that the adaptations implemented are both reusable and flexible, AO software metrics are applied to measure these properties.

The remaining structure of this paper is as follows. Section 2 gives an introduction to auto-adaptive systems and the background of the main technologies being used in this work: policies, Framed Aspects, dAOP and reflection. Section 3 then details the implementation used and Sect. 4 evaluates this work by giving a series of performance measurements and evaluates the reusability/flexibility of the adaptations used. Our implementation is compared to a number of related systems in Sect. 5. Section 6 then describes potential future work. Finally, Sect. 7 concludes this paper by summarising its key findings.

2 Background

This section will describe the background of the various technologies used in this work, including a brief description of existing adaptive systems.

2.1 Adaptive Systems

There are several examples of adaptive systems implemented in different ways to meet different goals, such as component based solutions [13, 14], middleware implementations [15], or AOP based solutions [3, 7, 16]. One similarity they all posses is the ability to alter their behaviour dynamically to suit the current context of use.

Adaptive component-based middleware systems [17] are currently a very active research area. A component is a module of code that is well defined with a set of required and provided interfaces [18]. Typically this style of adaptive system ensures that components are independent of each other, i.e., that components are loosely coupled. This makes it easy to dynamically replace unsuitable components with components which are more suitable.

However, these component-based solutions do posses certain inherent limitations. Firstly, such solutions cannot capture crosscutting concerns. If a particular type of behaviour needs altering and a number of components co-operate to implement this, each component involved has to be modified and then replaced. This is obviously inefficient, reduces the maintainability of the system and can possibly cause great disturbance in the system. Another restriction is that component-based solutions only allow coarse-grained changes, in that the finest granularity of change is the component level. They do not allow changes to elements contained within the component such as methods, attributes, etc. Again this could result in inefficient changes since whole components must be replaced to make a small change.

Implementing the framework using an AOP based solution will remove these restrictions. A finer-granularity of change can be applied to each component of the system. In addition to replacing a whole component, the behaviour of a single method or an attribute can also be modified; allowing much more precise and efficient changes.

Using AOP will allow the structure of the adaptations to be kept at run time and so ease the process of applying and later removing adaptations. Additionally, the joinpoint model of AOP techniques provides us with a natural mechanism to define adaptation points in the system. Several dynamically adaptable systems have already been implemented using AOP; Sect. 5 will compare and contrast these approaches with our framework.

AOP can be used to implement an auto-adaptive system in two contrasting ways. Either *the required changes* can be encapsulated or *the adaptation process* can be encapsulated. Examples of this latter method are described in [19] and [20]. These implementations allow the programmer to create the application logic with little thought required on how to make the system adaptable as long as good software design practices are followed. The adaptation logic is encapsulated inside an aspect and woven to the base-system to make it adaptable. These aspects typically contain hooks to allow adaptations to be added at run time. Auto-adaptive systems that employ this approach are generally implemented using a static AOP implementation such as AspectJ [21]. This limits their overall dynamicity as changes to the adaptation process or the addition of new points of adaptation are not possible without stopping the system. Our approach follows the former approach, where the required adaptations are encapsulated inside *dynamic* aspects. This approach provides greater flexibility and achieves high levels of adaptability, whilst still achieving good levels of separation of concerns.

2.2 Policies

To achieve *auto*-adaptation, a method to define the conditions when changes should take place is needed. Policy rules are widely used to implement this type of behaviour, a policy rule being defined as a rule governing the choices in behaviour of a managed system [22]. Event-Condition-Action (ECA) rules are a commonly used technique, often encountered in active-databases [23]. ECA rules are so named as they are processed when a defined *event* occurs. When this event is triggered the *condition* specified is evaluated. If the condition is true then the defined *action* is executed. This format fits well with the structure of our framework and the ECA model can be easily extended to be used with dAOP.

These policies can also be used to define relationships that may exist between the adaptations. Defining these relationships within the policies keeps all information regarding system behaviour in one location. If not specified, certain relationships could cause undesirable interactions to occur between aspects. This is especially relevant in auto-adaptive systems where the aspects are woven automatically. A set of examples are given in Sect. 3.6 to illustrate such interactions and how our policy specification can prevent them.

Formal analysis of aspect interaction is an area that has not been widely addressed in the AOSD community [24]. One key exception is the work of Douence et al. [25] which involves the formal identification of aspects that interact at the same joinpoints and then the resolution of the interaction problems identified. Our approach aims to address the occurrence of such interactions solely at an implementation level.

2.3 Framed Aspects

Framed Aspects are the amalgamation of Frame technology [11] with AOP. Frame technology was originally conceived in the late 1970s to provide a mechanism for creating reusable segments of code using a combination of metavariables, code templates, conditional compilation and parameterisation. A developer initially creates a specification from which the "framed" code is automatically processed to generate concrete customised code. Using a generative process in this way allows code to be reused in many different systems and contexts making frames ideal for the creation of code libraries or software product lines.

Frames alone cannot cleanly encapsulate crosscutting concerns. Any evolution of the software may require changes to be made across a number of modules and could cause architectural erosion [26]. Frames also require that the variation points are explicitly specified in the code; this can cause difficulties reading and understanding the code. By combining Frames and AOP, system features which are crosscutting, can be encapsulated within a single module.

Framed Aspects [27] is one such approach to amalgamate Frame technology with AOP. The Framed Aspects implementation is a customised version of *XML-based Variant Configuration Language* (XVCL) [28] for use with AOP which allows features to be composed together in a noninvasive manner. In Sect. 4, Framed Aspects and XVCL are compared using a quantitative metric to illustrate the differences between the techniques. Figure 1 shows the modules required to implement Framed Aspects.

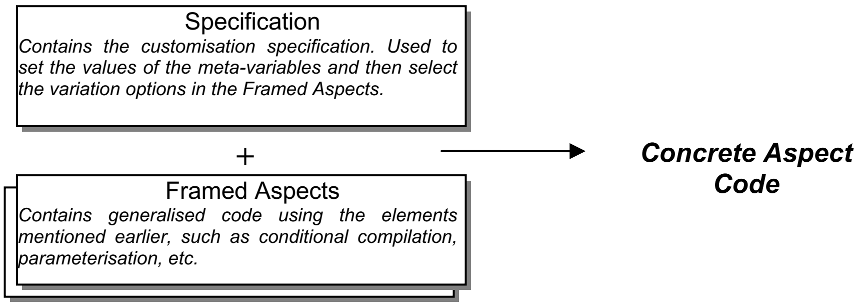

Specification
Contains the customisation specification. Used to set the values of the meta-variables and then select the variation options in the Framed Aspects.
+
Framed Aspects
Contains generalised code using the elements mentioned earlier, such as conditional compilation, parameterisation, etc.
Concrete Aspect Code

Fig. 1. Framed aspect composition and the required modules

For aspects to be successfully parameterised and generalised, variation points within the aspect need to be identified using an appropriate Frame construct. [27] explains the Framed Aspect process and methodology in more detail and contains an example to show how a simple cache aspect can be generalised using frames, in order to make it reusable. In Sect. 3.5 of this paper a variation of this cache example is used to show how Framed Aspects can be used in an auto-adaptive environment.

Framed Aspects were chosen for their power and lack of constraints regarding what elements can be parameterised. This is an extremely valuable and flexible property when applying the Framed Aspect approach to adaptations. In our proposed framework the Framed Aspect specification (see Fig. 1) is generated automatically at run time based on the current state of the target system. This allows the generated adaptations to be customised to the current needs of the target system.

2.4 Dynamic AOP

Dynamic AOP is a branch of AOP that allows aspects to be woven dynamically to the base-code of a system at load-time or run time. This eases tasks, such as testing, patching systems and adapting system behaviour. To clearly illustrate the benefits of dynamic AOP, a comparison with static AOP follows.

Static AOP typically weaves the aspects at compile time. This is less flexible than dAOP in that the aspects and pointcuts need to be predetermined and defined before the application is deployed. However, static AOP allows much more efficient weaving and advice execution. For example, the overhead for executing AspectJ advice is almost identical to a standard Java method-call [29]. In contrast, when a new aspect is woven dynamically using AspectWerkz [30] a significant overhead is experienced and advice execution time is higher due to the reliance on reflection [31]. This is clearly a trade-off between flexibility and performance. Our focus on auto-adaptive systems and the benefits of a flexible framework led us to pursue dAOP techniques. The effects on performance will be discussed in Sect. 4.

The potential problem of undesirable interactions between aspects is more prominent in dAOP when compared to static AOP. When using static AOP, it is more likely that any issues can be identified earlier and problems can be prevented. Some AOP techniques allow relationships and aspect priorities to be defined, such as the precedence rules used in AspectJ [21]. However, we have found that the majority of the rules are not ideal or explicit enough to prevent some of the interaction issues that can occur in an auto-adaptive system. To resolve this we have implemented our own relationship definition technique that will be described in Sect. 3.

A number of different dAOP techniques exist which could have formed the basis for our framework, such as AspectWerkz [30], JAC [32], JBOSS AOP [33] or Prose [9] (see [34] for a comprehensive comparison of these techniques). We selected AspectWerkz due to a number of factors:

– AspectWerkz is a general purpose technique which places no restrictions on the type of system it can be used with;
– it has an extensive joinpoint model, important when implementing an adaptive system since this means fewer restrictions on the points in the system where adaptations can be applied;
– aspects in AspectWerkz are standard Java classes, which eases the creation of the aspect code as programmers will not have to learn a new language;
– AspectWerkz not only allows new aspects to be introduced at run time but also allows new pointcuts to be added to the target system—an important feature for implementing unanticipated adaptations;
– it maintains aspect structure, which eases the removal and querying of woven aspects.

As mentioned above, aspects used within AspectWerkz are implemented as standard Java classes and advice is implemented as methods within these classes. AOP structures (such as pointcuts, weave model, etc.) can be specified in one of three ways: as annotations within the aspect classes, in a separate AOP XML file, or (as used in our approach) specified programmatically at run time.

2.5 Reflection

Reflection plays an important role in our framework and underpins various features used in dAOP in general. Reflection allows the introspection of a system and the gathering of information at run time [35]. By using reflection in our framework we can determine structural information such as field types and method signatures or more dynamic information such as the values of fields/parameters. This reflective information is used in our approach, in conjunction with the Framed Aspects, to generate the concrete aspect code; this process is described in more detail in Sect. 3.

2.6 Summary

To summarise, the framework for our implementation of auto-adaptive systems combines a number of technologies:

- ECA-based policies to define behaviour choices;
- Framed Aspects to give flexibility and reuse;
- dAOP to allow changes to be made dynamically at run time;
- Reflection to gather run-time information.

Using Framed Aspects, dAOP and reflection together allows the operator to dynamically weave code that is specifically generated for the target system's current status. By combining these technologies together with ECA rules, allows explicit control of *when* and *how* the aspects should be generated and woven, a feature not available in current dAOP techniques.

3 Framework Implementation and Behaviour

This section describes the implementation details of our framework. It will cover how the various technologies are combined and describe the various elements which contribute to implementing and managing an auto-adaptive system. Examples will also be given that illustrate how auto-adaptive behaviour can be applied and how undesirable aspect interactions can be prevented.

3.1 Policies

As mentioned earlier, policies are needed to ensure that the system will be adapted in the desired way and at the appropriate times. The policies in our solution are defined in an XML file (as it is relatively easy to be read by both humans and machines) and follows the structure of ECA rules:

- the *event* is some joinpoint being reached;
- the *condition* is some test performed on the state of the target system;
- the *action* involves the weaving or removing of an aspect.

Defining these ECA rules separately from the actual adaptations makes the framework much more maintainable as the behaviour specification is not tangled with other code. This also improves the reuse of the adaptations since defined policies tend to be system specific, whereas the adaptations should be as generic as possible.

The policies are initially processed when the system starts and are constantly monitored for changes so that any modification can have an immediate effect. From these policies, a set of aspects are generated and woven dynamically (see Sect. 3.2). When the specified conditions occur, these generated aspects initiate the execution of the specified actions. Figure 2 is an example of a policy and the attributes that it must contain.

```
<policy name="CachePolicy">
        <weave-condition property="ExecutionTime"   condition="more-than" value="2000"/>
        <remove-condition property="PacketsReceived" condition="more-than" value="20000"/>
        <points pointcut="execution(* autoadaptive.Test*.test*(..))"/>
        <aspect-type type="Cache" frame="true" advice=”cacheAdvice”/>
</policy>
```

Fig. 2. Example of the XML used to define a policy

A description of each of these fields and their function follows below. These fields are used to define the required behaviour, i.e., when an adaptation should be performed and what adaptation should take place. Note that a number of additional fields will also be used to define the relationships between the adaptations to prevent undesirable interactions; these are described in detail in Sect. 3.6.

Policy Name

This is a required field to give the policy a unique identifying name. The policy name is used to refer to and identify the policy whenever actions need to be performed upon it, such as it being removed or edited.

Weave Condition

This field requires three parameters to be specified that are combined to form the condition that triggers the actions of the policy. The three parameters specify the property to monitor, the operator for the condition, and the trigger value (e.g., the policy in Fig. 2 checks whether the execution time is more than 2000 ms).

Hardware resources can be observed including memory properties, CPU usage, hard disc usage, network properties and the execution time of a particular method. These physical properties can be monitored in any target system. In addition to these types of properties, run-time properties of the system can also be monitored, such as field values, parameter values and return values of methods. Obviously, these run-time properties are entirely dependent on the target system and the run-time properties it implements.

Weave-conditions can be combined using a set of AND/OR logical operators. These are implemented by listing the weave conditions sequentially and/or nesting conditions within another. Listing the conditions sequentially is equivalent to providing a set of conditions where any of them can occur for the action to be executed.

Whereas nesting the conditions allows the programmer to create a set of conditions where each of the conditions must be true for the action to be executed.

Remove Condition

This is an optional field that allows the programmer to specify a condition that triggers the *removal* of the adaptations associated with the policy. This field requires a similar set of parameters as the ones used to define the weaving conditions. Any system property can be used as part of this condition, i.e., different properties can be used in the remove condition and the weave condition. Care has to be taken to prevent inconsistent conditions from being defined. For example the weave and remove conditions could intersect, and the aspect just woven would then be immediately removed. To protect the operator from these conditions, the remove condition is evaluated *before* the aspect is woven. If it holds true then the aspect is not woven to prevent unnecessary operations. These conditions can also be listed sequentially or nested to specify complex conditions.

Points

In order to correctly monitor the target system for the specified conditions, the programmer needs to identify the points that are of interest. The extensiveness of the AspectWerkz joinpoint model allows the operator to naturally pick a wide variety of points and so we have chosen to use this format directly. This model was also extended slightly to allow certain events occurring within the framework to be captured, such as a particular aspect being woven or the actions of a policy being executed. However, such pointcuts are merely syntactic and are actually converted to AspectWerkz pointcut patterns at run time.

Aspect Type

The required aspect-type field allows the programmer to specify the aspect that has to be woven when the weave conditions are met. The parameters of this field allow the programmer to use two groups of aspects—Framed Aspects or concrete aspects (i.e., aspects which are already compiled and ready to be woven). If a concrete aspect is specified (indicated by setting the frame attribute to false), when the conditions are met the aspect can simply be woven. However, if a Framed Aspect is specified (indicated by setting the frame attribute to true), a concrete aspect needs to be generated using the current system state, as described in Sect. 3.2. An optional attribute included with this field (the pointcut attribute) allows the programmer to specify where the aspect should be woven. This enables the aspect to use a different pointcut from the one which the aspect monitoring for the weave/remove conditions occurring is attached. Finally, the advice attribute is used to specify which advice should be executed within the associated aspect.

3.2 Framed Aspects

Due to the nature of adaptive systems and the unpredictable state of the system at the time changes are required, it is often desirable to modify the adaptations to suit the current state. As described in Sect. 2.3 Framed Aspects allow the creation of code templates via parameterisation; these templates can be reused in a variety of different systems and scenarios. The same Framed Aspect can be used multiple times and in different systems but have different behaviour depending on the specification provided. Our framework uses the system state to generate this specification; therefore as the state evolves so does the specification.

In our implementation Framed Aspects are used for two purposes. The first is to generalise and construct the aspects used to observe the system behaviour for the weave conditions; these aspects are called *monitoring aspects*. The second purpose is to generalise the aspects that alter the system's behaviour, which we call *effector aspects*.

3.2.1 Monitoring Aspects

All the information required to generate and then weave the monitoring aspects is specified in the policies. The policies are processed to generate aspects that monitor the system for the specified conditions. Once the monitoring aspects have been woven, the conditions specified are evaluated each time the associated *event* (joinpoint execution) occurs. If the *condition* is true, then the *actions* associated with this policy are carried out.

If a policy is removed from the policy definition file at run time, the system is updated to reflect this change by removing *all* aspects that implement this deleted policy (i.e., both monitoring and effector aspects). Also, if significant changes (such as a change to the conditions or actions) are made to a policy at run time then aspects associated with this policy are removed, regenerated and rewoven. Additionally, if an entirely new policy is added at run time, this is processed to create a new monitoring aspect which is then woven dynamically to the target system.

3.2.2 Effector Aspects

The second purpose of the Framed Aspects is to enable concrete *effector aspects* to be generated that alter the target system behaviour. These concrete aspects are specifically suited to the current target system state. All that is needed is a specification that will bind a particular system attribute (or a function of a system attribute) to a particular parameter in the Framed Aspect. This will be illustrated through an example in Sect. 3.5.

Whilst any type of Framed effector aspect can be created, the difficult part is *effectively* parameterising the aspect code. The right balance has to be reached between generalising the aspect enough so it can be effectively reused/customised and making it too complex to use. Guidelines can be suggested to aid the programmer develop Framed Aspects that are both flexible and usable. These issues are discussed in more detail in [11].

The specification to assign the system run-time data to a Framed Aspect metavariable is specified in a user-defined binding class that must implement the IFramedAspect interface. A series of probes and help-classes are provided to ease the collection of run-time data and the setting of these metavariables.

3.3 System Management

In order to maintain and control the system, our framework includes an overall management system, responsible for:

- processing the defined policies at start-up and during run time;
- generating monitoring and effector aspects to implement these policies;
- ensuring aspect relationships are maintained to prevent undesirable interactions;
- weaving new aspects and also removing aspects when necessary;
- maintaining a list of aspects that have been woven to the base-system to determine the state of the system and to ensure relationships are maintained;
- monitoring for changes in the policies and making the necessary dynamic changes.

The AspectWerkz API allows the basic retrieval of the currently woven aspects. However, to allow more sophisticated queries to be made, this retrieval functionality is extended in our framework. Information can also be retrieved regarding an aspect's origin, such as its parent policy and other related aspects (such as requires and incompatible-with). This extra information eases the process of maintaining system consistency when weaving an effector aspect to the target system.

3.4 Dynamic Weaving

When using AspectWerkz, new aspects can be attached to existing pointcuts with relatively little overhead. However, when a new pointcut needs to be added, extra operations need to be performed that increases the overhead. AspectWerkz uses load-time modification of classes to weave aspects; hooks are added to the bytecode as classes are loaded. If a new pointcut is to be added to a class which has already been loaded, the "HotSwap" mechanism [36] must be used to reload the affected classes. Unfortunately, this method does have a relatively significant overhead as will be explained in Sect. 4. However, our belief is that the ability to weave new aspects and add new pointcuts is vital. Adaptations unanticipated at development time may be required, potentially needing to be woven at points where no pointcuts currently exist. Aspects can also be detached from pointcuts with relatively little overhead; however removing pointcuts results in a similar overhead to adding pointcuts.

When a pointcut is either added or removed, each class affected by the pointcut needs to be HotSwapped. Although this operation is similar to the process of replacing components as described earlier in Sect. 2.1, the disturbance is in fact much less. HotSwapping of classes does not block or halt any invocations and the system is not suspended. Therefore this process removes these disturbances experienced in component solutions [8].

Note that one potential solution to prevent this overhead completely would be to instrument every possible joinpoint. However, a problem with this solution is that this to adds an overhead. Each time a potential joinpoint is reached a check is performed to gauge whether an aspect should be executed. Although the overhead of each individual check is not as large as the overhead of a single HotSwap operation, the overhead will be significant over long periods of time since many checks will have to be performed. This accumulative effect is undesirable for long-running systems. Additionally, instrumenting every possible joinpoint will cause a significant rise in the memory foot-print of the target system.

3.5 Dynamic Cache Example

To illustrate how the framework operates and how it can be used to implement auto-adaptive behaviour, the following example involves creating a dynamic cache to demonstrate all the key features of our approach.

The GUIDE system [37] is an interactive tour guide which runs on a wireless hand-held device. The purpose of this system is to create tours of Lancaster depending on the tourist's interests and to deliver information regarding tourist attractions around Lancaster. Due to the mechanism used to deliver data to the device, a desirable feature would be to introduce a cache when delays are experienced on the wireless network. However, due to the resource constrained nature of the device it is not always desirable to have the cache present due to the memory constraints this imposes. Our approach allows a policy to be created dynamically that defines the conditions that should trigger the addition/removal of the cache aspect.

3.5.1 Policy Definition

The following policy (Fig. 3) defines a set of conditions and actions that implement the above dynamic cache behaviour. When this policy is processed, a monitoring aspect is generated automatically by the auto-adaptive framework; this aspect's role is to observe the execution time of the specified method.

The policy is given the name CachePolicy, which has to be unique to prevent name conflicts. The condition specified causes action to be taken when the average execution time of the specified method exceeds 500 ms. This condition is applied to methods matching the pattern * guide.Client.get(String). If the execution time exceeds 500 ms then, as specified, a concrete caching aspect is generated from the Framed cache aspect. Significantly, this policy also defines a remove-condition that identifies the conditions when the caching aspect should be removed. In this example, when there is less then 50 Mb of free memory the cache will be removed and so make available more memory to the rest of the GUIDE system. As a pointcut pattern has not been specified to define where the cache aspect should be woven, the cache aspect will be woven to the same pointcut as the monitoring aspect which implements this policy.

```
<policy name="CachePolicy">
    <weave-condition property="ExecutionTime" condition="more-than" value="500"/>
    <remove-condition property="FreeMemory" condition="less-than" value="50"/>
    <points pointcut=" execution(* guide.Client.get(String))"/>
    <aspect-type type="Cache" frame="true" advice=”cacheAdvice”/>
</policy>
```

Fig. 3. Policy to monitor the execution time and to weave a Cache Framed Aspect

3.5.2 Framed Monitor Aspects

To aid the implementation of the policies, a library of monitoring aspects is provided that can monitor various properties of the system state. Such properties have already been listed in Sect. 3.1 when describing weave conditions. The monitoring aspects have been parameterised so that concrete aspects can be generated to suit the policies.

The Framed code in Fig. 4 monitors the execution time of the specified method and is enclosed by an ifdef Framed Aspect command. If the property to be monitored is the execution time, then this section of code will be selected and included in the concrete monitoring aspect. All the monitoring aspects are built in this manner by using conditional compilation to select and choose the appropriate segments of code.

```
<ifdef item-in="property" value="ExecutionTime">
        long ExecutionTime= ProcessInformation.getExecutionTime();
        if(ExecutionTime<@Operator><@TestValue>){
          <adapt frame="ImplementChanges.frame"/>
        }
</ifdef>
```

Fig. 4. Framed monitor aspect code

The *if*-statement in Fig. 4 has been parameterised (using the value of `@` command) to allow the operator and test value to be substituted so the framework can monitor for the desired conditions. The `adapt` frame command allows code to be inserted that is encapsulated in other frames. Figure 5 shows the contents of the ImplementChanges frame, this code is responsible for implementing the required actions when the specified conditions occur.

```
<ifdef item-in="frame" value="true">
 <@Type> aspect= new <@Type>();
 aspectName<@Name>= AW.weaveEffector(<@Name>, this.getClass().getName(), <@Type>,
     <@Expression>, <@Frame>, aspect);
</ifdef>
```

Fig. 5. Framed code to weave the effector aspect

A number of elements in Fig. 5 have been parameterised to customise the actions taken. These parameters are used to define the type of effector aspect to weave, the pointcuts where it should be woven and also information regarding its origins, i.e., the policy and monitor aspect that requested it to be woven. It is important to emphasise that the parameterisation of these aspects leads to highly customisable code.

3.5.3 Framed Effector Aspects

The Framed cache aspect is shown in Fig. 6. As can be observed, the basic structure of the code contains a class that will act as a caching aspect, which itself contains a subclass (CacheDS) which is the data structure for storing cached items.

Firstly, the parameterised value that specifies the classname allows multiple instances of the same Framed Aspect to be generated. The Framed cache could be used in other policy definitions; if the classname were not parameterised then this would cause naming conflicts within the JVM. The next parameterised values specify the cache size and percentage to delete allowing the programmer to bind a property of the system state to these parameters. For example, the programmer could specify that the cache size/percentage to delete should be based on the amount of free memory.

Finally, elements within the CacheDS class are also parameterised to allow different types to be stored within the cache. The type to be cached can be retrieved via reflection at run time and then bound to the TYPE parameter to create a cache to store the specific type. Care has to be taken to ensure that the aspects are only applied to methods that use the same types to prevent type incompatibilities from occurring.

```
<frame name="Cache">
        public class <@CLASSNAME/>{
          private int CACHE_SIZE= <@CACHE_SIZE/>;
          private int PERCENTAGE_TO_DEL= <@TO_DEL/>;
          private HashTable cache= new HashTable(CACHE_SIZE);

          public Object cacheAdvice(final JoinPoint jp) throws Throwable{
            <@RES_TYPE/> key= (<@RES_TYPE/>) jp.getRtti().getParameterValue()[0];
            CacheDS result= (CacheDS)cache.get(key);
             if(result!=null)
               return result;
              else{
                result= new CacheDS(jp.proceed());
                cache.put(key,result);
              }
            return result.getData();
          }
          class CacheDS{
             private <@TYPE/> data;
             private long timestamp;
             public CacheDS(<@TYPE/> d){
                data= d; }
             public <@TYPE/> getData(){
                return data;
             } } }</frame>
```

Fig. 6. Framed cache code

```
set= objFactory.createFrameTypeSetType();
set.setVar("CACHE_SIZE");
int size= (MemoryInformation.getFreeMemory()/100)*10;
set.setValue(size);
setList.add(set);
```

Fig. 7. Code to bind the amount of free memory to the CACHE_SIZE parameter

Figure 7 shows the binding of the amount of free memory to the CACHE_SIZE parameter to create the frame specification (TO_DEL parameter is set in a similar way).

With all these elements implemented, the original client system executes as before but with the monitoring advice being executed around any methods matching the pointcut pattern specified in the policy. When the average execution time of any of these methods rises above 500 ms, an instance of cache aspect is generated and woven.

Figure 8 shows the generated code of the concrete cache aspect that has been customised for the current run-time conditions. As can be observed, all the parameterised elements have been substituted for concrete values. The cache size and percentage to delete have definite values which are based on the amount of free memory. The classname is created which is a combination of the policy name and the Framed Aspect it is generated from. The types used in the CacheDS class have also been substituted for the type to be cached. In this example, this is the return type of the method being monitored (determined using reflection).

Note that some of the policies will be statically defined and loaded when the system starts. However, a dAOP approach is still essential since if, later during the execution of the system, policies are removed then the aspects that implement these policies have to be dynamically removed. A further benefit of dAOP is that the

Framed Aspects in our framework rely on run-time information to generate the concrete aspects. dAOP is needed to weave these run-time generated aspects.

```
public class ResponseTime_Cache{
  private int CACHE_SIZE= 200;
  private int PERCENTAGE_TO_DEL= 2;
  private HashTable cache= new HashTable(CACHE_SIZE);

  public Object cache(final JoinPoint jp) throws Throwable{
    //cache implementation
  }
  class CacheDS{
    private Document data;
    private long timestamp;
    public CacheDS(Document d){
      data= d;
    }
    public Document getData(){
      return data;
    }
} }
```

Fig. 8. Concrete cache aspect

Summary of Process
To summarise, the general order of events for creating/applying an adaptation in our framework is as follows:

– the policies are parsed;
– Framed Aspects are used to generate a concrete monitoring aspect for each policy;
– these aspects monitor the specified system conditions and when necessary execute the appropriate actions;
– the actions involve a combination of generating concrete aspects from the specified Framed Aspects, weaving aspects or removing aspects;
– policies are continually monitored for changes and the system is dynamically altered to reflect these modifications.

3.6 Interaction Issues

This section will examine some of the potential undesirable interactions that can occur between aspects and will discuss how they can be prevented. The examples presented extend the previous cache example.

3.6.1 Incorrect Execution Order

Suppose the policies of the target system are altered such that, in addition to the cache aspect, an authentication aspect is now required. If this authentication aspect is woven to the same joinpoint as the cache, an undesirable interaction could occur.

In situations where multiple aspects are woven to the same joinpoint, an advice chain is generated that represents the order in which the advice will be executed. If the proceed method is called from within some advice and the end of the advice chain has not been reached, the next advice in the chain is executed. However, once the last advice in the chain is reached, calling the proceed method will result in the advised joinpoint being executed. This advice chain may be broken at any point by an item of advice *not* calling the proceed method. This results in any remaining advice, and the advised joinpoint, not being executed.

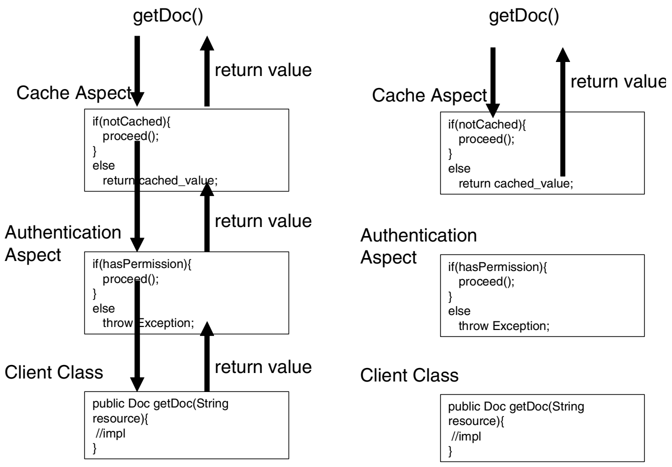


Fig. 9. Flow diagram that shows the sequence of events when an item is added to the cache (*left*) and when an item is retrieved from the cache (*right*)

As can be observed in Fig. 9 (left), situations where the required resource has *not* been cached the advice chain is executed normally, with both the authentication advice and advised method being executed. However, if the required resource *is* cached (Fig. 9 right) then the advice execution chain is broken as the cache advice does not call the proceed method. This causes the authentication aspect not to be executed and users could be granted access to resources they are not authorised to.

This is obviously an undesirable situation and a solution is needed to prevent this type of interaction from occurring. Our approach allows priorities to be assigned both explicitly and implicitly using the rank element of the policies. Firstly, both aspects can be assigned definite priorities (via their respective policies) so that the desired execution order can be achieved, as shown in Fig. 10.

```
<policy name="AuthenticationPolicy>
  …
  <rank value="1"/>
</policy>
<policy name="CachePolicy>
  …
  <rank value="2"/>
</policy>
```

Fig. 10. Policy definitions with both aspects being given a definite rank

Both related policies in Fig. 10 have been given a definite rank, with the authentication policy having a priority of 1 and the cache policy having a priority of 2. This means that the aspects associated with the authentication policy will always be executed before the aspects associated with the cache policy, regardless of the order in which the aspects were woven.

However, this method for defining priorities is not very flexible or dynamic. For example, when a new policy is added at run time, the priority of the other policies may need to be modified to assign the new policy the desired priority. This could turn into a time consuming (manual) activity if several policies need to be edited so that the new policy can be assigned the correct priority. A more flexible solution that is supported by our framework is to assign the new policy a priority that is *relevant* to the other policies. As shown in Fig. 11, this will allow the new policy to be assigned the correct priority without having to edit any other policies.

```
<policy name=”AuthenticationPolicy>
  …
  <rank value=”++CachePolicy”/>
</policy>
<policy name=”CachePolicy>
  …
  <rank value=”1”/>
</policy>
```

Fig. 11. The same policies as Fig. 10 but the authentication policy is given a rank relative to the cache policy

The policy shown in Fig. 11 specifies that the aspects woven by the authentication policy should be executed *before* the cache aspects. Conversely if the aspects needed to be executed *after* a policy, the ++ should be replaced with --.

Although in Fig. 11 the authentication policy rank is only relative to one other policy, multiple relationships can be specified if necessary. This allows complex ordering to be defined with a new aspect able to be inserted between a pair of existing aspects or before/after a set of aspects.

3.6.2 Incompatible Aspects

A scenario could occur where a set of aspects are incompatible with another set of aspects and, in order to operate correctly, one set needs to be absent/removed from the target system. In such a case, the framework is responsible for ensuring that no combination of incompatible aspects are woven simultaneously to the target system.

Suppose in our example that the policies are modified again so that an encryption aspect must be woven at the same joinpoint as the cache and authentication aspect. The purpose of this new aspect is to ensure the privacy of the users of the target system by encrypting the objects retrieved by the user. However, this aspect is incompatible with the cache aspect for two reasons. Firstly, if the cache aspect is executed before the encryption aspect, the objects would be cached with no encryption applied. Secondly, even if encrypted objects were cached this would still be inappropriate as there is no guarantee that the user accessing the cache has the correct key to decrypt the object due to the possibility of keys expiring.[1]

[1] Cryptographic keys often have an associated expiry time, after which new keys have to be generated.

It is clear from this scenario that if the encryption aspect is required then the cache aspect should be removed from the target system (assuming that the encryption aspect has precedence over the cache aspect). This relationship can be specified using the incompatible-with element of our policy structure shown in Fig. 12.

```
<policy name=”EncryptionPolicy>
  …
  <rank value=”++CachePolicy”/>
  <incompatible-with aspectname="Cache" frame="true" remove-aspect="true"/>
</policy>
```

Fig. 12. Specifies that the cache aspect is incompatible with the encryption aspect

Figure 12 shows the encryption policy has been assigned a relatively higher priority than the cache policy. This ensures the encryption aspects will always have precedence over the cache aspect. The incompatible-with element is used to specify that the encryption policy and its aspects are incompatible with the cache framed aspect.

Before any aspects are actually woven, checks are carried out to ensure that the current configuration of the target system allows the encryption aspect to be woven, e.g., the framework checks for the presence of any incompatible aspects. If an incompatible aspect is detected, then the highest priority policy gets precedence. In this example, the encryption policy has priority over the caching policy; hence the cache aspect is removed to ensure the incompatible-with relationship is not violated.

3.6.3 Requires Aspect

Alternatively, an aspect may actually *require* the presence of another aspect for it to operate correctly. Suppose that for accounting purposes, logged authentication became necessary, i.e., both authentication *and* logging aspects needed to be woven. Our approach allows a ‘requires’ relationship to be specified between the authentication aspect and the logging aspect. This can be used to ensure that both aspects are woven at the same time and to the same pointcuts.

Our policy specification can be used to define this type of relationship as shown in Fig. 13. The element displayed can be used multiple times to define all the aspects/advice that are required for the policy to be successfully implemented.

```
<policy name=”AuthenticationPolicy>
  …
  <requires-aspect aspectname="LoggingAspect" frame="false" advice=”log”/>
</policy>
```

Fig. 13. Specifies a ‘requires’ relationship between the authentication and logging aspect

Adding this requires-aspect element will result in the logging aspect being woven to the same pointcut and at the same time as the authentication aspect. The logging aspect will have the same priority as the authentication aspect, but will be executed after it due to it being specified later in the authentication policy.

If the required aspect is to be woven at a different joinpoint to the current policy, the optional pointcut attribute can be included within the required aspect definition.

3.6.4 Resolution Aspects

The final relationship between aspects to be examined here concerns resolution aspects [38]. A resolution aspect is used to resolve conflicts between potentially incompatible aspects and allows them both to operate together.

Continuing with our running example, this type of relationship exists between the logging aspect (that was required by the authentication policy) and the encryption aspect. The encryption aspect was introduced to provide some form of privacy to the users of the target system. However, this requirement would be broken by the logging aspect due to it storing the entries it makes in plain-text.

A resolution aspect can be woven to allow these two aspects to co-exist without conflicting. To achieve this, an aspect is required that encrypts the contents of the log file generated by the logging aspect. However, this log file encryption aspect is only required when *both* the logging and encryption aspects are present. A set of conditions can be defined that specifies when the resolution aspect is required. A special weave condition property, AspectWoven, can be used to check whether the specified aspects are woven. The policy definition to achieve this is shown in Fig. 14.

```
<policy name="ResolutionPolicy">
  <weave-condition property=”AspectWoven” condition=”equals” value=”LoggingAspect”>
    <weave-condition property=”AspectWoven” condition=”equals” value=”EncyptionAspect”/>
  </weave-condition>
  <points pointcut="actionexecute(AuthenticationPolicy)||actionexecute(EncryptionPolicy)"/>
  <aspect-type name=”LoggingResolution” joinpoint=”execution(* *.RemoteObject.get*(..))” ad-
           vice=”encryptLog”/>
  <rank value=”--AuthenticationPolicy”/>
</policy>
```

Fig. 14. The resolution policy to fix the interaction between the logging and encryption aspect

As can be observed, two weave-conditions have been defined, one nested inside the other to implement a logical AND operation between them. The first weave-condition specifies that the logging aspect needs to be present; the second specifies that the encryption aspect has to be present. Only when both of these conditions are met is the operation to weave the logging resolution aspect triggered. The remaining elements in Fig. 14 are used to specify the aspect to be woven (in this case the LoggingResolution aspect) and the priority that the aspect should take. It should be noted that the points element in this example specifies a pointcut unique to our framework (i.e., an extension to the AspectWerkz joinpoint model, as discussed in Sect. 3.1). The pointcut specified in Fig. 14 attaches the monitoring aspect to events occurring in other policies; in this case when the actions of AuthenticationPolicy or EncryptionPolicy are executed. Other policy events supported include the removal of adaptations associated with a particular policy and the execution of a particular policy monitoring aspect.

As the target system executes, the conditions could alter so that either the logging or the encryption aspect is removed. This will result in the resolution aspect no longer being needed. The policy in Fig. 14 can be extended by specifying the conditions under which the logging resolution aspect should be removed. This will allow the system to adapt itself according to the current conditions. These extensions are shown in Fig. 15.

```
<policy name="ResolutionPolicy">
      ...
  <remove-condition property=”AspectWoven” condition=”notequals” value=”LoggingAspect”/>
  <remove-condition property=”AspectWoven” condition=”notequals” value=”EncyptionAspect”/>
</policy>
```

Fig. 15. Policy extensions to remove the logging resolution aspect when it is no longer required

The changes made to the resolution policy specify that the logging resolution aspect should be removed when either the logging aspect *or* the encryption aspect is not present. This logical OR operator is implemented by listing the conditions serially.

3.6.5 Other Issues

Outside policy definitions, there are additional issues that need to be addressed by the framework to ensure that policies are correctly and consistently applied. The first issue involves the actions taken by the framework when a policy cannot be correctly applied due to some incompatible aspects with higher priorities already woven. Under these circumstances, to prevent aspects being deployed in a state which is incompatible, none of the actions specified by the policy are performed. If the state alters and the incompatible aspects are no longer present, the aspects will then be deployed automatically. However, the incompatible relationship only comes into effect when *both* sets of incompatible aspects should be woven to the target system at the *same* time.

A more subtle issue that could arise is the problem of intra-policy inconsistencies. We have already addressed how inter-policy consistency is addressed through the use of priorities, but intra-policy inconsistency is more difficult to solve. For example, a policy could specify that it requires a particular aspect, yet at the same time specify that it is also incompatible with that same aspect. This would obviously cause issues when the policy came to be applied. To prevent such issues arising, when the policies are first loaded, checks are carried to ensure such relationships have not been specified. If any intra-policy inconsistencies are present, the affected policy is not loaded and the user warned of the problems detected.

3.6.6 Interaction Summary

This subsection has, with examples, covered some of the undesirable interactions that can occur between aspects that are applied dynamically and automatically at run time. We have addressed how these can be prevented with the specification of the execution order, incompatible aspects, requires aspects and resolution aspects. It is vital that these interactions are prevented to allow the target system and the aspects to operate correctly. Our policy specification allows users to specify a variety of types of relationships that could exist between aspects. These relationships are enforced at run time by the auto-adaptive framework to ensure that the system operates as intended. It is worth noting that these undesirable interactions are addressed on an aspect-by-aspect basis. Although this may seem to be a limiting factor, it is necessary to deal with conflicts at this level to ensure precise relationships can be defined. If necessary, this approach could be extended to allow conflicts to be defined on a per-policy basis. Furthermore, the use of aspect interaction catalogues [39] could aid the process of both identifying and specifying such conflicts.

3.7 Ensuring Weaving Consistency

When multiple aspects are required to implement a particular adaptation, it is important that transient inconsistencies do not occur. To prevent this, aspects must be woven atomically to ensure that all required aspects are woven simultaneously and there is never a time during the adaptation process when a required aspect is absent.

In our framework, before an adaptation that involves weaving multiple aspects is applied, all incompatible-with relationships are queried to ensure none of these will be broken. If the adaptation passes these checks then the process of weaving the aspects begins. This initially involves modifying the bytecode of the classes that the aspects affect. After this process the affected classes still require HotSwapping to allow the modified bytecode to come into affect. The affected classes are HotSwapped concurrently to ensure that the woven aspects become active simultaneously and so avoiding the issue of transient inconsistencies.

4 Evaluation

To illustrate the extent to which we have met our objectives, a series of evaluation techniques have been used. A number of performance measurements have been carried out to show the overhead added from using our framework. In order to assess the aims relating to reusability, flexibility, and maintainability of the adaptations, an AO metric is applied to measure these properties. In addition, to demonstrate the effectiveness of aspects and more specifically Framed Aspects on these properties, certain elements used within the framework were re-implemented using OO and XVCL to act as a comparison to our implementation. The target system used in this evaluation is the GUIDE system described in Sect. 3. The cache aspect presented will also be used throughout this section to measure appropriate operations. It should be noted that the actual functionality of the aspect being applied does not affect the overhead experienced.

4.1 Performance Discussion

As discussed earlier, there is an overhead added to systems that are implemented using dAOP. The framework presented allows programmers to adapt system behaviour in various ways and also allows the adaptations themselves to be customised via parameterisation. This extra flexibility inevitably adds an overhead since the data used to create the concrete aspects must be gathered, the aspects need to be generated and finally the generated aspects must be woven dynamically. Also, the execution of the advice that implements the adaptations introduces their own delay. This section will illustrate the extent of these overheads; all tests were run on Java 1.5, on a Pentium M 1.4 Ghz processor, 512 Mb RAM machine.

4.1.1 Weaving a New Aspect to a New Pointcut

The readings shown in Table 1 present the length of time taken to weave an entirely new aspect to a new pointcut. This action involves generating a concrete aspect (from a Framed Aspect), modifying the bytecode of the classes to implement the pointcut and then HotSwapping these classes in order for their new implementation to take effect.

The time required to generate a concrete aspect from a Framed Aspect is dependent on the number of Framed Aspect commands used; as the number of commands increases so to will the required generation time. To keep this time consistent in each of the measurements, the same Framed Aspect will be used. The framed cache aspect described earlier in Sect. 3.5 will be used as we feel this represents a typical Framed Aspect with a typical number of commands. To obtain a range of measurements, the pointcut is simply altered to affect a wider range of classes.

Table 1. Times for weaving a new aspect to a new pointcut including aspect generation time

Affected classes	0	1	2	3	4	5	6	7	8	9	10	11	12
Time (ms)	0.40	22	30	43	61	72	89	103	132	133	147	149	166

The figures show how the time to weave a new aspect to a new pointcut increases as the number of affected classes increases. This can be expected due to the number of operations required to modify the bytecode of *each* of the affected classes. One result to note is that when zero classes are affected, this does not necessarily mean that no classes are affected by the pointcut, but could mean that no classes affected by the pointcut have been loaded yet. The execution time for this value is significantly lower than the rest of the values due to the fact that no class requires bytecode modifications (and consequently no HotSwapping). Instead the bytecode of any affected class will be modified as it is loaded, the overhead of this operation is examined in Sect. 4.1.3.

4.1.2 Weaving a New Aspect to an Existing Pointcut

The results shown in Table 1 required bytecode modifications and HotSwap operations to be executed. However, these operations do not have to be carried out when a new aspect is woven to an *existing* pointcut, thus reducing the total weave time. The times for executing this operation (which again includes the time to generate a concrete aspect from a Framed Aspect) are shown in Table 2.

Table 2. Times for weaving a new aspect to an existing pointcut

Affected classes	1	2	3	4	5	6
Time (ms)	0.298	0.307	0.304	0.307	0.311	0.317
Affected classes	7	8	9	10	11	12
Time (ms)	0.320	0.322	0.327	0.330	0.331	0.330

As can be observed from these results the time required to weave a new aspect to an existing pointcut is significantly lower than when weaving a new aspect to a new pointcut. This reduction is because the hooks to implement the pointcut have already been added. In these tests the aspect is simply generated and then attached to the pointcut. A slight increase in the execution time can be observed as the number of affected classes increases, this is due to checks having to be performed for each affected class to ensure the pointcut is implemented.

It is important to differentiate the results between these two operations as they will be executed with differing frequencies. Under normal operating conditions, this second operation (weaving a new aspect to an existing pointcut) will be executed more frequently since, once a monitoring aspect has been woven, all subsequent effector aspects woven to the same pointcut will reuse the existing pointcut. As can be seen in the examples presented in Sect. 3, it is likely that effector aspects will be woven to the same pointcut as its associtated monitoring aspect.

4.1.3 Class Loading

Mentioned earlier in Sect. 4.1.1 was the fact that a new pointcut could be added which does not currently affect any loaded classes. It is expected that when a class affected by a pointcut is loaded it would take longer to load than a new class not affected by any pointcuts. To examine this a simple test was performed. This first involved loading a class affected by a pointcut and then loading the same class when not affected by any pointcut (both were loaded through the modified AspectWerkz class-loader). The results are shown in Table 3.

Table 3. Results of loading classes affected by a pointcut compared with ones not affected

Class modifcations	No modifications	Pointcut added
Load time (ms)	1.6	23

As can be observed it takes a substantially longer length of time to load a class which requires modifications than the time required to load a class which requires no modification. This delay is introduced due to the modification of bytecode to implement the required pointcut and the checks to determine whether this modification is necessary. An interesting point to note is that the time to load and modify a new class is almost identical to the time to modfiy and HotSwap a single class (see Table 1), 23 ms compared with 22 ms.

4.1.4 Removing an Aspect

The next set of results present the average time required to remove an aspect. This required two sets of readings to be taken. The first measured the time to remove an aspect from the target system and the second when both an aspect *and* a pointcut are removed.

Both of these operations are required for the following reason. When an aspect is removed from the target system it may be inappropriate to remove the pointcut to which it is attached as the pointcut may be still in use by other aspects. However, when the pointcut is no longer in use then it is appropriate to remove the pointcut and avoid any unnecessary future overhead. Tables 4 and 5 show the measured times to execute these operations.

Table 4. Times for removing an aspect from a pointcut

Affected classes	1	2	3	4	5	6
Time (ms)	0.285	0.284	0.292	0.295	0.302	0.294
Affected classes	7	8	9	10	11	12
Time (ms)	0.303	0.306	0.308	0.314	0.315	0.307

Table 5. Times for removing an aspect and a pointcut

Affected classes	1	2	3	4	5	6
Time (ms)	8	17	19	20	26	27
Affected classes	7	8	9	10	11	12
Time (ms)	33	35	37	40	48	45

The same pattern which occured in the add aspect measurements also occurs in the above results. The times of both sets of results increase as the number of affected classes also increase. Again this is due to operations having to be executed for each affected class. As before, there is a significant difference between the times recorded for removing just an aspect compared to when both an aspect and a pointcut need to be removed. This difference is again due the operation involving bytecode modifications and HotSwapping.

The measured times to remove just an aspect are comparable to the times to add just an aspect, since both of these operations are similar in complexity. However, there is a difference in execution times between removing an aspect and pointcut compared with adding an aspect and pointcut. This can be explained due to the increased processing required to generate and add the bytecode to implement a pointcut. Removing a pointcut involves the simpler task of only removing the bytecode.

4.1.5 Advice Execution

The final set of measurements involve the execution time of a piece of advice within the auto-adaptive framework. This will be the most frequently executed operation and so it is vital that the overhead is at a minimum. The results displayed in Table 6 show the average execution time of an an empty regular Java method, an empty advice executing in AspectWerkz and an empty advice executing in our framework. These results are provided to give a comparison of overhead from our framework.

Table 6. Measured execution times of advice/methods

Implementation	Regular Java method	AspectWerkz advice	Auto-adaptive framework advice
Time (ns)	5	610	920

As can be observed, the execution of AspectWerkz advice is substantially slower than the execution of a regular method. This is due to the reflective information which is gathered when the pointcuts are reached and the increased levels of indirection. There is also a difference between the execution time of advice in our framework and with the execution of advice in the regular version of AspectWerkz. This increase is caused by extra context information being collected by our framework whenever joinpoints are met. This information is then used by the framework to determine what actions (if any) should be executed.

It is always desirable to improve system performance and our framework is no exception. AOP and in particular dAOP are still in their infancy. As the technology gains in maturity, the efficiency of approaches used should improve, and should result in improved performance in our framework. This has already been shown in the major performance improvements in a more recent version of AspectWerkz [40].

4.1.6 Summary

This section has provided performance figures for significant operations performed within our framework. Although some of the figures are relatively high, namely those that involve bytecode modification and HotSwap operations, these are operations that will be performed the least frequently. Conversely, the operations that will be performed the most frequently, such as advice execution or adding a new aspect to an existing pointcut, have a much lower overhead and so should minimise the impact on the overall overhead. Obviously there will be some overhead from using the framework but this is a trade-off from gaining increased flexibility and adaptive behaviour.

The main factor that affects the scalability of the framework is the pointcut definitions. If a very specific pointcut is specified (i.e., one with no wildcard tokens) then the size of the target system will not affect the overhead experienced. However, if a generalised pointcut (i.e., one with wildcard tokens) is specified then the number of classes present in the target system will affect the overhead (as per the presented results). Furthermore, the pointcut type can affect the performance. For example, weaving an aspect to a call pointcut is more expensive then an execution pointcut. An execution pointcut only requires the target side to be advised whereas a call pointcut requires all classes where the identified method is called to be advised. As such, the performance overhead is entirely customisable by the user. Users can be guided to avoid unnecessary overhead by using the most appropriate pointcut type (e.g., using execution pointcuts instead of call pointcuts) and providing more concise pointcut patterns.

4.2 Software Properties Discussion

In order to be able to quantitatively assess the reusability, maintainability and flexibility of the adaptations being applied to the target system, a suitable metric had to be chosen which can measure the desired properties. A number of metrics have previously been proposed for Object-Oriented methodologies [41, 42] to assess them through empirical studies. However, applying these existing metrics to AOSD implementations is unsuitable due to the way AOSD alters the coupling and cohesion between components [43]. A metric is needed that takes these differences into account, and allows AOSD implementations to be evaluated using empirical studies.

The majority of previous work in developing metrics for AOSD techniques has focused upon qualitative assessment rather than quantitative [44]. To assess the software properties objectively, the AO quantitative assessment framework developed at PUC-Rio was selected [45, 46]. This framework relies on both a quality model and a metric suite. The quality model maps the relationships between the attributes to be evaluated and the various metrics in the metric suite. The metric suite defines various metrics that capture information regarding software attributes such as separation of concerns, coupling, cohesion and size in terms of physical properties of the code. Existing metrics were extended and customised to suit AO software whilst allowing them to still to be applied to OO software. Furthermore, entirely new metrics were also introduced which again can be applied to both AO and OO software to offer comparisons[2] between the two paradigms. These changes and new metrics are detailed in [46]. The list of metrics and the attributes that they map on to are shown in Fig. 16 and Table 7 which are taken from [47].

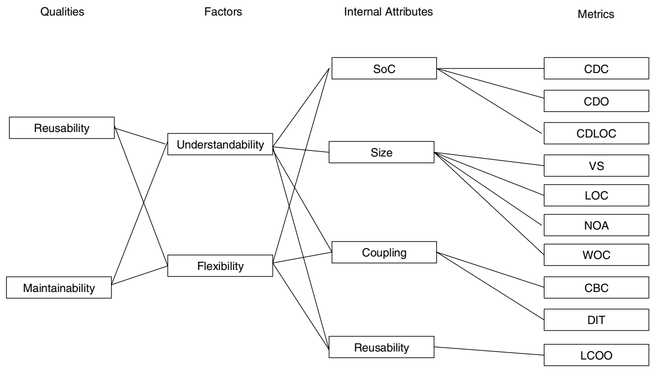


Fig. 16. The quality model used to assess the reusability and maintainability

Table 7. Lists the metrics used and the attributes they map to

Attributes	Metrics	Definitions
Separation of Concerns	Concern diffusion over components (CDC)	The number of classes/aspects that a given concern is spread over and the number of class/aspects that access them
	Concern diffusion over operations (CDO)	The number of methods/advices that a given concern is spread over. Additionally, the number of methods/advices that access them are counted as this contributes to the implementation of the concern and so contributes to the diffusion
	Concern diffusions over LOC (CDLOC)	The number of transition points for each concern; a transition point is where there is a concern switch within the code
Coupling	Coupling between components (CBC)	The number of classes that are used in attribute declarations, formal parameters, return types, throw declarations, and local variables. Each class used is only counted once
	Depth inheritance tree (DIT)	How far down the inheritance tree an aspect/class is
Cohesion	Lack of cohesion in operations (LCOO)	The number of method/advice pairs per class that do not access the same instance variable
Size	Lines of code (LOC)	Number of lines of code in a class/aspect
	Number of attributes (NOA)	Number of attributes in a class/aspect
	Weighted operations per comonent (WOC)	A sum of the number of methods/advices in each of the classes/aspects and the number of parameters

[2] An important point to note is that it is difficult to directly compare AO and OO. For example, the notion of a pointcut does not exist in OO. As such, only equivalent features can be directly compared.

4.2.1 Applying the Metrics

Initially, the metrics were applied to the concrete cache aspect (generated from the Framed cache aspect) used earlier in Sect. 3.5. In order to offer a comparison to the AO implementation the cache was re-implemented using OO. In addition to re-implementing the cache using OO, it was also necessary to modify the base-code of the target system to allow the OO cache to be executed correctly. The modifications to the target system are taken into account in the metric readings.

As an extension to comparing the OO and AO implementations we also applied the metrics to the Framed version of the cache aspect. The aim of this was to illustrate the benefits of using Framed Aspects. To allow the metrics to be applicable to the Framed Aspect code a number of the metrics had to be altered slightly. Below is a summary of the assumptions/changes made:

- Concern Diffusion over Components (CDC): include the number of frames that contribute to implement a given concern.
- Coupling Between Components (CBC): do not count parameterised types.
- Lines of Code (LOC): include all frame commands in the count.
- Weighted Operations per Component (WOC): include the number of Framed Aspect commands.

Again, to provide a comparison to the Framed Aspect adaptations we also implemented the adaptations using XVCL. The above assumptions to apply the AO metrics to Framed Aspects are applicable to XVCL.

```
public URL getResource(String res){
  URL value= cache.checkCache(res);
  if(value==null){
    value= server.getDoc(res);
    cache.cacheObject(res,value);
  }
  return value;
}
```

```
public URL checkCache(String key){
  if(cache.get(key)==null)
    return null;
  else
    return ((CacheDS)cache.get(key)).getData();
}
public void cacheObject(String key,URL value){
  CacheDS result= new CacheDS(value);
  cache.put(key,result);
}
```

Fig. 17. a Changes made to the target system. **b** The methods required to implement caching behaviour when OO is used.

The differences between the AO and OO implementation and the necessary changes made to the target system base-code are highlighted in Fig. 17a. Once the changes had been made to the adaptations and base-code, we applied the metrics to both versions. The measurements recorded are shown in Tables 8 and 9.

Table 8. Metric readings for the cache concern implemented using AO

Resource	CDC	CDO	CDLOC	CBC	DOIH	LCOO	LOC	NOA	WOC
Base_AO				2	0	0	10	1	2
Cache_AO				7	0	0	34	3	2
Cache AO total	1	4	1	9	0	0	44	4	4

Table 9. Metric readings for the cache concern implemented using OO

Resource	CDC	CDO	CDLOC	CBC	DOIH	LCOO	LOC	NOA	WOC
Base_OO				3	0	0	16	2	2
Cache_OO				3	0	0	37	3	5
Cache OO total	2	6	8	6	0	0	53	5	7

As can be seen there is an improvement in all but one of the metrics when using AO to implement the caching feature. The only metric where AO fairs worse is the CBC metric. This is due to the extra types required for the aspect to be successfully implemented, such as the JoinPoint and MethodRtti types that allow access to the parameters provided to the advised method. Coupling to these types does not affect the aspect's reusability. The two most significant improvements when using AO is found under the Concern Diffusions over LOC (CDLOC) and WOC metrics. The large increase of the CDLOC metric in the OO implementation is due to the changes to the base-code that are necessary to call the cache concern at the appropriate places. As a result, there are concern switches between the original concern and the cache concern. There is also a significant 43% increase in the WOC metric when using OO, since more operations are required to implement the cache when using OO compared to AO, hence the complexity is increased.

To examine the impact of Frames, the metrics were also applied to the Framed implementation of the cache aspect. The results from applying the metrics to this module are shown in Table 10.

Table 10. Metric readings for the cache concern using Framed Aspects

Resource	CDC	CDO	CDLOC	CBC	DOIH	LCOO	LOC	NOA	WOC
Base_AO				2	0	0	10	1	2
Framed Aspect				5	0	0	36	3	8
Framed AOP total	1	4	1	7	0	0	46	4	10

As can be observed, there is very little difference between the Framed Aspect implementation (Table 10) and the normal AO implementation (Table 8). However, the two metrics values that do alter are significant ones: WOC and CBC. The WOC increases due to the frame commands used in the Framed Aspect increasing the complexity. However, the CBC metric drops by 22%. This is significant since it proves that reusability and flexibility is improved when using Framed Aspects due to a reduction in coupling between the cache aspect and other components. An important point to make when comparing the separation of concerns metrics of the Framed Aspect implementation to the normal Aspect-Oriented implementation is that there is no increase in these values. This indicates that Framed Aspects maintain the same levels of separation of concerns and so do not detract from the advantages gained from using AOP.

As mentioned earlier, an alternative approach to parameterising/generalising code is XVCL [28]. To illustrate the differences between XVCL and Framed Aspects, we have implemented the cache concern using XVCL. We have applied XVCL to the OO cache implementation, as XVCL has not been specifically designed for use with an AOP approach and so it would be inappropriate to apply XVCL to the AO cache. Table 11 shows the results from applying the metric to the XVCL code.

The most obvious change with the XVCL implementation is the increase in the number of components: a 200% increase compared to the AOP implementations and a 50% increase compared to the standard OO implementation. This increase was due to the introduction of a Cache_Hook module to separate the tests performed on the cache to determine whether the requested resource has been cached from the base code. This separation allowed the cache concern to be omitted when necessary, like the AO version of the cache.

Table 11. Metrics results from an XVCL OO cache implementation

Resource	CDC	CDO	CDLOC	CBC	DOIH	LCOO	LOC	NOA	WOC
Base_XVCL				2	0	0	16	1	4
Cache_Hook				2	0	0	15	1	4
Cache_XVCL				1	0	0	39	3	13
XVCL total	3	6	12	5	0	0	70	5	21

As can be seen, the other separation of concerns metrics (CDO and CDLOC) are much higher compared to any other implementation. Although the Cache_Hook component was introduced to improve separation, this is only partially achieved since XVCL requires commands to be added to components to identify where the concern needs to be inserted. This is achieved in Framed Aspects with the use of pointcuts and the weaving process, so there is a complete separation between the base-code and the concern to be applied. As expected, the metrics show that XVCL does not separate crosscutting concerns as well as Framed Aspects. However, comparing the CBC metric with all the previous results, the XVCL implementation reduces the coupling the most. However, the CBC metric found in the AOP and Framed Aspect metrics are artificially high due to them requiring extra components to implement the aspects and to retrieve the parameter values passed to the advised method. The coupling to these components does not reduce the reusability or flexibility. As can be observed, the XVCL implementation significantly increases both the lines of code (a 52% increase compared to Framed Aspects) and weighted operations per component metrics (a 110% increase compared to Framed Aspects). This is due to the increased number of XVCL commands required to integrate the cache concern to the appropriate points in the base-code.

From these metric results we can conclude that Framed Aspects do contribute to improving the reusability and flexibility. However, it is also clear that the maintainability does suffer when using Framed Aspects. The effects of this can be limited by providing appropriate tool support to aid the user manage the Framed Aspects. Such tool support is described in Sect. 6.

The results presented also show that Framed Aspects are a more suitable solution for our needs than XVCL. Framed Aspects improve the separation of concerns, and reduce the complexity and the lines of code required to implement a concern when compared to XVCL.

Additionally, we can also conclude from the adaptations used in our case-studies that AOP is a suitable programming paradigm to implement an auto-adaptive system. In specific cases when AOP may not be the most optimum paradigm, standard HotSwap can be used to allow the target system to be adapted whilst still using OO.

5 Related Work

This section will compare a number of related AOP based auto-adaptive systems with our framework to illustrate the differences and benefits of our implementation. To recall, our overall aim was to implement a framework that can make a system adaptable to suit its current state. This was then divided into a number of subaims, a set of which are used as a guide for comparing our work with other related work:

– easily add dynamic adaptive behaviour to existing systems;
– be able to extend/modify adaptations;
– create flexible and reusable adaptations;
– ensure correctness and compatibility between the aspects and the system.

As mentioned in Sect. 2.1, a number of approaches have been implemented that use AOP to encapsulate the *adaptation process*. One of these approaches [19] uses the fractal component model to provide customization. However, since AspectJ is used to identify adaptation points within the system, this prevents new adaptation points being added once the system has been started and limits the dynamic behaviour. Hence, this approach cannot be accurately compared with our work.

A notable piece of related work is JAC which will be briefly described below. However, we focus mainly on Quality Objects (QuO) and Prose based implementations, since these systems are the most closely related to our work and implement similar features.

5.1 JAC

JAC [32] is a dAOP approach that allows the parameterisation of aspects. JAC enables parameters to be passed into the aspects using configuration files; this allows the aspect to be customised for a particular scenario. Unlike our solution, JAC does not allow complete parameterisation (i.e., Framed Aspects allow any element within the code base to be parameterised) and so the aspects cannot be completely customised.

JAC also provides an interesting feature known as Wrapping Controllers, which allow the programmer to create entities that deal with aspect relationships. This gives the programmer control over when and which aspects get executed, useful to prevent incompatible aspects from being executed together. The specification to implement these relationships is done at a low-level which increases the complexity of this definition process. In comparison, the relationships in our approach are defined at a higher-level and so can ease the definition of such constraints.

5.2 QuO

QuO [3] is a domain specific implementation focusing on adapting systems that are implemented using middleware. The aim of this work is to improve Quality of Service (QoS) provided by such systems by applying adaptations to the system. QuO perceives QoS as an aspect and weaves the aspects that alter the QoS properties of the system into the boundary between the middleware and the application. The aspects are defined in QuO's own Aspect Specification Language (ASL), which employs a joinpoint model consistent with traditional AOP approaches. QuO also implements its own Contract Definition Language (CDL) to assign conditions to 'regions' of code which are used to specify when advice should be woven.

A limitation of this is that QuO requires the various definition files all to be compiled to generate the adapt-ready application before it is executed. In comparison, our framework allows true dynamic adaptations. Any change made to the policies at run time will result in aspects being removed and new aspects woven to reflect the changes made. Changes can also be made to the Framed Aspects used; this will affect any future aspects that are generated from the altered Framed Aspects.

QuO's Contract Definition Language specification appears to be highly extensible with few restrictions being placed on the programmer regarding what properties of the system can be monitored. Similarly our approach allows the programmer to extend the attributes monitored, with a number of hardware resources and system attributes available to be monitored by default.

A feature in QuO currently lacking is the ability to customise the code via parameterisation. This would improve the reuse of the code and allow the code to be customised for a particular system/scenario. The use of Framed Aspects allows any part of the aspects used in our framework to be generalised; this is an extremely powerful feature that makes the aspects completely customisable.

QuO allows the order of advice execution to be specified via an aspect precedence definition. However, there is no way of specifying any other more complex relationships that may exist between the aspects used. In contrast, our approach allows dependencies and relationships to be specified in four flexible ways. Incompatible relationships, required relationships, priorities of policies/aspects (either explicitly specified or assigned relative to another policy) and combinations of woven aspects to trigger the weaving of a resolution aspect can all be provided.

5.3 Prose

A large amount of research has been performed using Prose [9] to implement adaptable systems. The majority of this work has involved the adaptation of networked/mobile systems. MIDAS [7] is one example of a system implemented using Prose. The aim of MIDAS is to allow the distribution of Prose aspects to mobile nodes and alter the node behaviour to suit the current location. As Prose is a dAOP technique the aspects can be woven at run time to achieve dynamic adaptation.

Another system that also relies on Prose is Java Adaptive components for Dynamic Distributed Architectures (JADDA) [16]. Although this solution is component based and mainly achieves dynamic adaptation by altering the architecture by introducing new XML Architecture Definition Language (xADL) [48] files, it still relies on Prose for some functionality to fully configure the system. JADDA allows the basic parameterisation of Prose aspects by placing them into an aspect template and generating concrete aspects when required. String values defined in the xADL file are used to complete the aspect template to generate a concrete aspect.

Jadabs [49] is a dynamic lightweight container for small mobile devices which again makes use of Prose to allow the behaviour of a component to be altered at run time. However, when Jadabs is running on resource constrained device, the heavyweight nature of Prose makes it unsuitable; Nanning [50], a more light-weight but less functional AOP technique, is used in these conditions.

As Prose allows dynamic behaviour, these related systems (MIDAS, JADDA, and Jadabs) all permit new aspects and pointcuts to be introduced dynamically at run time and have an immediate effect on the target system. This enables new aspects to be created whilst the target system is executing and then woven dynamically.

Prose has been heavily used for adapting distributed system behaviour according to context information based on the location of nodes. This is comparable to our approach in that when nodes change, location aspects are either woven or removed to alter their behaviour. However, our approach is more flexible in that our framework can be applied to any type of system not just distributed systems. Also, multiple attributes can be monitored and combined to form complex requirement/behaviour specifications. Our approach can also be easily extended to monitor other new system attributes by adding to the framed monitoring aspects.

The parameterisation used with Prose/JADDA is very basic with only a small number of elements parameterisable, such as the classname and method name. An additional limitation in the JADDA approach is that conditional compilation is not implemented. Framed Aspects allow any element within the code to be parameterised allowing much more flexibility and customization.

Finally, with the exception of priorities to allow advice ordering, there is no explicit support within Prose for the definition of relationships between aspects. We believe that the ability to define complex relationships is a very important feature to support when implementing an auto-adaptive system. This was highlighted in Sect. 3.6, where relationships were used to handle some of the undesirable interaction problems that can occur in auto-adaptive systems.

6 Future Work

The core implementation of the auto-adaptive framework is now complete with several extensions currently in progress to improve the functionality of the framework, as discussed below.

We believe the use of a GUI tool will help improve the creation/management of the policy files and the Framed Aspects. Currently, the policies are relatively easy to create although it is recognized that they may become difficult to manage when a large number of them are present. Also, errors can be easily introduced when specifying the policies. Tool-based support will aid the management of the policies by presenting the policies in a more manageable format to the user. Additionally, the tool will be able to restrict the values assigned to certain attributes within the policies and perform checks to ensure the policies specified are correct with respect to the target system (i.e., the attributes to monitor exist) prior to run time.

One of the current problems of Framed Aspects is the readability of the code. As the Framed code has XML frame commands interspersed amongst standard Java code, the readability as a whole is affected. By implementing a GUI these commands could be formatted to improve readability of this combination of Java code and XML. This tool could also be used to check for any incompatibility problems that may occur when the policy and Framed Aspects are applied to the system.

Currently no semantic and type checking is performed to ensure type-safe aspects are generated from the Framed code. This problem is limited slightly in our approach as the specifications (used to generate the concrete aspects) are created programmatically based on information gathered by the framework. Whilst this reduces the potential for errors, some checks are still desirable to eliminate other problems.

A limitation of the current implementation is that the framework can only control the behaviour of a system local to the machine where the framework is executing. For example, if the behaviour of a node gets adapted in a particular fashion, it may be necessary to adapt other nodes with which it communicates in a similar way to ensure they remain compatible. This type of behaviour would require a distributed policy to be created/modified to ensure a set of nodes all followed certain behaviour and all remained compatible when one of the nodes was adapted.

7 Conclusion

The overall goal of this work has been to develop a flexible framework capable of adding auto-adaptive behaviour to existing systems. The solution we have proposed involves policies, Framed Aspects, dynamic AOP and reflection. By adopting these four technologies in combination, our framework provides good support for various different properties of auto-adaptive systems, properties that are not currently supported in existing systems. The use of Frame technology, reflection and dynamic AOP enables the dynamic weaving of code that has been specifically generated based on the target system's current status. Furthermore, the inclusion of ECA rules in our framework allows the explicit control of *when* and *how* the generated aspects should be woven, a feature not available in current dynamic AOP techniques.

We believe we have achieved our goal and that our framework represents an advance in this research area with respect to other systems. The combination and diverse properties of the various technologies used has allowed a framework to be created that provides highly dynamic and flexible behaviour. Our framework can be applied to any existing system with no implementation changes necessary to the target system. However, an understanding of the existing system is required so that the correct adaptations and behaviour can be implemented.

As shown in our evaluation results, the use of Framed Aspects gives high levels of flexibility and reuse in the adaptations being applied to the system. The aspects generated can be customised to suit a particular scenario due to the generalised aspect code.

As our framework generates the concrete aspects dynamically while the system is running, any changes made to the Framed Aspects will be reflected in the aspects woven to the system. Also, our approach allows changes to be made to the policies at run time that will also be reflected immediately in the system behaviour. These techniques allow true dynamic behaviour in that the adaptations themselves can be altered and these changes will take immediate effect. By delaying the processing of the Framed Aspects to run time and generating the concrete aspects using run time information, we have illustrated a new novel use of Frame technology. This dynamic behaviour adds an overhead to the execution of the target system but our results show that this overhead is acceptable compared to the benefits gained from such features.

The problem of unanticipated aspect interaction has also been discussed and perhaps is more critical in our work due to the fact that aspects will be woven automatically and dynamically. To help to eliminate this problem, the elements in the policy file allow relationships to be specified. Requires relationships, incompatible-with relationships and priorities can all be defined to facilitate the correct operation of the system.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J-M. Longtier, J. Irwin. Aspect-oriented programming. In: *Proceedings European Conference on Object-Oriented Programming*. Jyväskylä, Springer; Finland, pp. 220–242, 1997

[2] D. Gupta. A formal framework for on-line software version change. *IEEE Transaction on Software Engineering*, 22(2):120–131, 1996

[3] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, J. Zinky. Building adaptive distributed applications with middleware and aspects. In: *Third International Conference on Aspect-Oriented Software Development*. ACM, Lancaster, pp. 66–73, 2004

[4] P. Falcarin, G. Alonso. Software architecture evolution through dynamic AOP. In: *First European Workshop on Software Architectures (EWSA), co-located with ICSE 2004*. Springer, St Andrews, Scotland, pp. 57–73, 2004

[5] P. Greenwood, L. Blair. Policies for an AOP based auto-adaptive framework. In: *NODe Conference in Conjunction with Net.ObjectDays*. Lecture Notes in Informatics (LNI), Erfurt, Germany, pp. 76–93, 2005

[6] R. Hirschfeld, K. Kawamura. Dynamic service adaptation. *ICDCSW*, 2:290–297, 2004

[7] A. Popovici, T. Gross, G. Alonso. A proactive middleware platform for mobile computing. In: *ACM/IFIP/USENIX International Middleware Conference*. ACM, Rio de Janeiro, pp. 455–473, 2003

[8] I. Warren. A Model for dynamic configuration which preserves application integrity, in computing. Lancaster University; Lancaster, 2000

[9] A. Popovici, T. Gross, G. Alonso. Dynamic weaving for aspect-oriented programming. In: *First International Conference on Aspect-Oriented Software Development*. ACM, Enschede, pp. 141–147, 2002

[10] E.C. Lupu, M. Sloman. Conflicts in policy based distributed systems management. *IEEE Transaction on Software Engineering*, 25(6):852–869, 1999

[11] P. Basset. Framing software reuse—lessons from Real World. Yourdon Press Prentice Hall, 1997

[12] F.-N. Demers, J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In: *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*. pp. 29–38, 1995

[13] J. Hillman, I. Warren. An open framework for dynamic adaptation. In: *Sixth International Conference on Software Engineering (ICSE 2004)*. IEEE Computer Society, Edinburgh, pp. 594–603, 2004

[14] R.S. Moreira, G.S. Blair, E. Carra. Supporting adaptable distributed systems with FORMAware. In: *Fourth International Workshop on Distributed Auto-adaptive Reconfigurable Systems (DARES 2004) - ICDCS 2004*. Tokyo, pp. 320–325, 2004

[15] H. Duran-Limon, G.S. Blair, G. Coulson, Adaptive resource management in middleware: a survey. *IEEE Distributed Systems Online*, 5(7):1, 2004

[16] P. Falacarin, G. Alonso. Software architecture evolution through dynamic AOP. In: *First European Workshop on Software Architectures (EWSA), co-located with ICSE 2004*. Springer, St Andrews, 2004

[17] G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, N. Parlavantzas. Reflection, self-awareness and self-healing in OpenORB. In: *Proceedings of the First Workshop on Self-healing Systems*. ACM, Charleston, pp. 9–14, 2002

[18] C. Szyperski. Component software: beyond object-oriented programming. Addison-Wesley, Reading, 1998

[19] P. David, T. Ledoux. Towards a framework for self-adaptive component-based applications. In: *Fourth IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, Paris, pp. 1–14, 2003

[20] Z. Yang. An aspect-oriented approach to dynamic adaptation. In: *Proceedings of the First Workshop on Self-healing Systems*. ACM, Charleston, pp. 85–92, 2002

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. An overview of AspectJ. In: *ECOOP 01*. pp. 327–353, 2001

[22] J. Keeney, V. Cahill. *Chisel:* a policy-driven, context-aware, dynamic adaptation framework. In: *Workshop on Policies for Distributed Systems and Networks*. IEEE Computer Society, pp. 3–14, 2003

[23] O. Diaz, A. Jaime. EXACT: an extensible approach to active object-oriented databases. *VLDB*, 6(4):282–295, 1997

[24] R. Filman, T. Elrad, S. Clarke, M. Aksit. Aspect-oriented software development. Addison-Wesley, Reading, 2004

[25] R. Douence, G. Muller, M. Sudholt. A framework for the detection and resolution of aspect interactions. In: *GPCE 2002*. Pittsburgh, pp. 173–188, 2002

[26] J. Van Gurp, J. Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002

[27] N. Loughran, A. Rashid. Framed aspects: supporting variability and configurability for AOP. In: *Eighth International Conference on Software Reuse*. Madrid, pp. 127–140, 2004

[28] XVCL, XVCL Homepage. (Web Site) http://fxvcl.sourceforge.net. Cited 2005

[29] E. Hilsdale, J. Hugunin. Advice weaving in AspectJ. In: *Third International Conference on Aspect-Oriented Software Development (AOSD)*. Lancaster, pp. 26–35, 2004

[30] A. Vasseur. Java dynamic AOP and runtime weaving—how does AspectWerkz address it? In: *Dynamic Aspects Workshop held in conjunction with AOSD 2004*. Lancaster, pp. 135–145, 2004

[31] G. Coulson, P. Grace, G. Blair, L. Mathy, D. Duce, C. Cooprt, W.K. Yeung, W. Cai. Towards a component-based middleware framework for configurable and reconfigurable grid computing. In: *Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE Computer Society, pp. 291–296, 2004

[32] R. Pawlak, L. Seinturier, L. Duchien, G. Florin. JAC: a flexible solution for aspect-oriented programming in Java. In: *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. Springer, Kyoto, pp. 1–24, 2001

[33] B. Burke. JBoss tutorial. In: *Third International Conference on Aspect-Oriented Software Development*. Lancaster, pp. 1, 2004

[34] P. Greenwood, L. Blair. Using dynamic AOP to implement an autonomic system. In: *Dynamic Aspects Workshop*. Lancaster, pp. 76–88, 2004

[35] P. Maes. Concepts and experiments in computational reflection. In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. ACM, Orlando, pp. 146–155, 1987

[36] M. Dmitriev. *Safe Class and Data Evolution in Large and Longed-Live Java Applications*. Glasgow, 2001

[37] K. Cheverst, N. Davies, K. Mitchell, A. Friday, C. Efstratiou. Developing a context-aware electronic tourist guide: some issues and experiences. In: *Proceedings of CHI 2000*. Netherlands, pp. 17–24, 2000

[38] J. Pang, L. Blair. An adaptive run time manager for the dynamic integration and interaction resolution of feature. In: *22nd International Conference on Distributed Computing Systems Workshops*. IEEE Computer Society, Vienna, pp. 445–450, 2002

[39] F. Sanen, E. Truyen, B. De Win, W. Joosen, N. Loughran, G. Coulson, A. Rashid, A. Nedos, A. Jackson, S. Clarke. Study on interaction issues. Katholieke Universiteit Leuven: Leuven, 2006

[40] AWBench. AWBench: AOP benchmark. (Web Site) 2004 http://docs.codehaus.org/display/AW/AOP+Benchmark. Cited April 2005

[41] S. Chidamber, C. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994

[42] N. Fenton, S. Pfleeger. Software metrics: a rigorous and practical approach. PWS, London, 1997

[43] A.F. Garcia, U. Kuleska, S. Sant'Anna, C. Chavez, C. Lucena. Aspects in agent-oriented software engineering: lessons learned. In: *Sixth International Workshop on Agent-Oriented Software Engineering*. Springer, Utrecht, pp. 25–37, 2005

[44] J. Hannemann, G. Kiczales. Design pattern implementation in Java and AspectJ. In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, Seattle, pp. 161–173, 2002

[45] A.F. Garcia. From objects to agents: an aspect-oriented approach. In: *Computer Science Department*. PUC-Rio; Rio de Janeiro. pp. 319, 2004

[46] C. Sant' Anna, A. Garcia, C. Chavez, C. Lucena, A. von Staa. On the reuse and maintenance of aspect-oriented software: an assessment framework. In: *Brazilian Symposium on Software Engineering*. Manaus, pp. 19–34, 2003

[47] A. Garcia, S. Sant'Anna, E. Figueiredo, U. Kuleska, C. Lucena, A. Von Staa. Modularizing design patterns with aspects: a quantative study. In: *Fourth International Conference on Aspect-Oriented Software Development (AOSD)*. Chicago, pp. 3–14, 2005

[48] E. Dashofy. A highly-extensible XML based architecture description language. In: *WICSA*. pp. 103–112, 2001

[49] A. Frei, G. Alonso. A dynamic lightweight platform for ad-hoc infrastructures. In: *Third IEEE International Conference on Pervasive Computing and Communications*. Kauai Island, pp. 373–382, 2005

[50] Nanning. Nanning AOP. http://nanning.codehaus.org/. Cited 2005

Editorial: AOP Systems, Software and Middleware

Aspect-oriented software development (AOSD) has grown into a well-established field within the software engineering community. AOSD is mainly concerned with developing and applying modularization techniques for the management of crosscutting concerns in large-scale applications. Though AOSD and, in particular, techniques for aspect-oriented programming (AOP) have been regularly introduced at the application level, their usefulness at the middleware and systems programming level is rapidly proving essential.

Middleware infrastructures and system-level software—including industrial component platforms, operating systems, and networking abstractions—are rich targets for applying aspect orientation (AO) for several reasons. First, as basic infrastructures on which higher-level applications are built, they are typically subject to many different crosscutting concerns. Second, as a main transparency mechanism for shielding the applications from intricate hardware details over various different underlying platforms, systems software tends to require a great deal of customization. Often, this customization is particularly amenable to the application of AO. Furthermore, the degree of customization achieved in this manner can easily give rise to a software product line allowing the combination and selection of features almost *à la carte*. Third, modularization of crosscutting concerns in the context of systems software frequently requires code bases of tens of thousands of lines of code to be handled, which, without adequate methodological and programmatic support, would be a daunting endeavour. Finally, middleware and systems software are subject to the imperative of optimal performance: this forbids the use of many modularization techniques which are appropriate for applications geared towards end users. For these reasons, results on AO for infrastructure software are now regularly published in the realm of relevant international software engineering and systems conferences and workshops, such as the AOSD and Middleware conferences, ACM SIGOPS sponsored events, and the series of workshops on hot topics in OS (HotOS).

Overview of articles and the evaluation process. In this special section, six articles are presented which deepen and extend previous work of their respective authors. Three of these articles treat AOP techniques for middleware infrastructures, while the three remaining ones apply AOP to crosscutting concerns at the system level. In the following, we briefly present the main issues of the various articles and position them in their respective research contexts.

The first group of three articles explore AOP techniques for middleware infrastructures:

Aspect-Oriented Development of Crosscutting Features in Distributed, Heterogeneous Systems by *Eric Wohlstadter and Premkumar Devanbu*

This paper presents the *Dado* approach to modularize crosscutting concerns in heterogeneous component-based environments. It presents the notion of *adaptlets* that provide means to apply aspects to CORBA objects based on flexible-typed interactions modifying stubs and skeletons.

Shakeins: Non-Intrusive Aspects for Middleware Frameworks by *Tal Cohen and Joseph (Yossi) Gil*

Tal and Gil's article focuses on a different method to integrate AO support with an existing middleware infrastructure (here the J2EE platform): *shakeins* link aspects to classes uses an inheritance-based approach, similar to mixins. The resulting system, *AspectJ2EE*, allows the application of aspects to existing applications at deployment time.

Run-time and Atomic Weaving of Distributed Aspects by *Eddy Truyen and Wouter Joosen*

The last article in this group focuses on the important problem of consistently applying dynamic aspects to an executing distributed system. To this end, the authors present how the *Lasagne* system ensures such a property by a two-phase protocol: aspects are first deployed at runtime and then activated.

Three articles treat AOP techniques for system-level applications:

TOSKANA: A Toolkit for Operating System Kernel Aspects with Nice Applications by *Michael Engel and Bernd Freisleben*

Engel and Freisleben explore the use of dynamic AO techniques to modularize crosscutting concerns in operating systems. Their *TOSKANA* toolkit enables runtime application of aspects to the NetBSD kernel.

Lean And Efficient System Software Product Lines: Where Aspects Beat Objects by *Daniel Lohmann, Olaf Spinczyk, Wolfgang Schröder-Preikschat*

This article presents an application of AOP techniques to the important application domain of software product lines. Specifically, the *Aspect-C++* system is used to improve resource efficiency in a product line of weather stations.

Providing Configurable QoS Management in Real-Time Systems with QoS Aspect Packages by *Alekšandra Tesanović, Mehdi Amirijoo, and Jörgen Hansson*

The last paper in this special section presents how AOP techniques can be used to modularize quality-of-service properties, which typically crosscut existing applications, using aspects. Furthermore, the authors present an application of a corresponding aspect-based infrastructure to the real-time database system COMET.

To conclude some remarks on the evaluation process. The six articles presented here have been selected out of ten original submissions each by means of an evaluation by at least three reviewers; all have been revised two or three times over a period of 9 months.

July 2006 Yvonne Coady, University of Victoria, Victoria, Canada
Hans-Arno Jacobsen, University of Toronto, Toronto, Canada
Mario Südholt, INRIA—École des Mines de Nantes, Nantes, France
Associated editors

Aspect-Oriented Development of Crosscutting Features in Distributed, Heterogeneous Systems

Eric Wohlstadter[1] and Premkumar Devanbu[2]

[1] University of British Columbia, Vancouver BC, Canada
wohlstad@cs.ubc.ca
[2] University of California, Davis CA, USA
devanbu@cs.ucdavis.edu

Abstract. Some "non-" or "extra-functional" features, such as reliability, security, and tracing, defy modularization mechanisms in programming languages. This makes such features hard to design, implement, and maintain. Implementing such features within a single platform, using a single language, is hard enough. With distributed, heterogeneous systems, these features induce complex implementations which crosscut different languages, OSs, and hardware platforms, while still needing to share data and events. Worse still, the precise requirements for such features are often locality-dependent and discovered late (*e.g.,* security policies). This paper describes an interface description language, Dado, to help program crosscutting features in CORBA based middleware software through an aspect-oriented programming. A Dado specification comprises pairs of *adaptlets* which are explicitly modeled in an extended version of CORBA IDL. Dado supports flexible and type-checked interactions (using generated stubs and skeletons) between adaptlets and between objects and adaptlets. Dado also supports co-ordinated, run-time binding of adaptlet-based features to applications in a distributed setting.

1 Introduction

This paper addresses the problem of supporting the development of late-bound, crosscutting features in distributed heterogeneous systems.

Crosscutting features stubbornly resist confinement within the bounds of modules. It is well known that features such as logging, transactions, security, and fault-tolerance typically have implementations that straddle module boundaries even within the most sensible decompositions of systems [1, 2, 3, 4]; we present a sample security feature in the next section which provides yet another example. The scattered implementation of such features makes them difficult to develop, understand, and maintain. To worsen matters, the requirements of such features are often *late bound*. As a specific example, access-control (security) policies may vary with each installation site, even for the same software. Some localities may require client identification for the provision of certain types of services, others may not. These types of feature requirements are locality dependent, discovered late, and change often. It is highly desirable to accommodate these requirements without requiring programmers to undertake the difficult challenge of making a scattered set of changes to a broad set of modules, often late in the game.

Distributed heterogeneous systems (abbreviated DH) are part of the IT infrastructure in many organizations: many needed software functions are provided by systems assembled from pieces running on different platforms and programmed in different languages. Distribution arises from pressures such as globalization and mobility. Heterogeneity arises from considerations such as performance, legacy systems, weight, size, vendor specialization, and energy consumption. In this context, crosscutting feature implementations are scattered across different languages, operating systems, and hardware platforms. Crosscutting feature implementation elements in one platform need to correctly exchange information with existing application code, and with such elements on other platforms. In a WAN context, the presence of different, incompatible features (e.g., different security policies) may even cause the application to fail. Some platforms may be too resource-limited or performance-constrained to support some types of software evolution techniques (e.g., reflection). In some cases, source code may not be available for modification, so middleware-based wrapping might have to be used. However, since feature implementations may crosscut platforms, all these different techniques of software evolution should be allowed to co-exist, and interoperate. Traditional aspect-oriented approaches deal with crosscutting features by encapsulating a broad set of scattered and tangled implementation fragments into a single aspectual module. However, they are less useful in this setting:

- The systems are distributed, and so are the changes required to implement the features. However, these *distributed feature-implementation elements need to exchange pertinent information*. This exchange must not unduly impact performance, due to additional, distributed message exchange between remote sites.
- The systems are heterogeneous, so scattered and tangled elements may need to be incorporated into different languages, operating systems, etc., in a *consistent* manner.
- Crosscutting features in this setting are late bound. This raises the important issue of *co-ordination*. A feature introduced in one distributed application element must be properly synchronized with a corresponding feature in a partner application.

In this paper, we describe Dado, an approach to developing features that require code changes, in a DH setting, to both client- and server-side of a remote interaction. We are specifically interested in these types of *end-to-end* aspects. We refer to the matched client–server components that implement these types of aspects as *adaptlets*. Adaptlets are given first class status in the middleware development process. The specification of adaptlets is separated into a language independent interface definition; platform specific compilers can generate transformations through a variety of techniques. Adaptlets support distributed information exchange between adaptlet elements; they also support co-ordinated late-binding between clients and servers. Adaptlets are thus a reformulation of the aspect-oriented programming paradigm to deal with special problems that arise in the heterogeneous, distributed setting.

We expand upon an approach previously published in a related conference paper [5]. New case studies are presented in Sects. 6 and 7 using new extensions (such as generic type parameters) to our previous approach; additional performance measurements and technical details are included. A new adaptlet inheritance mechanism, inspired by mixin-layers [6] is also presented.

The paper begins with a motivating example in Sect. 2. We then survey the broad technical area in Sect. 3. Section 4 presents our research goals in more detail. Section 5 describes the current status of our Dado implementation (including the run-time, code-generation, and deployment tools), which is based on the OMG CORBA standard. Section 6 lays out the details of the motivating example in a concrete case study. Section 7 provides an additional example with some extensions to Dado. Section 7 describes closely related projects. Finally we conclude with an overall view of the work, the current limitations, and our future plans.

2 Security Example

We begin our example with the description of an application, The Remote Collaboration Tool (RCT) [7], where a new security concern must be addressed. RCT is a distributed multi-user interactive environment that supports collaborative activities in teaching and research settings. The architecture is based on client–server remote procedure-call (RPC) through the CORBA object-oriented middleware. Users on the RCT can self-organize into different groups to engage in private chat-room sessions or file-sharing. Groups can be used to support classroom settings where each student is a member of a group for each registered class. Multimedia functionality such as interactive voice and collaborative workspaces is supported.

Some components of the RCT are prone to denial-of-service (DoS) attacks because of the amount of computation required by the components. A DoS attack occurs when a malicious client (or set of malicious clients) overloads a service with computationally expensive requests, hindering timely response to legitimate clients. When deploying RCT in an environment where some clients may not always be trusted, it is desirable to provide protection from such attacks. Also consider that in certain classroom settings it is not difficult for malicious users to gain unauthorized access. The client puzzle protocol (CPP) [8] is one approach that can be used to reduce the risk of this security problem. CPP protects a component by intercepting client requests and refusing service until the client software provides a solution to a small mathematical problem.

Adding this feature requires changes to many different server components in the RCT implementation and also to the client. In a case study (Sect. 6), we identified 76 out of 82 operations in RCT where this feature would apply. The changes are crosscutting and programs running on different platforms and in different languages might need changing; the RCT client is written in Java and the server is written in C++ using ORBs from two different CORBA vendors (resp. JacORB and TAO). Since some platforms may have performance or battery limitations (e.g., PDAs or laptops) or be remotely located, different evolution strategies should be allowed and allowed to inter-operate. Changes to different elements must be made consistently to ensure correct interaction. Changes must be properly deployed in the different elements, otherwise versioning errors may result. Since the functions for CPP may apply to other applications, it would be desirable to reuse the same implementation, should the platforms be compatible. This paper presents the details of Dado which is our IDL-centric approach to manage these kinds of crosscutting changes in a disciplined fashion.

```
(1) adaptlets CPP {
(2)    client {
(3)       around void catchPuzzle();
(4)    };
(5)    server {
(6)       around void CPP();
(7)       context puzzleSolution(PuzzleResponse solution);
(8)    };
(9) };
```

Fig. 1. CPP adaptlets

Dado uses an enhanced IDL model to describe these changes, as in Fig. 1. These descriptions include two interfaces, one for an *adaptlet* component to be deployed on a client and one for an adaptlet component deployed on a server. Adaptlets consist of three types of operations: advice (as in AspectJ), request operations (not shown in this example), and context operations.

In the example, the client side has one advice (line 3) used to bracket the execution of an application invocation. Clients receive puzzle challenges through exceptions that can be caught by this code. The server side includes an advice operation `CPP` (line 6) that should be bound to application operations protected by the CPP, the bindings (not shown) can be specified in a separate step. The `context` operation `puzzleSolution` (line 7) allows clients who have been presented with a puzzle challenge to return their solution through a type-checked interface. We return to a more thorough discussion of this example in Sect. 6, after more details of the approach have been presented.

3 Background

There are a variety of approaches to dealing with crosscutting features. Our survey here is limited by space to be representative rather than exhaustive; no judgment of omitted or included work is implied.

Several programming languages support novel ways of encapsulating crosscutting features. Classical syntactic program transforms [9] were perhaps among the earliest to provide the capability of broad changes to programs. Reflection [10] provided means of introducing code changes at run-time in languages such as Smalltalk. Composition Filters [11] enable programmers to cleanly separate functional object behavior from code dealing with object interactions. Compile-time [12, 13] reflection is used in Open C++/Java and had been extended to load-time in Java using bytecode editing [14]. Mixin-layers [15] provide a way of adding features to methods in several different classes simultaneously; our work uses a similar approach to extend matched client–server middleware features. Implicit Context [16] is a method for separating extraneous embedded knowledge (EEK) (or crosscutting knowledge) from the design of a program, and re-weaving it back in later. Monads and monad transformers [17] have been used in lazy, pure functional languages to capture crosscutting features such as states and side-effects. They work by encapsulating the basic notion of a computation, and then allowing fundamental evaluation mechanisms such as value propagation to be overridden.

Languages such as HyperJ [1], AspectJ [2] provide different approaches to implementing crosscutting features in Java. A detailed comparison of these differing approaches is beyond the scope of this paper; suffice to say we are interested in a DH setting, thus transcending language boundaries. Our approach uses the *joinpoint* model of AspectJ including pointcuts and advice. We assume the reader is familiar with this paradigm; more details can be found in [2].

Middleware-based approaches are certainly relevant. Interceptors [18, 19] provide a way of inserting extra functionality into every invocation that originates or arrives at a request broker; middleware-specific APIs provide means for interceptor code to reflect upon the details of the intercepted invocations. Some works use specially constructed reflective ORBs [20, 21]. While these reflective methods are suitable for implementing crosscutting features, (and for some highly dynamic features may be the only way to do it) the use of the low-level reflection APIs, along with the need for frequent use of type-casting makes programming difficult and error-prone; thus it would be preferable to use more statically checkable methods when possible. Proxies and wrappers [22, 23] are another approach. However, they are typically tailored for a specific application object interface; so thus, it would not be possible to reuse a wrapper to implement the same security policy on entirely different components.

Container models [24] address this problem through code generation. They provide a *fixed* set of features (depending on the container vendor) to apply to application components. Via configuration files and code-generation, features selected from a given set can be added to any component. Newer containers such as JBoss [25], AspectJ2EE [26], AspectWerkz [27], Spring [28], and JAC [29] provide aspect-oriented development and deployment of container features. Just-in-Time Middleware [30] provides an automated customization approach to middleware, in the context of traditional CORBA middleware. In contrast to our work, these approaches do not focus on support for matched client–server features by modeling aspects in the IDL.

Section 8 surveys several other closely related works, that are easier to relate to ours after more details have been presented.

4 Research Problems and Approach Overview

As illustrated in Sect. 2, late-bound, crosscutting features such as security require new programming models beyond the ones provided by aspect-oriented programming and middleware such as CORBA. In Dado, we provide these using extra functional elements called adaptlets which are co-located with (potentially distributed) application software components. Adaptlets are inspired by the pointcut and advice style design in AOP languages such as AspectJ. However, the split client–server and heterogeneous language context of adaptlets differentiates the approach from standard AspectJ aspects. Before going into the full technical details of Dado we begin with a motivating discussion of the main goals of our project. Each sub-section begins with a description of a problem we sought to address, followed by a brief outline of our approach with a reference to the text (in Sect. 5) that describes the approach in detail.

4.1 Heterogeneity

The precise means of incorporating adaptlets into the application execution (such as source/binary transformation or runtime wrapping) may depend on the platform; even so, heterogeneous adaptlets should co-exist and inter-operate correctly. In CORBA, this heterogeneity is managed through IDL-based tools to generate communication code, as well as a run-time ORB platform; we take a similar approach for adaptlets by extending the IDL.

Dado employs an enhanced IDL and code-generation to support the following:

- Explicit interface-level modeling of adaptlets and their interaction with application components.
- Ability to implement adaptlets in different languages, while supporting:
- safer interaction (via static type-checking) between adaptlets, with automated generation of marshaling and instrumentation code. An extension to support parametric types is presented in Sect. 7.

More details can be found in Sect. 5.2.

4.2 Customized Deployment

Features such as security, fault-tolerance, billing etc. are highly location-dependent. We therefore must be able to support *late binding* and *flexible deployment* of Dado adaptlets. Consider that container standards such as J2EE allow independent container developers to develop features that are customized for specific applications at deployment time. Likewise, we would like to allow vendors to build crosscutting features (based on adaptlets) independently of application builders, and then allow deployment experts to combine them with applications to suit their needs. Here we also seek an abstract, platform-independent way of specifying binding and deployment.

Dado allows separation of the adaptlet interface definition from a deployment description, which specifies the precise deployment context of adaptlets (using a pointcut language and inheritance mechanism similar to AspectJ). This allows a deployment expert to tune the connection between adaptlets and different application components. Details can be found in Sect. 5.3.

4.3 Adaptlet Communication

The distributed pair of adaptlets comprising a crosscutting feature implementation (in a client–server setting) may need to communicate with each other. For example, the client side may send credentials to a server side (for a security feature). The server side may send the client the cost accrued for providing a particular service, or send the client side information on the remaining balance of pre-paid credit. As another example, the server side may require the client side to cache a response, and send back a time-to-live (TTL) value. However, application IDL interfaces may have already been fixed, so adaptlets would need to resort to communicating through dynamically typed data if additional IDL support were not provided.

Dado allows adaptlets on the client and server side to communicate via messages. However, rather than inducing additional middleware invocations, multiple messages are piggy-backed within the single pre-existing application invocation. Unlike traditional interceptor-based approaches, these interactions statically-typed in our IDL. More details can be found in Sect. 5.4.

4.4 Dynamic Feature Recognition

Because of run-time conditions, applications may desire to use several different features. For example, in response to increasing load, a server may want to instruct clients to redirect certain requests to other servers for a given period of time. As another example, a server may respond to security conditions (e.g., a DoS attack) by engaging protection mechanisms. In these settings, clients may find that servers have (at run-time) incorporated new features, and must respond by deploying matching feature implementations. In a dynamic, widely distributed context spanning multiple administrative domains, clients may become aware only at run-time of the adaptlets associated with a server object. Thus adaptlets may need to be deployed at runtime.

Dado clients transparently (without additional programming) discover the adaptlets associated with a server, and deploy additional client-side adaptlets as needed. More details can be found in Sect. 5.5.

5 Dado

Currently, the process of building DH systems using middleware such as CORBA begins by specifying the high-level interfaces using IDL. IDL specifications are then implemented by developers on different platforms and perhaps in different languages. When implementation is complete, the users of the distributed system can run ORBs on a network as suited to the application and organizational needs, and deploy the constituent application objects.

Dado brings three new roles into this process (see Fig. 2): the *adaptlet architect*, *adaptlet programmer*, and *adaptlet deployment expert*.

An adaptlet architect begins the process by describing an adaptlet (which implements a feature that crosscuts a client and server) in an enhanced IDL, the Dado Description Language (DDL). The interface describes a matched pair of client/server interfaces, which consist of several operations, just like an IDL interface. These interfaces are then compiled using DDL compilers for a target implementation language (e.g., Java or C++), producing marshaling routines and typing environments. Adaptlet programmers then provide an implementation, consistent with the generated typing environments.

The deployment expert binds an implemented adaptlet to a given application by specifying bindings using a pointcut language. The deployment expert will need to understand both the application and the adaptlet, and select the bindings based on the specific installation. Currently, applications are instrumented in two different ways: through generated AspectJ code (for Java) or generation of middleware wrappers (for C++). A specialized CORBA IDL compiler (the Dado IDL compiler) is provided for this purpose.

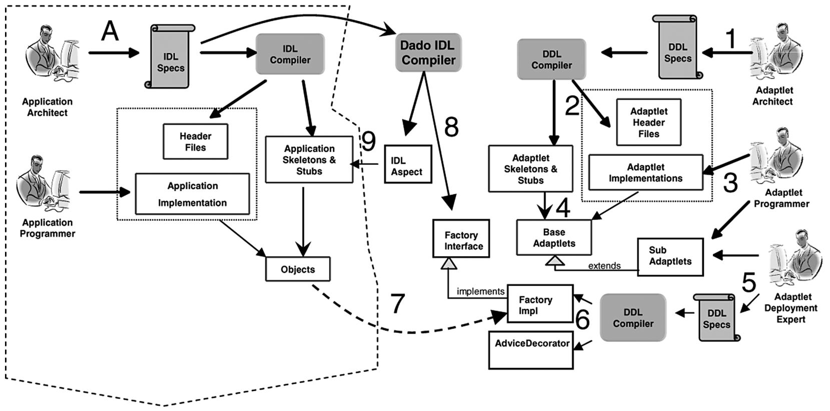


Fig. 2. Dado Development Process. The *left hand side* (within the *dotted lines*) indicates the conventional CORBA process. On the *right*, the adaptlets development begins (*1*) with modeling the interface of Dado adaptlets using DDL; from this the DDL compiler generates (*2*) marshaling code, and typing environments for adaptlet implementations. The programmer writes (*3*) the adaptlet implementations and links to get (*4*) the adaptlets. Now, the deployment expert extends adaptlets using (*5*) pointcuts, and these are used to generate (*6*) a FACTORY for obtaining DECORATOR implementations at run-time (*7*). Applications are instrumented independently by the Dado IDL compiler. An AspectJ aspect or TAO PROXY is generated (*8*) and used to modify the existing application (*9*).

Throughout this section we present the details of Dado using the running example of an invocation timing feature. The current Dado implementation includes the DDL language for adaptlets, a DDL compiler for C++ and Java, and the Dado IDL compiler to modify applications for two different open-source ORBs (JacORB [31] and the TAO [32] ORB).

5.1 Timing Example

In this section we use the running example of a remote invocation timing monitor. This simple example is used to ground the discussion of newly introduced concepts; we return to the security example in Sect. 6. Here we discuss the high-level requirements for the timing implementation.

One could write code (e.g., using interceptors [19, 33] or aspects) to attach to the client that will record the time each invocation leaves and each response arrives. However, the client may also want the invocation arrival-time at the server, and the reply sending-time in order to compute the actual processing time. This scenario demands more coordination between interacting client and server components.

This section, through the example, demonstrates how Dado provides four critical elements. First, clients must be able to ask the server for timing statistics. Only some clients will request this feature; others may not. Timing measurements will only be taken and returned when requested. Second, servers may return data through a type-checked interface. Third, clients need some way to modify existing software to add logic for requesting timing statistics; different means should be allowed. Finally, client and server adaptations should be coordinated; clients will not request timing statistics from servers unable to provide them.

5.2 Dado Description Language

Dado adopts the philosophy that IDL-level models provide an excellent software engineering methodology for distributed systems. In addition to promoting better conceptualization of the design, one can construct tools to generate useful marshaling code and typing environments for static type-checking.

A client and server adaptlet can be described together in a shared scope by declaring both inside the `adaptlets` element. This scope can be viewed as a role-based collaboration [6] of client and server adaptlets. The client and server adaptlets described in this way share a common name, used to coordinate run-time deployment. The scope also provides a way to coordinate client and server adaptlets at the interface level. This may be needed at deployment time, through pointcuts and advice, and at run-time, through `request` and `context` messages.

Adaptlet Inheritance. Adaptlet interfaces can be extended through object-oriented multiple interface inheritance. Client adaptlets in a sub-`adaptlets`[1] scope implicitly inherit from the client adaptlet in the super-adaptlets scope; a server adaptlet is similarly extended. This is analogous to the inheritance provided by mixin layers where classes in one layer simultaneously extend from classes in another layer, but specialized for the case of client/server interface definitions.

Inheritance provides for the description of generic adaptlets that are subsequently tailored for specific applications. The architect will write interfaces for client and server adaptlets, which are then implemented by programmers. A deployment expert can then add an adaptlet to an application by extending either the adaptlet interfaces (through DDL multiple inheritance), adaptlet implementations (through Java single inheritance or C++ multiple inheritance), or both. The inheritance model expands the notion of AspectJ's aspect inheritance to client/server adaptlet collaborations.

Adaptlet Operations. Each adaptlet may support several operations, which may be of three different kinds. First, `before` or `after` advice operations are specified as standard CORBA operations (with an implicit void return type) or optionally tagged as `around` (including a specific return type). Advice operations (Sect. 5.3) provide only the signature of an advice; the body is implemented in a programming language binding. Advice operations may be bound via `pointcut` patterns to application interfaces. They basically provide additional code that is run *every time* certain operations defined in an IDL interface are invoked. Code in the distinguished `around` advice have full access to modify intercepted application information, whereas other advice do not (as in AspectJ). Second, adaptlets can also include `request` operations. These are asynchronous events that may be generated by any adaptlet advice causing an operation to be

[1] We use the terms sub-, super-, and base-adaptlets/adaptlet in analogy to sub, super, and base classes.

invoked on a matched adaptlet partner (Sect. 5.3). Third, `context` operations provide similar functionality but must be polled for by an adaptlet partner, rather than triggering operation invocation (Sect. 5.4).

DDL for Timing Example. In Fig. 3, we see the base `client` and `server` Timing adaptlets. Each adaptlet includes one `request` operation.

The operation `timeRequest` (line 8) is a `request` message that can be generated by the client side of the adaptlet. This message is added to the application invocation and dispatched to the server-side adaptlet. The server adaptlet can respond to a `timeRequest` by taking two timing measurements to determine the actual execution time for that application method invocation. It can then report the results back to the client, using the client request `timeResult` (line 3).

In order to trigger this exchange of `request` messages some advice (line 4) must be added to the client side. Later Sect. 5.3 shows how a deployment expert can extend the client adaptlet to include the advice.

DDL Language Mapping. The DDL compiler generates typing environments, as well as stub and skeleton functions. The generated typing environments (C++ header files or Java interfaces) ensure that matched client/server adaptlet implementations can safely inter-operate. Adaptlets can currently be implemented in either C++ or Java but must be written in the same language as the application. This is primarily for performance reasons; if adaptlets are in a different language, it would be necessary to endure an expensive traversal through the layers of the middleware stack to get from an application object to an adaptlet.

```
(1) adaptlets Timing {
(2)    client {
(3)       request timeResult(TimeBase::IntervalT interval);
(4)       timedOperation();
(5)    };
(6)
(7)    server {
(8)       request timeRequest();
(9)    };
(10)};
```

Fig. 3. Timing `client` and `server` adaptlets. Written by Service Architect as in item 1, Fig. 2. Deployment specified in Fig. 5.

Each adaptlet interface is used to generate the appropriate programming language interfaces and the specialized communication code for `request` and `context` operations. These are then used by programmers to build the adaptlet implementations. Figure 4 shows the Java implementation code for the client base-adaptlet.

From the `adaptlets` in Fig. 3, the DDL compiler can generate the interface `Timing_Client`[2] which is referenced in Fig. 4, line 1. The declaration of the

[2] The suffix _Client is appended for a client adaptlet and _Server for a server adaptlet.

```
(1) class Timing_Client_Impl implements Timing_Client {
(2)
(3)    Timing_Server_Stub _that;
(4)    void initialize(Timing_Server_Stub that) {
(5)       _that = that;
(6)    }
(7)    void timedOperation() {
(8)       _that.timingRequest();
(9)    }
(10)   void timeResult(TimeBase.IntervalT tim) {
(11)      System.out.println(time.lower_bound +
(12)                         " " + time.upper_bound);
(13)   }
(14)}
```

Fig. 4. Timing implementation. Written by Service Programmer as in item 3, Fig. 2.

`interface` (not shown, however implied by the implementation) includes three methods.

The first, `initialize`, allows the implementation to obtain a reference to the server side stub of the matched partner. It is important to note that this is not a standard remote object stub but a custom stub generated for `request` and `context` operation semantics. The reference can be used to communicate through operations declared on the server side, such as `timeRequest` (Fig. 3, line 8).

A second operation, `timedOperation` has been added to trigger the timing `requests` for particular client operations. This is achieved on line 8 by invoking the `timeRequest` operation on the provided server adaptlet stub. However, the actual pointcut is left for a deployment expert to fill in later.

The third method defined in the interface is `timeResult` (lines 10–12). This method will be invoked whenever the server-side adaptlet adds a `timeResult request` by using a `client` adaptlet stub. Here, the implementation simply prints out the results. So far, in this section we provided an introduction to the adaptlet interface language called DDL. We saw how the DDL is used to generate interfaces implemented by adaptlet programmers. The implementation could make use of stubs that refer to the matched `client` or `server` adaptlet. The details presented were motivated by three concepts: that adaptlet implementation should be provided a statically typed-environment for communication, that communication details would be automatically generated, and that generic and application specific components could be separated through inheritance.

5.3 Application Instrumentation for Feature Deployment

In order to trigger adaptlet behavior at runtime, application code must somehow be modified, or execution intercepted to capture the right events. A wide range of binary and source-code, static and dynamic instrumentation mechanisms have been reported [2, 14, 34]. Middleware, also, can support highly dynamic reflective mechanisms [35, 36]. We have explored heterogeneity in the implementation of the triggering mechanism; thus, while the pointcut specifies the high-level design of the binding, different implementation strategies are possible. Applications are prepared statically, for adaptation by adaptlets at run-time. This is done in a generic fashion. The preparation steps do not depend on the adaptlets that will be added. Instrumentation takes place on the client and server CORBA stubs and skeletons. During the instantiation of stubs or skeletons, a FACTORY [37] is consulted to obtain a reference to a list of DECORATORS which wrap the stub or skeleton. Each decorator also acts as an ADAPTER from the interface of an application to a particular adaptlet. The generated decorators are hard-coded to know what advice should execute and how to map parameters from the application to the adaptlet operations. Adaptlets can be added and removed at run-time through management of decorators, however in this paper we only focus on the case where clients must add adaptlets to match those of a particular server. Service programmers need not concern themselves with details of how decorators are generated or deployed, they simply implement methods of the generated interface. Likewise, a deployment expert can use interface level pointcuts to provide the binding to an application. Optionally, deployment experts can specialize an adaptlet using standard OO inheritance mechanisms.

For C++, we make use of two mechanisms available in the TAO ORB: the TAO smart proxies [18] and the Object Reference Factory [33]. Similar mechanisms do not currently exist in JacORB, so AspectJ was a natural alternative.

Both the smart proxy and the Object Reference Factory essentially provide the same functionality for our purpose; details vary in the implementation. The smart proxy acts on the client and the Object Reference Factory acts on the server. Both allow stubs or skeletons to be wrapped by decorators. Equivalent code that was implemented in the C++ decorators is implemented as generated AspectJ advice for the Java implementation.

Naturally, client- and server-side adaptlets, even if using different languages or different instrumentation mechanisms, are fully inter-operable. The adaptlet programmer remains agnostic with respect to the actual instrumentation mechanism that is used to trigger the adaptlet.

Binding Advice Operations. Recall that Dado introduces several new crosscutting feature-related roles into the software process: an *adaptlet architect*, *adaptlet programmer*, and an *adaptlet deployment expert*. When an architect decides that some additional behavior on the client or server of a distributed application is desirable, she can add an advice operation to the interface of an adaptlet. The programmer has the obligation to

```
adaptlets TimeStockServer : Timing {
   client {
      before() : call(float StockServer::getQuote(in string))
      -> timedOperation();
   };
};
```

Fig. 5. Timing deploy. Written by Deployment Expert as in item 5, Fig. 2.

implement each advice. The deployment expert can then add the behavior specified by the advice interface to a specific application object by writing an appropriate pointcut. Additionally, the deployment expert can specialize the advice implementation by overriding any advice using a sub-adaptlet.

In the example, to deploy `Timing` adaptlets for a given application object, the server-side would make the feature available by registering a server `Timing` adaptlet. When clients become aware of those server objects, the Dado run-time will automatically deploy client adaptlets based on the client's pointcuts (Fig. 5). Thus, for matched client/server adaptlets each client-side pointcut is implicitly conditional on the deployment of the particular server-side adaptlet for each server object. Depending on which server object an invocation is destined for helps determine whether the pointcut applies. Server adaptlets are only made aware of the existence of a matched client-side adaptlet when a `request` or `context` message is found. This asymmetric setup follows the CORBA distributed object philosophy [33] where client processes do not export any referenceable identity.

In this deployment (Fig. 5) the client would like invocations to the `getQuote` operation of a `StockServer` interface to be intercepted by the advice `timedOperation`. Here, a client deployment expert extends the `Timing` adaptlet interface independently from the server. The server side is unaware of the pointcut binding on lines 3, 4. The server-side does not need to specify any additional pointcut instructions, as the `request` operation `timeRequest` is invoked dynamically by the client-side adaptlet.

Adaptlet Request Operations. Some adaptlets can be implemented simply by executing advice on the client- or server-side. However, in some cases, additional information may need to be sent along from the client to the server-side adaptlet (or vice versa).

In our running example we presented a feature where a client-side adaptlet can request that a matching server adaptlet calculate server processing time for specific invocations, and then communicate this information back to the client adaptlet. This additional information conveyed between client and server adaptlets is contextual. It must be associated with some original CORBA invocation; in other words, it would not make sense to communicate this information "out-of-band" by introducing new operations into the application IDL. So, the timing behavior by the server adaptlet must occur before and after the processing of the invocation for which the client adaptlet requested measurements.

The architect can include operations tagged with the `request` modifier keyword to provide an extra communication path between `client` and `server` adaptlets that is associated with the current CORBA invocation. The body of `client` and `server` advice can be programmed to add `request` messages by using a reference which exposes the interface of request operations available to a client adaptlet by the server adaptlet and vice versa. A client-side `request` acts as a conditional `after` advice that is triggered by a server-side adaptlet, which provides the actual arguments. Likewise, a server-side `request` acts as a conditional `around` advice, triggered by the client-side.

Adaptlet requests give developers more ways of programming interactions between client-side and server-side adaptlets. Consider that a client adaptlet may require different types of actions to be taken at the server side. As a very simple example, a per-use payment adaptlet attached to a server object might accept e-cash payments, or a credit card. Another example is authentication. It could be based on Kerberos-style tokens, or on a simple password. We could include both options as possible parameters, in a single operation signature, along with an extra flag to indicate the active choice; this leads to poorly modularized operations with many arguments. Rather, we take the "distributed object" philosophy of supporting different requests at a single server object; we allow adaptlets on either side to support several different requests.

Adaptlet requests are implemented as one-way, asynchronous "piggy-backed" messages that are sent along with an invocation (from client to server) or a response (vice versa). Since multiple features can be present simultaneously, the requests are queued on each client and packaged with the original invocation. This is implemented using an extensibility element in the CORBA protocol called the Service Context.

Advice and `request` play different roles in adapting the execution of a distributed application. Advice operations are used to add code at points in the program determined by pointcut based deployment. Pointcuts create a connection between client programs and client adaptlets or server objects and server adaptlets only. The connection between client adaptlets and server adaptlets is made through request messages and is completely dynamic. The request messages serve both to convey additional information and invoke behavior to process the information. In essence, request messages provide a form of dynamic per-invocation adaptation [38] while supporting type-checked interactions and modular design through IDL declaration.

5.4 Adaptlet Context Operations

In some circumstances a feature may require extra data to be shared by the client and server adaptlets. Previously, we described how this could be achieved using a `request` message. When a `request` message arrived at a matched adaptlet, some operation was invoked that matched the signature of the message. This provided the semantics we required for conditionally executing timing requests.

In other situations, adaptlet advice will be required that should not be executed conditionally, but still needs to share data with it's partner. For example, consider an access control advice which will check client credentials and throw an exception when encountering an invalid password. In this case (as opposed to the timing example case) it would not make sense for the access control advice to be conditionally executed; *clients do not ask for access control*, they simply provide some data (e.g., a password) to be used by a server-side advice.

For these situations we provide the `context` operation which provides for the flow of context directly between advice. The technical differences between `request` and `context` operations are motivated by differing usages. Whereas a `request` allows an advice to dynamically trigger additional pre- or post-processing code to be executed with a particular RPC, a `context` operation simply allows an advice on one host to share data with an advice on another host.

Instead of being dispatched to a method site, context messages are stored as thread local variables and unmarshaled on-demand. The provided actual arguments of the context operation can be polled for by an advice through a local operation. This operation returns either true or false, indicating whether the message is available. So, in the case of an access control advice on a server, the advice would throw an exception if a password message is not available or is not valid, and allow a procedure call to proceed otherwise. We further show how `context` is useful in the implementation of the CPP server adaptlet (Sect. 6).

5.5 Transparent Late Service Binding

In a WAN environment such as the Internet, where servers are discovered at run-time, clients cannot predict the set of features provided by (or required by) a particular server until it is located. Static approaches that install new features based only on type information cannot easily provide this kind of late binding.

When server objects are associated with an adaptlet (this happens at deployment time via a configuration file), they are assigned an external object reference that is used by the client side run-time to detect the applicable features.

Essentially, the reference encodes information about the adaptlets associated with this object. Our implementation uses the Tagged IOR[3] Components which is an extensibility element of the CORBA IOR format. This information is used by the Dado interception logic on the client-side to dynamically engage the corresponding client-side adaptlets during stub creation.

The process is graphically illustrated in Fig. 6. When an application object registers itself with a naming service, the reference encodes all active adaptlets (Arrow 1). Subsequently, a retrieved reference (2) is intercepted by the Dado runtime, which decodes the applicable adaptlet identifiers from the reference. For each operation in the interface, it injects the appropriate advice into the execution path of invocations originating from the client.

A special case occurs when a client adaptlet or server adaptlet independently extend from a pair of matched adaptlets; perhaps a deployment expert needed to perform some customization of one side. These sub-types may not match exactly by simply comparing the names of the types. In this case, the adaptlets are coordinated through their “least” common base-type. This can be seen in in Fig. 5. The client adaptlet type `TimeStockServer` will match any server adaptlet of type `Timing`, so that the client adaptlet’s reference to the server adaptlet will properly expose the expected operations.

The late-binding process ensures that client adaptlets are deployed to match adaptlets on the server and removes the responsibility of detecting and deploying applicable adaptlets for each object from the programmer.

5.6 Measurements

High-level programming abstractions, such as those offered by Dado, can often come with some additional run-time overhead. Benefits to program development and maintainance must be carefully evaluated with a clear understanding of this overhead. Here we measure the incremental effect of the actual marshaling work induced by the new communication code (generated by the DDL compiler), as well as for dispatching adaptlet advice and request. For this reason, we use null advice and request methods that do no computation, so that we can focus primarily on the actual overhead of the runtime.

[3] Interoperable Object Reference: used by the CORBA ORB to locate server objects.

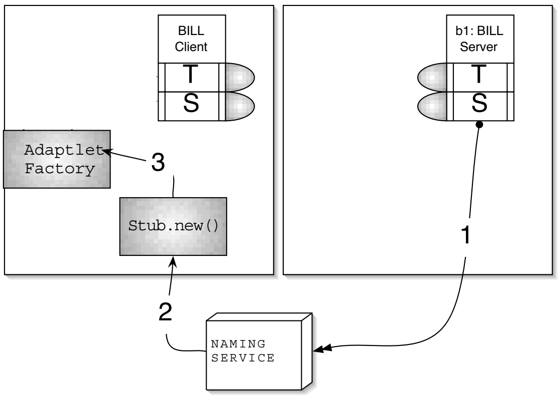


Fig. 6. *Late-binding adaptations* (*1*) Server object, with Security and Transaction adaptlets, named "b1" of type "Bill" is registered with a Naming service. The identifiers "Transaction" and "Security" are tagged to the external object reference. When client looks up object named "b1", the returned object reference (*2*) is intercepted by Dado component. Dado attempts (*3*) to find client-side adaptlets for "Transaction" and "Security" from client-side factory. Factory creates and binds transactions (*T*) and security (*S*) adaptlets to client application object.

The measurements were taken for a single client–server pair over a local area network. The client machine was a 1.80 GHz Intel Pentium with 1 GB main memory running Linux 7.1. The client middleware was JacORB 1.4 on JDK 1.4. The server machine was an 800 MHz Intel Pentium Laptop with 512 MB main memory running Microsoft Windows 2000. Server software used TAO 1.2 compiled in C++ Visual Studio. The DDL interface to the adaptlet used for performance measurement is shown in Fig. 7.

As can be seen, there is one client-side advice and one server side request. The client-side advice is bound to a method-call (with a single argument of type `string`) by a pointcut. In our implementation (not shown), the client-side advice simply captures the string argument from the invocation and calls the server side request, passing along the string argument. So the overhead we are measuring (beyond the normal CORBA invocation overhead) includes the additional cost of: intercepting the invocation on the

```
adaptlets Test{
  client {
    grabArg(string arg);
    before(in string arg) : call(* StockServer::getQuote(arg))
                              -> grabArg(arg);
  };

  server {
    request putArg(string arg);
  };
};
```

Fig. 7. Test adaptlets

client-side, dispatching the client-side advice, executing the client-side request stub, marshaling the additional data, transmitting the additional data over the wire, unmarshaling the data on the server side, and dispatching and executing the request implementation on the server side. All measurements given here are for round-trip delays for a simple invocation that sends a "hello world" string. The data is averaged over 1,000 invocations, and is given in milliseconds.

Experiment	100 Base-T RPC (ms)
1. Plain CORBA	0.65
2. with 1 advice execution, 1 request	1
3. with 10 advice executions, No request	0.68
4. with 10 advice executions, 10 request	1.52
5. Plain CORBA with equivalent raw data Payload for 10 requests	1.38

Fig. 8. Micro-benchmarks

The first row is the plain unloaded CORBA call (Fig. 8), as a baseline for comparison. The second row is a CORBA call with one adaptlet advice, and one additional request. In the third row, we show the effect of applying the advice ten times. The fourth row shows the effect of executing the advice shown on the second row ten times, and attaching a request for each advice. The critical *fifth* row shows an interesting comparison: it measures the plain CORBA call, with additional "piggy-backed" data, *exactly equivalent to 10 request messages*, without any adaptlet code whatsoever. This row corresponds to the precise straw-man comparison for sending data without Dado, and corresponds to the way interceptor-based features (such as Transactions and Real-Time, as per [33], page 30 of Chap. 13) are currently programmed.

As can be seen, the advice itself, which does not send any data, does not induce very large overheads (comparing rows 1 and 3, it is about 5% in both cases for ten advice invocations). We believe the overhead for sending *request*s is largely due to the base cost of marshaling and adding our "piggybacked" messages using the Portable Interceptor API. Another source of overhead may be the use of thread local variables. This is required because the Portable Interceptor API does not provide a way for interceptors to pass data up to higher middleware layers on a language runtime stack. Finally, request messages require an additional hashtable lookup to map operation identifiers to function pointers.

In the future we plan to construct more tests to further isolate sources of overhead in the individual Java and C++ code, to enable optimization of our implementation. We also plan to perform tests using a more realistic client workload, as opposed to the single client–server setup.

6 Client Puzzle-Protocol

As Sect. 2 discussed, CPP defends against cpu-bound DoS attacks. The protocol intercepts client requests and allows only those accompanied by a puzzle solution. The time it takes to solve the problems is predictable; fresh problem instances are created for each request. The need to solve puzzles throttles back the client, preventing it from overloading the server. Typically the puzzle involves finding a collision in a hash function, e.g., finding an input string that hashes to a given n bit value modulo 2^m, for $n > m$. Such puzzles are very easy to generate and require about 2^m times as much effort to solve, as to generate. Here, m is called the difficulty, size, or length of the puzzle. The advantage of CPP over traditional authentication techniques is that CPP defends against DoS even in the face of forged or stolen credentials; additionally, CPP can work in an environment where clients are anonymous. This example is made more concrete through Fig. 9 which describes the data types used in the example first presented in Sect. 2.

```
(1) struct PuzzleChallenge {
(2)    long puzzleID;
(3)    short puzzleLength;
(4)    Octet64 puzzlePreImage;
(5)    Octet16 puzzleHash;
(6) };

(7) exception PuzzleException {
(8)    PuzzleChallenge challenge;
(9) };

(10) struct PuzzleResponse {
(11)   long puzzleID;
(12)   Octet64 puzzleSolution;
(13) };
```

Fig. 9. Client-puzzle protocol messages

A server can generate a `PuzzleChallenge`, as in line 1. The `PuzzleChallenge` is delivered to a client through the `PuzzleException` (line 7) which encapsulates the challenge. The server will not honor the client's invocation until it has solved the puzzle. It can provide a solution through the structure on line 10. Now, we discuss the rationale for this protocol.

By providing a solution to the puzzle a client has *proven* to the server that it has performed a certain computational task. In other words, the client has *expended* some amount of CPU resources. Pragmatically, no client can ever obtain an infinite amount of CPU resources. The CPP is designed so that legitimate clients expend a minor amount of resources. However, since a server can increase the difficulty of puzzles as it becomes overloaded, launching a DoS attack would require more resources than is practical.

6.1 CPP Implementations

The implementation of the CPP is completely application agnostic (recall the DDL given in Fig. 1, Sect. 2). The only interaction with the application is through modification of the client/server control flow. Server responses are preceded by the issuing of a PuzzleException and subsequent verification of the PuzzleResponse. First, we present the client implementation where PuzzleExceptions are caught and used to construct the PuzzleResponse.

```
(1) CPP_ClientImpl implements CPP_Client {
(2)    void catchPuzzle(catchPuzzle_Proceed proceed) {
(3)       try {
(4)           proceed.proceed();
(5)       }
(6)       catch(PuzzleException pc) {
(7)           try {
(8)               PuzzleSolution ps;
(9)               ps.puzzleID = pc.challenge.puzzleID;
(10)              recursiveSolver(pc.challenge.length,
(11)                              pc.challenge.puzzlePreImage);
(12)              ps.puzzleSolution= pc.challenge.puzzlePreImage;
(13)              _that.puzzleSolution(ps);
(14)              proceed.proceed();
(15)          }
(16)          catch(PuzzleException pc2) {
(17)              throw new NO_PERMISSION("CPP Error");
(18)          }
(19)      }
(20)   }
(21)}
```

Fig. 10. CPP_ClientImpl: Declaration of `_that` and `initialize` not shown

The client implementation, Fig. 10, consists of one operation catchPuzzle. This code is placed around invocations that may potentially raise a PuzzleException. In Dado, a special argument (shown on line 2) is provided to every around advice. It provides the same functionality as the proceed keyword in the AspectJ language, which invokes execution of the original application code. However, in Dado, no special language is used for the implementation of adaptlets. The proceed object encapsulates a reference to the next adaptlet decorator on the list. When the function proceed is invoked, it indirectly causes any other adaptlet advice to be invoked. Currently, exceptions are not statically checked, in the future we plan to implement a more sophisticated exception handling mechanism. In lines 3–5 an attempt is made to contact the server by proceeding with the invocation. In case an exception is raised, it is handled in lines 7–15.

First, the ID of the puzzle is copied (line 9) so that the server may correlate challenges and responses. We note that it is not feasible for a client to guess a solution by forging an ID. Second, a "helper" method, recursiveSolver (not shown), is called to solve the puzzle (line 10). Once the puzzle is solved it is added to the invocation through the puzzleSolution context message (line 13). Once again the invocation is attempted on line 14. If failure occurs for the second time, the client adaptlet gives up and throws an exception to the application (line 17).

```
(1) CPP_ServerImpl implements CPP_Server {
(2)    void CPP(CPP_Proceed proceed){
(3)       PuzzleResponseHolder resp;
(4)       if(_that.puzzleSolution(resp) && checkResponse(resp))
(5)          proceed.proceed();
(6)       else {
(7)          PuzzleChallenge challenge = createChallenge(16);
(8)          throw new PuzzleException(challenge);
(9)       }
(10)   }
(11)}
```

Fig. 11. CPP_ServerImpl: Declaration of `_that` and `initialize` not shown

Now we turn to the CPP_Server implementation in Fig. 11. Here also, only one operation is required to be implemented since the context operation is only a vehicle for communicating `PuzzleResponses` to the existing advice. The advice is declared on line 2. It must first check for an incoming response (line 4). If one is available, it must also verify its validity (also line 4) by a call to the helper method `checkResponse` (not shown). At this point the invocation is allowed to `proceed`. Otherwise, a new challenge will be created (line 7) by calling `createChallenge` (also not shown). The challenge is delivered to the client through an exception. If client and server deployment have been properly coordinated, the advice on the client side will catch the exception. Otherwise, the exception will propagate to the application. One of our motivations for explicitly describing matched client and server adaptlets is to help ensure this proper coordination.

6.2 RCT Implementation

To test the compatibility of our implemented Dado software, we set out to deploy an example on a realistic CORBA application. This was done to see if we had made some unrealistic assumptions in our design choices. Although a single case study cannot serve to completely validate our implementation, it has proven useful in understanding some CORBA use-cases that were not previously anticipated. Here we present our example deployment using the CPP adaptlets and the RCT application.

As was discussed in Sect. 2.2, the RCT is a client/server application running over the CORBA middleware. Since this software was developed by an outside third-party we felt it made a good candidate for our case-study. An initial problem involved the server-side software implemented in C++. This software utilized the Omni ORB, while our software was written for the TAO ORB. Omni ORB does not support the mechanisms we required for transparent wrapping of client and server side objects (through the Object Reference Factory). After an initial review of the RCT implementation we judged that it would be straightforward to “port” the software to the TAO ORB. At this time we became familiar with the implementation details of the RCT server software. On the client side we had very little difficulty. Our AspectJ implementation depends only on the details of the generated CORBA stubs and skeletons. For Java, the interface of this code is standardized by the CORBA Portable Stubs/Server specification.

Previously, we had implemented the generic CPP Java and C++ adaptlets. We hypothesized that some applications would require protection on *some* but not *all* of the operations made available to clients. In these cases pointcuts could be used by a deployment expert to seamlessly apply the CPP protocol across those operations.

After reviewing the server software we noticed that a subset of the operations across the RCT interfaces had code that interacted with the server-local relational database. We identified this as a crosscutting concern. These operations were seen to be highly CPU intensive. Each query to the database involves communication across processes and execution of the relational query engine. The other operations (those not part of the database concern) simply return some values stored in the CORBA objects.

This database concern that we identified serves as the basis for the example deployment of the CPP adaptlets.

```
adaptlets CPP_RCT : CPP {
   pointcut databaseConcern() :!(call(* Server::*(..)) ||
                                 call(* PingServer::*(..)));
   client {
      around() : databaseConcern() -> catchPuzzle();
   };
   server {
      around() : databaseConcern() -> CPP();
   };
};
```

Fig. 12. CPP_RCT

CPP Deployment. As shown in Fig. 12, the pointcut `databaseConcern` is declared inside the `adaptlets` scope and shared by both the client and server adaptlets. This allows coordination of advice that throws puzzle exceptions and advice that catches puzzle exceptions. In this case, our late-binding mechanism will enforce that the client CPP_RCT adaptlet will only be deployed with matching server CPP_RCT adaptlets; any other sub-type of the CPP adaptlet *will not be allowed in its place*. This ensures deployment across the client and server is coordinated by the pointcut. The pointcut matches 76 operations in the interfaces of the RCT application except those declared in either the `Server` or `PingServer` interface.

It is often claimed that a pointcut, such as the one shown here, is more concisely able to capture the *intention* of the developer modifying an application. Here, the databaseConcern is captured and ameliorated by applying the CPP protocol.

6.3 CPP Measurements

To measure the end-to-end overhead of our generated code, we compare our adaptlet tool modified RCT application with a hand-modified RCT application. As we described earlier, our current implementation has not been optimized for performance and these results can help to highlight pitfalls in implementation and to focus further implementation efforts. We measure the round-trip latency for a synchronous RPC for eight different application configurations. Measurements were performed in two different environments. The first environment was a single 1.6 GHz Pentium M laptop computer with 1 GB of main memory running both client and server processes in Windows XP Professional. The second environment placed the client process on a 2 GHz Pentium IV computer running Linux at UCDavis and the server process on the aforementioned laptop computer at the University of British Columbia. First we present the details and measurements for the traditional application configurations where no tool support is used. Then we compare these results with the configuration and measurements of the adaptlet modified application.

The baseline "App" configuration measures the round-trip latency for a call to a function called `get_id_from_user_alias` on the RCT server. This was chosen as a representative application function for convenience because its execution presumes no particular application state. We hand-modified the original Java RCT 1.0 client application to take timing measurements using the `java.lang.System.currentTimeMillis()` function before and after 10,000 calls in the local environment and 100 calls in the Internet environment. The values presented are the average over the 10,000 (100) calls in milliseconds. We performed ten trials of the 10,000 (100) calls and took the average of each measurement. The standard deviation for the local measurements was quite small; for example, 0.0014 in measurement 1. In the case of the Internet measurements, the deviation was more significant; for example, 4.19 in measurement 3. These measurements were performed with and without the original server function body present. The server code is the RCT 1.0 application running on the TAO CORBA ORB compiled under Microsoft Visual Studio for C++. Measurements where the function body was removed are labeled as "App(null)" and are presented to see just the overhead of communication through the middleware.

In the "App with CPP" configurations we implemented the CPP protocol over the original application by adding a new application function, `get_challenge`, and modifying the signature of the `get_id_from_user_alias` function. A call to `get_challenge` is made before each call to the original functions to obtain a `PuzzleChallenge`. The signature of the application function was widened to include a parameter for the `PuzzleResponse`. This is the way one might implement the CPP protocol as part of the core application code. We include measurements where the `PuzzleChallenge` generation and `PuzzleResponse` solving code are removed in order to show just the communication costs. These measurements are labeled as "CPP(null)". These configuration serve as a baseline for including the CPP feature but not using any support from Dado.

By comparing measurements 1 and 2 in Fig. 13 we see that the overhead for CPP communication is about 93% (local) and 92% (Internet). Recall that the "App with CPP" configurations include an additional function call (to retrieve a puzzle challenge)

Experiment	RPC latency (ms) local	RCP latency (ms) Internet
1. App(null)	.1274	77.40
2. App(null) with CPP(null)	.2467	148.6
3. App with CPP(null)	1.207	151.6
4. App with CPP	132.3	349.4

Fig. 13. Traditional application measurements

so two RPC calls are required instead of one. Comparison of local measurements 2 and 3 shows the cost for the actual `get_id_from_user_alias` function, 0.9603 ms. Measurement 4 includes the cost for the application and CPP protocol functions. The CPP generation and solving functions add two orders of magnitude to the local end-to-end latency (131.3 ms) and more than doubles the latency over the Internet. Although we tested our CPP implementation for accuracy, we made no attempt to optimize the code which makes use of an MD5[4] hashing library.

Four more measurements were made in order to see the overhead from our tool generated code. The first measurement shows the cost of 10,000 (100) RPC calls hand-modified to use the PortableInterceptor API [19] to piggyback a `PuzzleResponse` to the client request and a `PuzzleChallenge` to the server response. Our generated code uses these PortableInterceptor API calls to package adaptlet `request` and `context` messages. This measurement was performed to see what percentage of overhead is due to the use of this API. This measurement is labeled as “App(null) with Interceptors”.

The final three measurements apply the CPP adaptlets previously presented in this section automatically to the RCT application using our tool support. We include measurements where the adaptlets simply pass dummy challenges and responses, “CPP Adaptlets(null)”, in order to highlight the cost of the generated code.

A comparison of local measurements 1 from Fig. 13 and 5 from Fig. 14 reveals the overhead due to the use of the PortableInterceptor API. As was detailed in Chapter 5 this cost can be high in a local (single host) setting, accounting for a 166% overhead or 0.2124 ms. By factoring this cost from measurement 6 we are left with 0.3295 ms which compared to measurement 2 gives us a 33% overhead for our remaining generated code in the local setting. Factoring the PortableInterceptor API out of measurements 7 and 8 gives us an overhead of 18 and 0.004% respectively when compared to measurements 3 and 4.

In the Internet setting a comparison of measurements 1 through 4 to measurements 5 through 8 shows a smaller overhead. For example, comparing measurement 3 to 7 gives a 2% total overhead. This variation falls within the range of the standard deviation for measurement 3 (4.19 ms) and so is fairly insignificant.

Our dependence on the PortableInterceptor API has an obvious drawback in terms of performance. We made the decision to use this approach because it does not require tightly integrating our tools with a particular CORBA middleware. Tighter integration could probably improve performance at the cost of portability. Even still, there is a

[4] MD5: A common hashing function which is conjectured to be one-way (irreversible).

Experiment	RPC latency (ms) local	RPC latency (ms) Internet
5. App(null) with Interceptors	.3398	77.64
6. App(null) with CPP Adaptlets(null)	.5419	150.9
7. App with CPP Adaptlets(null)	1.641	154.5
8. App with CPP Adaptlets	133.2	349.6

Fig. 14. CPP adaptlet measurements

significant performance overhead for local measurements 6 and 7. Thus we conclude that our current unoptimized implementation may not be suitable for many high performance embedded applications where communication and application latency is low.

7 Client-Side Cache

This next example illustrates an extension to DDL to provide for the use of parametric polymorphism and type adapter operations. A client adaptlet is used to request some key indexed values (e.g., stock quotes) from a server adaptlet. The key values are assumed to periodically change, as the server receives updates. In the example, type parameters are shown in italic font.

Using this client adaptlet, if a request can be answered from the cache, the server is never contacted. To complicate matters slightly, we also assume that the client-side adaptlet can receive from the server-side a TTL value. The server-side adaptlet maintains TTL values for various keys by intercepting updates and observing update frequency for various keys (i.e., some values may be updated more frequently than others). Type parameters in DADO IDL can be used in three places: on the adaptlets scope, the client interface, or the server interface. The first of these is rather different than the latter two and we discuss it first.

Type parameters can be attached to the definition of an `adaptlets` scope using the syntax for Java 1.5 generics. This ensures that some data of the advice and requests of both the client and server are type compatible. Type parameters on the `adaptlets` scope must be instantiated at the DDL level before `adaptlets` can be deployed.

Our current implementation uses *named type equivalence* so client and server programmers cannot independently instantiate type parameters. Types must be agreed upon at the interface-level. Even so, programmers can implement abstract generic adaptlets before concrete types have been determined. Additional code generated when the interface-level type parameters are determined is used to instantiate the programming language-level type parameters of the generic implementation. In the future, *stuctural type equivalence* could be used to provide additional flexibility and will require extensions to our late-binding mechanism to match type structures at run-time. Parameterization may also occur for the client or server interfaces. This is useful for the specification of generic adaptlets where type instantiation is a purely local phenomenon. Our tools generate a pure virtual base class (i.e., interface) in C++. This class can be implemented using generic types and later extended when the concrete types can be determined by a particular deployment scenario.

```
(1)adaptlets Cache<KeyT> {
(2)  client<ValueT> {
(3)    context set_TTL(long ttl);
(4)    around ValueT get(KeyT key);
(5)  };
(6)  server{
(7)    request ask_for_TTL(KeyT key);
(8)    update_TTL(sequence<KeyT> values);
(9)  };
(10)};
```

Fig. 15. Generic cache adaptlets

We start with an abstract interface to the client-side and server-side adaptlets (Fig. 15) necessary to carry out the caching feature. The client-side includes a context operation (line 3) to receive TTL values that are "piggybacked" on server responses. Additionally, an advice (line 4) is specified to short circuit client requests for which cached data is available. The cache is keyed through application data of type *KeyT*.

The server-side has one request and one advice operation. The request `ask_for_TTL` (line 7) allows the client-side adaptlet to signal the server-side that it is planning to cache the data in the current invocation response. The advice `update_TTL` (line 8) should be triggered whenever application events occur on the server-side that invalidate cache data. This allows the server-side to heuristically adjust its forecasted TTL values. The type parameter *KeyT* ensures that the cache key used to index the client-side is type compatible with the key type used by the server-side for tracking TTL values. Both the client-side and server-side adaptlets can have a polymorphic implementation given a suitable target language mapping (currently there is support for C++ templates). Figure 15 can be used to generate the appropriate boilerplate for implementing either a generic client or server adaptlet. In order to highlight the salient properties of our approach we do not present an actual implementation of these cache adaptlets here; the interested reader can find full implementation details in a related thesis [39].

Before deployment can proceed, the adaptlets scope type parameter must be instantiated as shown in Fig. 16. This ensure compatibility between certain operations in the client and server adaptlets as shown in Fig. 15. Now, deployment can proceed separately on the client-side and server-side.

```
adaptlets StringCache : Cache<string> {
};
```

Fig. 16. Instantiating the adaptlets scope type parameter at the interface-level

7.1 Client-Side Deployment

Figure 17 shows the instantiation of the Cache for the StockServer client. To cache the result of the `getQuote` operation the client-side cache type parameter *ValueT* is instantiated as float (line 2). The pointcut matches the invocation to `getQuote` and captures the argument to key the cache. Finally this pointcut is bound to the advice, `get`, that we want to intercept these invocations.

```
(1) adaptlets ClientStockCache : StringCache {
(2)   client : client<float> {
(3)     around(string x) : call(float StockServer::getQuote(x))
(4)       -> get(x);
(5)   };
(6) };
```

Fig. 17. Deploying the cache for the StockServer client

7.2 Server-Side Deployment

The server-side deployment, Fig. 18, consists of dispatching the advice `update_TTL` for operations where TTL calculation can be updated. The pointcut matches the operation `StockServer::setQuotes` and is applied so `update_TTL` will be dispatched before those invocations. Here, the application data-structure `QuoteList` does not match the expected structure of the instantiated sequence of string. Recall that because `strings` have been agreed upon as the cache key, our advice implementation expected to capture a sequence of strings for updating the cache; however, the only relevant operation in the application exposes an argument of type QuoteList. Using inheritance, a deployment expert can insert a customized type adapter function, `hashCodes`, to map the QuoteList, `ql`, captured by the pointcut into the correct format. In our extended DDL language, these adapter functions can be used to wrap advice execution as in line 5. This further demonstrates how it is often straightforward to deploy reusable generic adaptlets even when there are subtle syntactic mismatches with the intended application.

```
(1) adaptlets ServerStockCache : StringCache {
(2)   server{
(3)     StringSeq hashCodes(in QuoteList quotes);
(4)     before(in QuoteList ql) :
(5)       call(* StockServer::setQuotes(ql)) ->
(6)           update_TTL(hashCodes(ql));
(7)   };
(8) };
```

Fig. 18. Deploying the cache for the Stock server

7.3 Cache Example Conclusion

In this section we demonstrated the usefulness of matched client–server adaptlet through the presentation of caching with TTL values. This was done to highlight an additional contribution of our Dado methodology and as a study into the usefulness of aspect-oriented middleware software development.

Also in this section we identified the crosscutting concern of TTL caching. This not only spanned the client and server but also crosscut the structure of the server interface. Modifications were required at both the "get" and "set" operations in order to provide and update the TTL. This was intended to demonstrate how Dado expands the notion of aspect-oriented development into the specific domain of client/server applications. Generic types and type adapter operations provided for the reuse of off-the-shelf adaptlet implementations.

8 Closely Related Work

Dassault Systèmes CVM. The Dassault Systèmes Component Virtual Machine (DS CVM) [36] is targeted at container-based systems, and allows the implementation of custom container services. The DSCVM comprises an efficient, flexible CVM that essentially supports a meta-object protocol which can be used for instrumentation of middleware-mediated events. This allows the CVM to support the triggering of advice when the CVM executes specific events such as component method invocations. Pointcut "trigger" specifications are implemented using the DSCVM events. Advice can be bound to patterns of these events, and thus be used to implement services.

Dado is complementary to DS CVM in that Dado allows elements of crosscutting services to be placed on the client site in a coordinated manner, for reasons argued earlier. Dado also operates outside of a component/container model in "bare-bones" CORBA; thus it must (and does) allow heterogeneity in the implementation of triggering mechanisms such as source transformations, wrapper, etc. The heterogeneity assumption also influences our design of type-checked information exchange between client and server adaptlets, using generated stubs and skeletons.

PROSE. PROSE [40] provides an infrastructure for AOP where aspects can be added or removed dynamically. This is especially useful in mobile computing contexts when required features are dependent on the changing environment where devices are located. PROSE has been used for the construction of *spontaneous container* services which are dynamic, coordinated, J2EE services such as transactions, object persistence, and security. In contrast to our work, no attention is given to the details of AOP in the interface definition language.

QuO. The Quality of Objects (QuO) [41, 42] project aims to provide consistent availability and performance guarantees for distributed objects in the face of limited or unreliable computation and network resources. QuO introduces the notion of a "system condition", which is a user-definable measure of the system, such as load, network delay etc. System conditions can transition between "operating regions" which are monitored by the run-time environment. An important contribution of QuO is that adaptations can be conditionally run to respond not only to normal middleware events, but also to region transitions.

QuO's version of adaptlets, Qoskets, are confined to a single system. Unlike Dado, Quo provides no special support for communicating information from a client-side Qoskets to a server-side Qoskets.

AspectIX. The AspectIX [43] middleware for Java is motivated by similar goals as the QuO project. The use of *object fragments* allows the placement of adaptive code transparently at the client, server, or at an intermediate location. Policies are used by the client at run-time to determine the composition of fragments, allowing client specific QoS. Unlike Dado, AspectIX does not utilize an interface level model for declarative service/object bindings. Also, generic services or heterogeneity in instrumentation are not considered.

Lasagne. Lasagne [38] is a framework for dynamic and selective combination of extensions in component based applications. Each component can be wrapped with a set of decorators to refine the interaction behavior with other components. Every decorator layer is tagged with an extension identifier. Clients can dynamically request servers to use different sets of decorators at run-time. An innovative aspect of Lasagne is the usage of extension identifiers to consistently turn on and off adaptive behavior. However, the reliance on interceptors constrains all extensions to have the same interface. Any additional extension-specific information must be communicated using a "context" object, without the benefits of typechecking or automated marshaling.

Cactus. The Cactus [44, 45] framework provides customizations in distributed systems and has been applied in the area of distributed object computing. Software implementing customizations is known as an adaptive component (AC). ACs are injected into a system using the proxy technique. These ACs encapsulate alternative implementations of a specific service ala the Strategy pattern. When important state changes in the system occur ACs can respond by swapping in and out different strategies. ACs may communicate in order to reach consensus on strategy changes however no one particular consensus approach is dictated by their model. Cactus does not utilize static IDL level information for detecting application level events. Also, information shared between ACs is not described in an interface as are adaptlet request messages. Thus Cactus follows the reflective, dynamically typed approach to adaptation.

Remote Pointcut. DJCutter is an extension to the AspectJ language that provides support for *remote pointcuts* [46]. This enables advice to be executed on a local host when a pointcut is matched on a remote host. DJCutter also extends the scope of cflow pointcuts to execution threads that span more than one host. As noted by the authors, this work is complementary to ours because Dado does not address the above mentioned features but DJCutter does not provide any support for heterogeneous interface descriptions.

Software Architecture. In software architecture, connectors [47, 48, 49] have proven to be a powerful and useful modeling device. Connectors reify the concern of interaction between components, and are a natural foci for some crosscutting concerns. Implementations of architectural connectors have also been proposed [50, 51, 52, 53]. Some of these provide specific services [51, 52, 53] over DH middleware, such as security. Our work can be viewed as providing a convenient implementation vehicle for different connector-like services in a heterogeneous environment. The DDL language and compiler allow service builders to write client and server adaptlets that provide many kinds of "connector-style" functionality, while the DDL "plumbing" handles the communication details. Furthermore, the pointcut language allows a flexible way of binding this functionality to components, using pattern matching to bind events to adaptlets.

9 Conclusion

We conclude here with several observations about Dado, it's limitations, and our plans for future work.

Design Choices. The design space of a convenient framework to implement DH crosscutting features is quite large, comprising many dimensions such as synchronization mechanisms, scope of data, and the handling of exceptions. The current implementation of DDL has made some reasonable choices, but other choices will need to be explored as other application demands are confronted. Some examples: "cflow-scoped" state, i.e., state that is implicitly shared between adaptlets across a distributed thread of invocations; distributed features whose scope transcends a matched stub-skeleton adaptlets; other (e.g., synchronous) interactions between adaptlets. We would like to implement an adaptlet-per-object and adaptlet-per-session instance policy as well. Currently, only run-time exceptions are supported for adaptlets—in the Java mapping, better static checking would be desirable.

Implementation Limitations. Currently our implementation has some limits. As outlined earlier, the marshaling and dispatch functions need further optimization. Additionally, further tests are necessary using a more realistic client work-load.

Interactions. Feature interactions are a difficult issue that Dado adaptlets must deal with. We note here that it is currently possible to program interactions between two Dado services: one can write a third service that pointcuts adaptlets in each, and responds to the triggering of both by preventing one from running, changing argument values, return values etc. Recently, we have designed a feature mediation protocol for crosscutting distributed features [54]. Feature policies from both client and server are interpreted by middleware to dynamically choose applicable features.

In conclusion, Dado is an IDL-driven approach to programming crosscutting concerns in DH systems based on placing "adaptlets" at the points where the application interacts with the middleware. It supports heterogeneous implementation and triggering of adaptlets, allows client- and server-adaptlets to communicate in a type-checked environment using automated marshaling, provides flexibility in communication between adaptlets, allows flexible binding, and late deployment of adaptlets on to application objects.

Acknowledgments

We gratefully acknowledge support from NSF CISE 0204348. The paper has benefited greatly because of comments from the anonymous reviewers of ICSE 2003.

References

[1] Tarr P.L., Ossher H., Harrison W.H. Jr., S.M.S. N degrees of separation: multi-dimensional separation of concerns. In: *International Conference on Software Engineering*, 1999

[2] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W.G. An overview of AspectJ. In: *European Conference on Object Oriented Programming*, 2001

[3] Murphy G.C., Lai A., Walker R.J., Robillard M.P. Separating features in source code: an exploratory study. In: *International Conference on Software Engineering*, 2001

[4] Coady Y., Brodsky A., Brodsky D., Pomkoski J., Gudmundson S., Ong J.S., Kiczales G. Can AOP support extensibility in client–server architectures? In: *Proceedings, ECOOP Aspect-Oriented Programming Workshop*, 2001

[5] Wohlstadter E., Jackson S., Devanbu P. DADO: enhancing middleware to support crosscutting features in distributed heterogeneous systems. In: *Proceedings of the International Conference on Software Engineering*, 2003

[6] Smaragdakis Y., Batory D. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *Software Engineering and Methodology*, 11(2):215–255, 2002

[7] Amsler T., Walters R. Open RCT home. http://davinci.cs.ucdavis.edu

[8] Dean D., Stubblefield A. Using client puzzles to protect TLS. In: *Proceedings of USENIX Security Symposium*, 2001

[9] Balzer B. Transformational implementation: an example. *IEEE Transactions on Software Engineering*, 7(1), 1981

[10] des Rivieres J., Smith B. The implementation of procedurally reflective languages. *Technical Report ISL-4*, Xerox PARC, 1984

[11] Aksit, M., Wakita K., Bosch J., Bergmans L., Yonezawa A. Abstracting object interactions using composition filters. In: *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, 1993

[12] Chiba S. A metaobject protocol for C++. In: *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1995

[13] Tatsubori M., Chiba S., Itano K., Killijian M.O. Openjava: a class-based macro system for java. In: *OOPSLA Workshop on Reflection and Software Engineering*, 1999

[14] Chiba S. Load-time structural reflection in Java. In: *Proceedings of the European Conference on Object-Oriented Programming*, pp. 313–336, 2000

[15] Smaragdakis Y., Batory D. Implementing layered designs with mixin layers. In: *European Conference on Object Oriented Programming*, 1998

[16] Walker R.J., Murphy G.C. Implicit context: easing software evolution and reuse. In: *Foundations of Software Engineering*, 2000

[17] Liang S., Hudak P., Jones M. Monad transformers and modular interpreters. In: *Symposium on Principles of Programming Languages*, 1995

[18] Wang N., Parameswaran K., Schmidt D. The design and performance of meta-programming mechanisms for object request broker middleware. In: *Conference on Object-Oriented Technologies and Systems (COOTS)*, 2000

[19] Narasimhan P., Moser L., Mellior-Smith P. Using interceptors to enhance CORBA. *IEEE Computer*, 1999

[20] Kon F., Román M., Liu P., Mao J., Yamane T., Magalhães L.C., Campbell R.H. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2000

[21] Clarke M., Blair G., Coulson G., Parlavantzas N. An efficient component model for the construction of adaptive middleware. In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2001

[22] Fraser T., Badger L., Feldman M. Hardening COTS software with generic software wrappers. In: *IEEE Symposium on Security and Privacy*, 1999

[23] Souder T.S., Mancoridis S. A tool for securely integrating legacy systems into a distributed environment. In: *Working Conference on Reverse Engineering*, 1999

[24] Roman E., Ambler S., Jewell T. *Mastering Enterprise JavaBeans*. Wiley, New York, 2001

[25] JBoss. (*4.0* edn.). http://www.jboss.org

[26] Cohen T., Gil J. AspectJ2EE = AOP + J2EE: towards an aspect based, programmable and extensible middleware framework. In: *Proceedings of ECOOP'04*, 2004

[27] AspectWerkz. (*2* edn.). http://aspectwerkz.codehaus.org

[28] Spring Framework. (*1.2.3* edn.). http://www.springframework.org

[29] Pawlak R., Seinturier L., Duchien L., Florin G. Jac: a flexible framework for aop in java. In: *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection)*, 2001

[30] Zhang C., Gao D., Jacobsen H.A. Towards just-in-time middleware architectures. In: *International Conference on Aspect-Oriented Software Development*, 2005

[31] Brose G. JacORB: implementation and design of a Java ORB. In: *Proceedings of the International Working Conference on Distributed Applications and Interoperable Systems*, pp. 143–154, 1997

[32] Schmidt D. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995

[33] Object Management Group. *CORBA 3.0 Specification*. (*3.0* edn.)

[34] Romer T., Lee G.V.D., Wolman A., Wong W., Levy H., Bershad B.N., Chen J.B. Instrumentation and optimization of Win32/Intel executables using etch. In: *Usenix Windows NT Workshop*, 1997

[35] Kon F., Costa F., Blair G., Campbell R.H. The case for reflective middleware. *Communications of the ACM*, ACM, New York, USA, 45(6):33–38, 2002

[36] Duclos F., Estublier J., Morat P. Describing and using non functional aspects in component based applications. In: *International Conference on Aspect-Oriented Software Development*, 2002

[37] Gamma E., Helm R., Johnson R., Viissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1994

[38] Truyen E., Vanhaute B., Joosen W., Verbaeten P., Jorgensen B.N. Dynamic and selective combination of extensions in component-based applications. In: *International Conference on Software Engineering*, 2001

[39] Wohlstadter E. *Aspect-Oriented Development of Distributed Object Applications*. PhD thesis, University of California at Davis, 2005

[40] Popovici A., Alonso G., Gross T. Spontaneous container services. In: *European Conference on Object-Oriented Programming*, 2003

[41] Loyall J., Bakken D., Schantz R., Zinky J., Karr D., Vanegas R., Anderson K. QuO Aspect languages and their runtime integration. In: *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components*, 1998

[42] Vanegas R., Zinky J., Loyall J., Karr D., Schantz R., Bakken D. Quo's runtime support for quality of service in distributed objects. In: *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 1998

[43] Hauck F., Becker U., Geier M., Meier E., Rastofer U., Steckermeier M. Aspectix: a quality-aware, object-based middleware architecture. In: *Proceedings of the 3rd IFIP International Conference on Distributed Application and Interoperable Systems*, 2001

[44] Chen W.K., Hiltunen M., Schlichting R. Constructing adaptive software in distributed systems. In: *International Conference on Distributed Computing Systems*, 2001
[45] He J., Hiltunen M.A., Rajagopalan M., Schlichting R.D. Providing qos customization in distributed object systems. In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001
[46] Nishizawa M., Chiba S., Tatsubori M. Remote pointcut—a language construct for distributed aop. In: *Proceedings of the International Conference on Aspect-Oriented Programming*, 2004
[47] Mehta N.R., Medvidovic N., Phadke S. Towards a taxonomy of software connectors. In: *International Conference on Software engineering*, 2000
[48] Perry D.E., Wolf A.L. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, ACM, New York, USA, 17(4):40–52, 1992. http://doi.acm.org/10.1145/141874.141884
[49] Allen R., Garlan D. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, ACM, New York, USA, 6(3):213–249, 1997. http://doi.acm.org/10.1145/258077.258078
[50] Ducasse S., Richner T. Executable connectors: towards reusable design elements. In: *Foundation of Software Engineering*, 1997
[51] Dashofy E.M., Medvidovic N., Taylor R.N. Using off-the-shelf middleware to implement connectors in distributed software architectures. In: *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, California, United States, IEEE Computer Society, Los Alamitos, CA, USA, pp. 3–12, 1999
[52] Dashofy E.M., Medvidovic N., Taylor R.N. Using off-the-shelf middleware to implement connectors in distributed architectures. In: *International Conference on Software Engineering*, 1999
[53] Spitznagel B., Garlan D. A compositional approach to constructing connectors. In: *Working IEEE/IFIP Conference on Software Architecture (WISCA)*, 2001
[54] Wohlstadter E., Tai S., Mikalsen T., Rouvellou I., Devanbu P. GlueQoS: middleware to sweeten quality-of-service policy conflicts. In: *Proceedings of the International Conference on Software Engineering*, 2004

Shakeins: Nonintrusive Aspects for Middleware Frameworks[*]

Tal Cohen and Joseph (Yossi) Gil[**]

Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel
{ctal, yogi}@cs.technion.ac.il

Shaken, not stirred.

— Bond, James Bond (Goldfinger, 1964).

Abstract. *Shakeins* are a novel programming construct which, like mixins and generic classes, generates new classes from existing ones in a universal, uniform, and automatic manner: From a given class, a shakein generates a new class which has the same type as the original, but with different data and code implementation. We argue that shakeins are restricted, yet less chaotic, aspects. We further claim that shakeins are well suited for the introduction of *aspect-oriented programming* (AOP) into existing middleware applications. We introduce the ASPECTJ2EE language which, with the help of shakeins and a new *deploy-time* weaving mechanism, brings the blessings of AOP to the J2EE framework. A unique advantage of ASPECTJ2EE, which is less general (and hence less complicated) than ASPECTJ, is that it can be smoothly integrated into J2EE implementations without breaking their architecture.

1 Introduction

1.1 Enterprise Applications

Enterprise applications are large-scale software programs used to operate and manage large organizations. These make the world's largest, and arguably most important, software systems, including the programs that run government organizations, banks, insurance companies, financial institutes, hospitals, etc. Enterprise applications make the world go around!

It is often the case that enterprise applications run on heterogeneous platforms connecting various independent systems (called *tiers*) into a coherent whole. The various tiers of an enterprise application can include, e.g., legacy mainframe servers, dedicated database servers, personal computers, departmental servers, and more.

The core functionality served by enterprise applications is often quite simple. It does not involve overly elaborate computation or pose complex algorithmic demands. However, developing enterprise applications is considered a daunting task, due to orthogonal requirements presented by most of these applications: uncompromising reliability

[*] Part of the contents of this paper was presented in the European Conference on Object Oriented Programming [1] and in a recently submitted paper [2].

[**] Research supported in part by the IBM faculty award.

demands, unyielding security requirements, and complete trustworthiness that the applications must exhibit.

The staggering demand for rapid development of enterprise applications initiated a series of component-based *middleware architectures*, such as CORBA and DCOM. A prime example of these, and emphasizing client/server and multitier structures, is *Java 2, Enterprise Edition* (J2EE)[1] which uses Enterprise JavaBeans (EJB)[2] as its elaborate component model.

The immense complexity of middleware frameworks and enterprise applications makes these an appealing target for mass application of aspect-oriented programming (AOP) [3]. AOP is a methodology which can be used for encapsulating the code relevant to any distinct nonfunctional concern in *aspect* modules. (We assume basic familiarity with standard AOP terminology, including *joinpoint*—a well-defined point in the program's execution, *pointcut*—a specification of a set of joinpoints, *advice*—code that is added at specified joinpoints, *weaving*—the process of applying advices to joinpoints, and *aspect*—a language construct containing advices.)

AOP can also be thought of as answering the same demand [4, 5]. Indeed, the functionality of J2EE application servers can be *conceptually* decomposed into distinct aspects such as persistence, transaction management, security, and load balancing. The effectiveness of this decomposition is evident from Kim and Clarke's case study [6], which indicates that the EJB framework drastically reduces the need for generic AOP language extensions and tools.

Yet, as we shall see below, the EJB support for functional decomposition is limited and inflexible. In cases where the canned EJB solution is insufficient, applications resort again to a tangled and highly scattered implementation of crosscutting concerns. Part of the reason is that current J2EE servers do not employ AOP in their implementation, and do not enable developers to decompose new nonfunctional concerns that show up during the development process.

1.2 J2EE Services as Managers of Nonfunctional Concerns

A natural quest is for a harmonious integration of middleware architectures and AOP. Indeed, there were several works on an AOP-based implementation of J2EE servers and services (see e.g., the work of Choi [7]).

Ideally, with the J2EE framework (and to a lesser extent in other such frameworks), the developer only has to implement the domain-specific *business logic*. This "business logic" is none other than what the AOP community calls *functional concerns*. The framework takes charge of issues such as security, persistence, transaction management, and load balancing which are handled by *services* provided by the *EJB container*. Again, these issues are none other than *nonfunctional concerns* in AOP jargon.

Suppose, for example, that the programmer needs data objects whose state is mirrored in persistent storage. This storage must then be constantly updated as the object is changed during its lifetime, and vice versa. Automatic updates can be carried out by the *Container-Managed Persistence* (CMP) service of the EJB container. To make this

[1] http://java.sun.com/j2ee
[2] http://java.sun.com/products/ejb/

happen, the objects should be defined as *entity beans*. Entity bean types are mapped to tables in a relational database with an appropriate XML configuration file. This *deployment descriptor* file also maps each bean attribute (persistent instance variable) to a field of the corresponding table.

Another standard J2EE service is security, using an approach known as *role-based security*. Consider, for example, a financial software system with two types of users: clients and tellers. A client can perform operations on his own account; a teller can perform operations on any account, and create new accounts. By setting the relevant values in the program's deployment descriptor, we can limit the account-creation method so that only users that were authenticated as tellers will be able to invoke it.

Other services provided by the EJB container handle issues such as transaction management and load balancing. The developer specifies which services are applied to which EJB. Deployment descriptors are used for setup and customization of these services. Thus, J2EE reduces the implementation of many nonfunctional concerns into mere configuration decisions; in many ways, they turn into *nonconcerns*.

The prime example in this work is EJBs. But, the J2EE design guideline, according to which the developer configures various services via deployment descriptors, is not limited to EJBs; it is also used in other parts of the J2EE platform. For example, servlets (server-side programs for Web servers) also receive services such as security from their container, and access to specific servlets can be limited using role-based security. This is also true for Java Server Pages (JSPs), another key part of the J2EE architecture. Hence, in our financial software example, certain privileged Web pages can be configured so that they will be only accessible to tellers and not to clients.

The various issues handled by EJB container services were always a favored target for being implemented as aspects in AOP-based systems [8, pp. 13–14]. For example, Soares et al. [4] implement distribution, persistence, and transaction aspects for software components using ASPECTJ [9]. Security was implemented as an aspect by Hao et al. [5]. The use of aspects reduces the risk of scattered or tangled code when any of these nonfunctional concerns is added to a software project.

Conversely, we find that J2EE developers, having the benefit of container services, do not require as much AOP! Kim and Clarke's case study [6] comprised of an e-voting system which included five nonfunctional concerns: (1) persistent storage of votes, (2) transactional vote updates, (3) secure database access, (4) user authentication, and (5) secure communications using a public key infrastructure [6, Table 1]. Of these five nonfunctional concerns, *not one* remained crosscutting or introduced tangled code. The first three were handled by standard J2EE services, configured by setting the proper values in the deployment descriptors. The last two were properly modularized into a small number of classes (two classes in each case) with no code replication and no tangled code.

This is hardly surprising. The J2EE platform designers have provided what, in their experience, an enterprise application developer *needs* for his nonfunctional concerns. This was done without relying on aspect-oriented research concepts; it was done in order to scratch an itch. J2EE provides a robust solution to real-world problems encountered by real-world programmers when developing real-world multitier enterprise applications.

1.3 Shakeins and Middleware

In summary, despite the complexity of middleware systems, and despite the similarity between AOP and the services that frameworks such as J2EE provide, it is expected that integrating this modern programming technique into the frameworks that need it, will face reluctance. Some J2EE application server vendors, most noticeably JBoss, have recently integrated support for aspects into their products. However, the forthcoming version of the EJB standard (EJB 3.0[3]) chose not to adopt a complete AOP solution, but rather a rudimentary mechanism that can only be used to apply advice of a limited kind to a limited set of methods in an inflexible manner. The basic services are not implemented using such advise, but rather remain on a different plane, unreachable and nonmodifiable by the developer.

Plain aspects should make it easier to *add* services to J2EE, if re-engineered to use AOP. They should also make it simpler to *replace* existing services with alternative implementations. This paper makes the case that much more can be offered to the client community by *shakeins*, a novel programming construct, combining features of aspects with selected properties of generics and mixins [10]. Shakeins should make it possible not only to *add* services, but also address two issues that current J2EE users face: *configuration* of existing services and *greater flexibility* in their application. Shakeins should also enjoy a smoother entrance into the domain because they *reuse* the existing J2EE architecture, and because they simplify some of the daunting tasks (e.g., life-cycle methods, see Sect. 6) incurred in the process of developing EJBs.

In a nutshell, a shakein receives a class parameter and optional configuration parameters, and generates a new class which has the same type as the original, but with different data and code implementation. In a sense, shakeins are restricted, yet less chaotic, aspects. Like aspects, they can be used to modularize nonfunctional concerns without tangled and scattered code. Unlike traditional aspects, shakeins preserve the object model, present better management of aspect scope, and exhibit a more understandable and maintainable semantic model.

Shakeins are more restricted than aspects in that there are limits to the change they can apply to code. However, unlike aspects, shakeins take configuration parameters, which make it possible to tailor the change to the modified class.

We argue that the explicit, intentional, configurable, and parameterized application of shakeins to classes makes them very suitable to middleware frameworks and enterprise applications, based on the following reasons:

- The architecture of *existing* middleware frameworks is such that in generating enterprise application from them, the system architect may selectively use any of the services that the framework provides. We have that *a chief functionality of middleware frameworks is in the precise management of nonfunctional concerns,* and that this objective is well served by the shakeins construct.
- The services that a middleware framework offers are applicable only to certain and very distinguishable components of the application, e.g., EJBs in J2EE. The need to isolate the application of aspects is aggravated by the fact enterprise applications tend to use other software libraries, which should not be subjected to the framework

[3] http://www.jcp.org/en/jsr/detail?id=220

modifications. For example, a business mathematics library that an application uses for amortizing loan payments, should not be affected by the underlying framework security management. Shakeins are suited to this domain because of its fundamental property by which *crosscutting concerns should not be allowed to cut through an entire system.*

- Further, for modularity's sake, it is important that these parts of the enterprise application, which are external to the middleware, are not allowed to interact with the services offered by the middleware. The fact shakeins are forbidden from creating new types helps in *isolation of the crosscutting concerns.*
- To an extent, shakeins generalize existing J2EE technology, presenting it in a more systematic, programming language-theoretical fashion. While doing so, shakeins make it possible to enhance J2EE services in directions that plain aspects fail to penetrate: The configurability of shakeins makes it possible to generalize existing J2EE services. Explicit order of application of shakeins to components adds another dimension of expressive power to the framework. And by using *factories* (a new language construct, detailed in Appendix 1), such enhancements can be applied with a minimal disturbance of existing code.

1.4 AspectJ2EE

Shakeins draw from the lessons of J2EE and its implementation. To demonstrate their applicability to this domain, we introduce the AspectJ2EE [1] language, which shows how shakeins can be used to bring the blessings of AOP to the J2EE framework. AspectJ2EE is geared towards the generalized implementation of J2EE application servers and applications within this framework.

As the name suggests, AspectJ2EE borrows much of the syntax of AspectJ. The semantics of AspectJ2EE is adopted from shakeins, while adapting these to AspectJ. The main syntactical differences are due to the fact that "aspects" in AspectJ2EE can be parameterized, as will be revealed by the forthcoming exposition. In the initial implementation we have of AspectJ2EE, parameter passing and the application of shakeins are not strictly part of the language. They are governed mostly by external XML configuration files, a-la J2EE deployment descriptors.

A distinguishing advantage of this new language is that it can be smoothly integrated into J2EE implementations without breaking their architecture. This is achieved by generalizing the existing process of binding services to user applications in the J2EE application server into a novel *deploy-time* mechanism of weaving aspects. Deploy-time weaving is superior to traditional weaving mechanisms in that it preserves the object model, has a better management of aspect scope, and presents a more understandable and maintainable semantic model. Also, deploy-time weaving stays away from specialized JVMs and bytecode manipulation for aspect weaving.

Standing on the shoulders of the J2EE experience, we can argue that shakeins in general, and AspectJ2EE in particular, are suited to systematic development of enterprise applications. Unlike existing attempts to add AOP functionality to J2EE application servers, the AspectJ2EE approach is methodical. Rather than add aspects as an additional layer, unrelated to the existing services, our approach is that even the standard services should be implemented on top of the AOP groundwork. Using AspectJ2EE, the fixed set of standard J2EE services is replaced by a library of core aspects. These services can be augmented with new ones, such as logging and performance monitoring.

It should be noted that ASPECTJ2EE has its limitations compared to more traditional implementations of aspects; in particular, it is not at all suited to low-level debugging or nit-picking logging. (For example, access to *nonprivate* fields by classes other than the defining class is not a valid joinpoint.) However, it is not for these tasks that ASPECTJ2EE was designed, and it is highly suited for dealing with large systems and global AOP. (The field access restriction example does not mature into a major hurdle in the kind systems we are interested in; field management can always be decomposed into getter and setter methods, and in fact *must* be decomposed this way in J2EE applications, where fields are realized as *attributes* with proper joinpoints at their retrieval and setting.)

Moreover, the ASPECTJ2EE language has specific support for the composition of aspects that are scattered across program tiers (*tier-cutting concerns*), such as encryption, data compression, and memoization. We stress that unlike previous implementations of aspects within the standard object model, ASPECTJ2EE does not merely support "before" and "after" advices and "method execution" joinpoints. ASPECTJ2EE supports "around" advices, and a rich set of joinpoints, including control-flow based, conditional, exception handling, and object- and class-initialization.

Historically, the developers of enterprise applications are slow to adopt new technologies; a technology has to prove itself again and again, over a long period of time, before the maintainers of such large-scale applications will even consider adopting it for their needs. It is not a coincidence that many large organizations still use and maintain software developed using some technologies, such as COBOL, that other sectors of the software industry view as thoroughly outdated. The huge investment in legacy code slows the adoption of new technology.

We believe that the fact that shakeins preserve the standard object model, and rely on existing J2EE technologies, should contribute to widespread adoption of this new technology in the middleware software domain.

Outline. Section 2 explains in greater detail the shakeins programming construct while comparing it to other such constructs. Section 3 makes the case for representing J2EE services as aspects in general, and as shakeins in particular. In Sect. 4 we compare the shakeins mechanism with two important products introducing aspects into J2EE. Deploy-time weaving is discussed in Sect. 5, which also explains why this mechanism is suited to shakeins. We then show in Sect. 6 how shakeins can be married with middleware frameworks, by means of the ASPECTJ2EE language. Section 7 lists several possible innovative uses for ASPECTJ2EE, some of which can lead to substantial performance benefits. Section 8 concludes.

2 The Case for Shakeins

As explained above, a shakein S takes an existing class c as input, along with optional configuration parameters, and generates from it a new class $S\langle c\rangle$, such that the *type* of $S\langle c\rangle$ is the *same* as that of c. The *realization* of this type in $S\langle c\rangle$ may, and usually will, be different than in c. Further, the process of generating $S\langle c\rangle$ from c is *automatic*, *universal*, and *uniform*.

This section makes the case for shakeins, and explains this process in greater detail: Section 2.1 explains what we mean by making the distinction between the type defined by a class and its realization by the class definition. With this distinction, we explain in Sect. 2.2 why the mechanisms of aspects[4] and inheritance serve similar purposes but from different, foreign and hard-to-reconcile perspectives. Section 2.3 presents the shakeins mechanism to resolve this schism.

Section 2.4 explains that shakeins, just like mixins, have an explicit parameter-based application semantics, and discusses some of the benefits of this semantics. The theoretical discussion of this new language mechanism is concluded in Sect. 2.5 which shows how they simplify some of the subtle points of aspects.

2.1 The Five Facets of a Class

As early as 1971, Parnas [11] made the classical distinction between the *interface* and the *materialization* perspectives of a software component. It is only natural to apply this distinction to class definitions in JAVA and other mainstream object oriented languages. In this section we do so, while taking into notice that, unlike the software components of the 1970s, classes are instantiable.

Accordingly, the interface of a class presents two facets: the *forge* of the class, being the set of non-`private` constructor signatures; and its *type*, the set of externally accessible fields and methods as well as available cast operations. The materialization of a class has three facets: the *implementation*, i.e., the method bodies; the *mill*, being the implementation of the forge; and the *mold*, which is the memory layout used for creating objects.

Figure 1 shows a sample class[5] (a) alongside its forge (b), type (c), and mold (d).

2.2 The Aspects–Inheritance Schism

A key feature of OOP is the ability to define new classes based on existing ones. The five facets described above can be used for explaining inheritance in JAVA and similar languages: Inheritance *extends* the type, mold, and implementation of a class. By means of overriding, inheritance also makes it possible to *replace* or *refine* parts or all of the implementation. The mold, however, cannot be refined.

Interestingly, inheritance in mainstream programming languages *recreates* the forge from scratch, since constructor signatures are not inherited. Therefore, inheritance is allowed to make arbitrary and incompatible changes to the forge facet of a class. The mill, however, can only be refined, since constructors must invoke inherited versions.

Examining the notion of aspects from the perspective of modifications to the five facets, we see that they offer a similar repertoire of modifications. Aspects are advertised as means for breaking the implementation into its orthogonal concerns; accordingly, an

[4] We refer here to ASPECTJ-style aspects. Most other aspect-oriented languages are variants of this same style.

[5] A similarly structured class served as the running example in Kiczales and Mezini's [12] work on modularity and aspects.

```
class Point implements Shape {
   int x, y;
   public Point(int x, int y){
      this.x = x;
      this.y = y;
   }
   public Point(){
      this(0,0);
   }
   public Point(Point other){
      this.x = other.x;
      this.y = other.y;
   }
   public int getX() {
      return x;
   }
   public int getY() {
      return y;
   }
   public void setX(int x) {
      this.x = x;
   }
   public void setY(int y) {
      this.y = y;
   }
   public void moveBy(int dx, int dy) {
      x += dx; y += dy;
   }
}
```

(a) The class definition.

```
public Point(int x, int y);
public Point();
public Point(Point other);
```

(b) The forge.

```
int x, y;
public int getX(); public int getY();
public void setX(int);
public void setY(int);
// From Shape:
public void moveBy(int, int);
// From java.lang.Object:
public boolean equals(Object);
public int hashCode();
// ... etc.
// Upcast operations:
public (Shape)(); public (Object)();
```

(c) The resulting type.

int x	32 bits
int y	32 bits
Fields inherited from `Object`	
Hidden fields added by the JVM	

(d) The class's mold.

Fig. 1. The `Point` class definition and the ensuing forge, type, and mold

aspect may replace or refine the implementation. In many languages, aspects can also *extend* the type and the mold by introducing new members into a class. (This is known as *member introduction* in AspectJ. Conceptually, fields in an aspect that has per-object instantiation may also be viewed as fields added to the base object's mold.)

Still, unlike inheritance, the application of an aspect does not define a new type, nor does it introduce a new mold or a new implementation; it changes the mold and the implementation of existing classes *in situ*.[6] And, while member introduction makes it possible to modify the type of classes, it does not introduce a *new* type—because classes touched by aspects cease to exist in their original form; only the modified form exists. In particular, there cannot be instances of the original class in memory.

The similarity of aspects and inheritance raises intriguing questions pertaining to the interaction between the two mechanisms: Does the aspectualized class inherit from its base version? May aspects delete fields from the mold, or modify it by changing the type of fields? Likewise, can methods be removed from the type? Can their signature be changed? How does the application of an aspect to a class affect its subclasses?

The interaction of joinpoints with inheritance raises still more questions.[7] Suppose that class `B` inherits from `A`, with, or without overriding method `m()`. Then, does the pointcut **`call`**`(A.m())` apply to an invocation of method `m()` on an object whose

[6] Also, aspects can only *extend* the forge (using member introduction) but not replace it.

[7] In AspectJ, two different joinpoint types can be used to apply an advice to method execution: The **`execution`** joinpoint places an advice at the method itself, whereas **`call`** places the advice in at the method's client, i.e., at the point of invocation rather than at its target.

dynamic type is `B`? Conversely, does the pointcut **`call`**`(B.m())` ever apply if class `B` does not define a method `m()`, but inherits it from `A`?

The reason that all these questions pop up is that aspects were not designed with inheritance in mind. The original description of aspects [3] did not dedicate this new construct to OOP languages. Similarly, the founding fathers of OOP did not foresee the advent of AOP, and in many occasions inheritance was used for some of the purposes AOP tries to serve. We witness what may be called *The Aspects–Inheritance Schism*.

Gradecki and Lesiecki [13, p. 220] speculate that *"mainstream aspect-oriented languages . . . will possess simpler hierarchies with more type-based behavior defined in terms of shallow, crosscutting interfaces."* In other words, these two authors expect that this schism is resolved by programmers abandoning much of the benefits of inheritance. Our suggestion is that the resolution will be a re-definition of aspects cognizant of the class facets and of inheritance.

2.3 Shakeins as Class Re-implementors

The shakeins construct is a proposal to resolve the aspect–inheritance schism by making aspects which are a restricted form of inheritance. Like inheritance, shakeins generate a new class from an existing one. Yet unlike inheritance, they cannot extend the class type. In fact, they cannot change the base class type at all.

Thus, we can say that *shakeins make a* ***re-implementation*** *of a class*.

Shakeins allow only specific changes to the class facets. Programming constructs which restrict one or more facets are not strange to the community: an *abstract class* is a type, a partial (possibly empty) implementation, and an incomplete mold. *Interfaces* are pure types, with no implementation, forge, mill, or mold. Also, *traits* [14] have a fragment of a type and its implementation, but no forge, mill, or mold.

Another familiar notion on which shakein rely is that of multiple implementations of the same (abstract) type, as manifested, e.g., in the distinction between signatures and structures in ML. The major difference is, however, that a shakein assumes an existing implementation of a class, which is then modified by the shakein.

For concreteness, Fig. 2a shows a shakein `DisplayUpdating` that can be used to modify class `Point`, so that the display is refreshed after each coordinate change. The shakein works on any class that defines methods `setX(`**`int`**`)`, `setX(`**`int`**`)`, and/or `moveBy(`**`int`**`,`**`int`**`)`. The desired effect is obtained by applying this shakein to `Point`.

Figure 2b shows the `Confined` shakein. This shakein confines the range of valid values for `x` and `y` to positive integers only. It is applicable to class `Point`, or any class with **`int`** fields `x` and `y`.[8]

In the process of re-implementing a class, a shakein may introduce methods and fields to the class. Such members must be **`private`**, accessible to the shakein but not to the external world (including inheriting classes, and classes in the same package). The reason is that such accessibility would have implied a change to the class type, which is not allowed to shakeins. Shakeins are allowed to introduce **`concealed`** members, which are accessible to further re-implementations of this re-implementation.

[8] While the update presented by this shakein can cause methods that update `x` and `y` to throw an exception, it is an unchecked exception, and therefore it does not alter the methods' signature.

```
shakein DisplayUpdating {
  pointcut change() :
       execution(setX(int)) ||
       execution(setY(int)) ||
       execution(moveBy(int,int));

  after() returning: change() {
    Display.update();
  }
}
```

(a) A shakein adding display refresh operations.

```
shakein Confined {
  pointcut update(int v) :
       (set(int x) || set(int y))
       && args(int v);

  before(int v): update(v) {
    if (v < 0)
      throw new IllegalArgumentException();
  }
}
```

(b) A shakein for limiting the valid range of `Point`'s coordinates.

Fig. 2. Two shakeins that can be applied to the `Point` class

We will write `Confined<Point>` to denote the application of shakein `Confined` to `Point`, and `DisplayUpdating<Confined<Point>>` for the application of shakein `DisplayUpdating` to the result, etc.

The re-implementation property implies that although a class $S\langle c\rangle$ can (and usually will) be instantiated, it is not possible to define *variables* of this class. Instances of $S\langle c\rangle$ can be freely stored in variables of type c, as in the following JAVA like pseudo-code:

c `var = ` **`new`** $S\langle c\rangle$ `();`

In the `Point` example, we may then write

```
Point p = new DisplayUpdating<Point>();
```

The type preservation property of shakeins sets a clear semantics for the interaction of these with inheritance. As it turns out, the application of shakeins does not modify or break the inheritance structure. More formally, let c_1 and c_2 be two classes, and suppose that c_2 inherits from c_1. Then, we can write $c_2 \prec c_1$ to denote the fact that the type of c_2 is a subtype of c_1. Let S be a shakein. Then, we can also write $c_1 \simeq S\langle c_1\rangle$ to denote the fact that the type of $S\langle c_1\rangle$ is the same as c_1. Similarly, $c_2 \simeq S\langle c_2\rangle$. By substitution, we can obtain $S\langle c_2\rangle \prec S\langle c_1\rangle$, $c_2 \prec S\langle c_1\rangle$, and $S\langle c_2\rangle \prec c_1$. In fact, we have

Proposition 1. *For all classes c_1 and c_2 such that $c_2 \prec c_1$, and arbitrary shakeins S and S', $S'\langle c_2\rangle \prec S\langle c_1\rangle$.*

In our running example, shakein `DisplayUpdating` can be applied to any subclass of `Point`. If class `ColorPoint` extends `Point`, then the type of `DisplayUpdating <ColorPoint >` is a subtype of `Point`.

Figure 3 makes a graphical illustration of Proposition 1. It depicts a simple base class hierarchy consisting of classes c_1, c_2, c_3, and c_4, where $c_4 \prec c_2 \prec c_1$, and $c_3 \prec c_1$. There are also three shakeins, S_a, S_b, and S_c, where shakeins S_b and S_c are implemented using S_a, and each of the shakeins is applied to each of the classes.

We see in the figure that for all $i = 1, \ldots, 4$, the type of class c_i is the same as its three re-implementations $S_a\langle c_i\rangle$, $S_b\langle c_i\rangle$, and $S_c\langle c_i\rangle$. This common type is denoted by a round-cornered box labeled "Type c_i". As shown in the figure, the subtyping relationship is not changed by re-implementations; e.g., the type of class $S_a\langle c_4\rangle$ is a subtype of $S_b\langle c_2\rangle$'s type.

The figure should also make it clear how the type system of shakeins can be embedded in a standard JVM. Each shakein application results in generation of a JAVA class, which compiles into a distinct `.class` file. Both vertical dashed arrows, representing type inheritance, and horizontal arrows, representing shakein application, are translated to class inheritance, i.e., an **`extends`** relationship between classes.

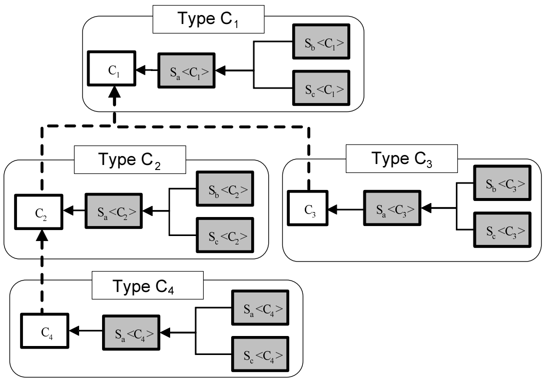


Fig. 3. A class hierarchy subjected to shakeins

To see why Proposition 1 holds in this embedding, recall that the program does not have any variables (including parameters and fields) of the shakein classes: All instances of class $S\langle c\rangle$ are stored in variables of type c. In the figure, instances of type $S_a\langle c_4\rangle$ are stored in variables of type c_4, which is upcastable to type c_2.

2.4 Parameterized Class Modification

Aspects are distinguished from inheritance in that they can be automatically applied to multiple classes. In contrast, a subclass is defined with respect to a specific superclass; the nature of the extension (both of the interface and the materialization) is specific to every such instance. To apply the same kind of change to multiple base classes, the details of the change must be explicitly spelled out each time. Thus, although inheritance is a kind of what is known in the literature as *universal polymorphism* [15], it is not a *uniform* mechanism; each inheriting class modifies the base class in an ad hoc manner.

It is therefore instructive to compare aspects to the parameterized version of inheritance, i.e., *mixins* [10]. A mixin, just like an aspect, makes it possible to apply the same kind of change (expressed in terms of inheritance) to multiple base classes.

Mixins were invented with the observation that there is a recurring need to extend several different classes in the exact same manner. They allow programmers to carry out such a change without rewriting it in each inheriting class. In languages such as MODULA-π [16] the repeating change to the base class can be captured in a mixin M, which, given a class c, generates a class $M\langle c\rangle$ such that $M\langle c\rangle$ inherits from c. In languages with first-class genericity, mixins can be emulated by writing, e.g.,

```
template<typename c> class M: public c { ... }
```

in C++, or `class M<c> extends c` in NEXTGEN [17].

Generic structures are also a kind of a universal polymorphism, but their application is *uniform*. The above emulation of mixins by generics makes it clear that mixins are both *universal* and *uniform* kind of polymorphism.

Here, we enrich and simplify the aspects approach with ideas drawn from the work on mixins. We argue that the seemingly pedestrian notation of mixins, $M\langle c\rangle$, hides much expressive power. The reason is that parameter passing, a familiar mechanism of programming languages, exhibits several useful features which are missing in the current implicit and declarative mechanism of aspect application. These features are:

(1) *Selective Application.* After a mixin was defined, it can be applied to selected classes without affecting others.
(2) *Nondestructive Application.* The application of a mixin M to a class c does not destroy c, and both classes $M\langle c\rangle$ and c can be used. Instances of both may reside in memory simultaneously and even interact with each other.
(3) *Explicit and Flexible Ordering.* Mixins M_1 and M_2 can be applied in any order, to generate either $M_1\langle M_2\langle c\rangle\rangle$, $M_2\langle M_1\langle c\rangle\rangle$, or both. Further, it is possible to apply the same mixins in different orders to different class.
(4) *Composition.* Mixins can be conveniently thought of as functions, and as such it makes sense to compose them. In some languages supporting mixins one can define a new mixin by writing, e.g., $M := M_1 \circ M_2$, with the obvious and natural semantics.
(5) *Configuration Parameters.* It is straightforward to generalize mixins so that they take additional parameters which control and configure the way they extend the base class. In the templates notation, one can write, for example:
```
template<typename c, char *log_file_name>
  class Log: public c {
    // Log into file log_file_name
  }
```
(6) *Repeated Application.* One can write $M\langle M\langle c\rangle\rangle$, but it is not possible to apply the same aspect twice to a class. This works well with parameterized mixins; e.g.,
`Log["post.log"]<Security<Log["pre.log"]<c>>>`
will generate two log files, one before and one after any security checks imposed by the `Security` mixin.
(7) *Parameter Checking.* It is useful to declare constraints to the parameters of mixins, and apply (meta-) type checking in passing actuals to these. Languages such as JAM [18] offer this feature in mixins.

None of these features is available in ASPECTJ-style aspects, where aspect application is global and destructive, its order globally set and not customizable per target class, aspects cannot be composed and take no class-specific configuration parameters, repeated application is impossible, and there is no parameter checking mechanism (e.g., it is not possible to ensure that aspects are applied only to specific kinds of classes, except in a very narrow sense).

Evidently, these differences are not a coincidence. AOP languages were designed with the belief that the composition of a system from its aspects is an implicit process, carried out by some clever engine which should address the nitty-gritty details. One of the chief claims of this paper is that this presumption does not carry to middleware applications, especially when we wish to preserve the investment in existing architecture. Complex systems are composed of many different components in many different ways. An automatic composition engine tends to make arbitrary decisions, whose combined effect is more likely to break larger systems. Our support for this claim is by the detailed description of the aspect-oriented re-structuring of J2EE.

Shakeins are similar to mixins in that they take parameters. As such they are a *universal* and *uniform* polymorphic programming construct. Shakeins are similar to generic structures in that they may take multiple parameters. However, whereas both mixins and generics suffer from their *oblivious* and *inflexible* mode of operation, i.e., they are unable to inspect the details of the definition of the base-, or argument-class, and generate specific code accordingly. As a result, mixins fail in tasks such as re-generating constructors with the same signature as the base, or applying the same change to multiple methods in the base class. Similar restrictions apply to generics. In contrast, shakeins use the same pointcut mechanism as aspects, and are therefore highly adaptable.[9]

The component-system Jiazzi [19] uses a mixin-like mechanism to implement *open classes*, which in turn can be used for implementing crosscutting concerns [20]. Jiazzi components are also parameterized, taking packages (sets of classes) as arguments. However, Jiazzi's open class approach differs from shakeins in that the base class is modified, both in its implementation and in its type; and such changes are propagated to all existing subclasses.

The chief parameter of a shakein is the class that this shakein re-implements. The definition of a shakein does not need to name or even mention this parameter, since it is implicit to all shakeins, in the same way that JAVA methods have a **this** parameter. Additional parameters, if they exist, are used to configure or direct the re-implementation.[10]

More formally, a shakein S takes an existing class c as input along with other configuration parameters, $\mathcal{P}_1, \ldots, \mathcal{P}_n$, $n \geq 0$, and generates from it a new class $S[\mathcal{P}_1, \ldots, \mathcal{P}_n] \langle c \rangle$, such that the *type* of $S[\mathcal{P}_1, \ldots, \mathcal{P}_n] \langle c \rangle$ is the *same* as that of c.

Figure 4a shows how configuration parameters can enhance the functionality of shakein `Confined` (first shown in Fig. 2b). As shown in the figure, the updated version of `Confined` is configured by four **int** parameters, specifying the minimal and

```
shakein Confined[int minX, int maxX,
                 int minY, int maxY] {
  pointcut updateX(int v) :
             set(int x) && args(int v);
  pointcut updateY(int v) :
             set(int y) && args(int v);

  before(int v): updateX(v) {
    if ((v < minX) || (v > maxX))
      throw new IllegalArgumentException();
  }

  before(int v): updateY(v) {
    if ((v < minY) || (v > maxY))
      throw new IllegalArgumentException();
  }
}
```

(a) A parameterized version of `Confined`.

```
shakein DisplayUpdating
     [pointcut change()] {

  after() returning: change() {
    Display.update();
  }

}
```

(b) A parameterized version of `DisplayUpdating`.

Fig. 4. Parameterized shakeins

[9] Another issue that plagues mixins (and generics that inherit from their argument) is that of *accidental overloading* [18]. Shakeins overcome this problem by using an automatic renaming mechanism, which is possible since no shakein-introduced member is publicly accessible.

[10] Note that aspects can also be thought of as taking implicit parameters. However, shakeins are distinguished from aspects in that their invocation is *explicit*, and that aspects take no configuration parameters.

maximal values for the `x` and `y` coordinates of its target class. To obtain an instance of `Point` restricted to the $[0, 1023] \times [0, 767]$ rectangle, one can write

```
Point p = Confined[0,1023,0,767]<Point>(511,383); //Initially at center.
```

Another kind of configuration parameter is a pointcut expression. Figure 4b shows a revised version of `DisplayUpdating`, which uses a pointcut parameter. The parameter `change` denotes the joinpoints whose execution necessitates a display update. An actual value of `change` specifies a concrete such set. For example, the following:

```
DisplayUpdating[
   execution(setX(int)) || execution(setY(int)) || execution(moveBy(int,int)
]<Point>
```

is an application of `DisplayUpdating` to class `Point`.

Consider now Fig. 5, showing class `Line` implemented using two `Points`. An application of `DisplayUpdating` to `Line` is by writing

```
DisplayUpdating[execution(moveBy(int,int))]<Line> .
```

```
class Line {                                 // cont.
   private Point a, b;
                                                public moveBy(int x, int y) {
   public Line(Point from, Point to) {              a.moveBy(x,y);
      a = new Point(from);                          b.moveBy(x,y);
      b = new Point(to);                         }
   }                                          }
```

Fig. 5. Class `Line`

This re-implementation of `Line` does not suffer from the redundant display updates problem [12], which would have occurred in traditional AOP, i.e., display updates occurring both in the implementation `Line` and its encapsulated `Points`. Thanks to the nondestructive semantics of shakeins, these two `Points` can be of the nonupdating variant. This does not prohibit other `Points` in the system (which are not part of `Lines`) to be display updating.

```
// Auxiliary shakein, used to confine updates to one axis:
shakein ConfinedUpdate[pointcut setValue(int v), int min, int max] {
   before(int v): setValue(v) {
      if ((v < min) || (v > max)) throw new IllegalArgumentException();
   }
}

// Compose the auxiliary shakein twice, once per axis:
shakein Confined[int minX, int maxX, int minY, int maxY] :=
   ConfinedUpdate[set(int x) && args(int v), minX, maxX] o
       ConfinedUpdate[set(int y) && args(int v), minY, maxY];
```

Fig. 6. A third version of `Confined`, using a composition of `ConfinedUpdate`

In contrast, an aspect-based solution should check that no `change` advice is in effect before executing this advice. This checking must be carried out *at runtime*, by examining the runtime stack with what is known in AspectJ as a **`cflowbelow`** condition.[11]

[11] Still, **`cflowbelow`**-based pointcuts can be used in shakeins where needed—for example, to prevent a call to `Point.moveBy()` from causing multiple display updates as it changes both point coordinate; see [12] for a discussion of this use of **`cflowbelow`**.

Figure 6 is a re-definition of the `Confined` shakein using pointcut parameters. Comparing the figure with Fig. 4a, we see that this implementation uses *shakein composition*. In fact, we have here a *repeated application* of the auxiliary shakein `ConfinedUpdate`, first for the Y-axis, and then, on the result for the Y-axis.

In one sentence summary of this section, we may refine the description of shakeins from Sect. 2.3: *Shakeins make a **configurable re-implementation** of a **class parameter**.*

2.5 A New Light on Aspect Terminology

By viewing shakeins as operators, taking classes as arguments and producing classes, we can clarify some of the illusive notions and terms used in traditional AOP jargon:

(1) *Aspect Instantiation.* The semantics of instantiation of aspects, i.e., the reification of aspects at runtime, can be quite confusing; ASPECTJ has are as many as five different modes of such instantiations.
In contrast, shakeins, just like mixins and generics, operate on code, and as such, they no longer exist at runtime. (Of course, the result of a shakein is a class which may have run-time instances.)

(2) *Aspect Precedence.* AOP languages make it possible to define global precedence rules for aspects. However, this declaration is never complete unless all participating aspects are known; and there is no possibility of applying a set of aspects to different classes in a different order.
As operators, shakeins can easily be applied in a specific order as the need arises.

(3) *Abstract Pointcut.* An *abstract pointcut* is a named pointcut with no concrete specification. An *abstract aspect* may include an abstract pointcut and apply advice to it. A *concrete* version of this aspect may be generated by defining a "subaspect" for it, which must provide a concrete value to the abstract pointcut. This mechanism provides a measure of flexibility; yet the terms are confusing when used in conjunction with OOP. An abstract pointcut does not offer dynamic binding to the concrete version, nor does it offer a signature that must be obeyed by all its implementors.
Standing at the shakein point of view, abstract pointcuts are nothing more than a poor man's replacement for pointcut parameters.

(4) *Abstract Aspects.* An aspect is abstract not only when it contains abstract pointcuts, but also when it contains abstract methods. These methods can then be invoked from advice, effectively using the TEMPLATE METHOD [21] design pattern. Using concrete subaspects that implement these methods, the programmer may define a different aspect each time.
A shakein may also define a method as abstract. Such a definition overrides the method definition in the shakein parameter. An application of such a shakein yields an abstract class, which can then be extended by other shakeins and provided with a concrete implementation for the missing methods. However, unlike in the case of abstract aspects, the implementation of the abstract methods in a shakein can optionally be different for each application of the shakein.

(5) *Aspect Inheritance.* Except in the case of abstract super-aspects, the notion of aspect inheritance is ill defined, and most aspect languages (including ASPECTJ) prohibit this—because a subaspect will clearly share the same pointcut definitions as its

parent, and the same advice; does this imply that each advice should therefore be applied *twice* to each matching joinpoint?
As shakeins are operators, shakein inheritance is nothing more than operator composition, as in **`shakein`** `S3<c> = S2<S1<c>>`.

It is therefore evident that while sharing the flexibility and expressive power of aspects, shakeins, by virtue of being parameterized operators on code, reconcile naturally with the concept of inheritance. They do not exhibit the confusing notions that accompany aspects, and their behavior is easy to understand within the domain of AOP.

3 The Case for AOP in J2EE

Having presented the advantages of shakeins, the time has come for evaluating their usefulness. To do so, we would like to go beyond the small examples presented above. The more thorough examination is in the context of a real-life application, and in particular, with respect to the J2EE implementation.

In this section, we examine first some of the existing limitations of the service-based architecture of J2EE (Sect. 3.1), and then proceed (Sect. 3.2) to explain how the marriage of AOP with J2EE is better served with shakeins, which also help in addressing these limitations.

3.1 Limitations of the Services-Based Solution

Even though the J2EE framework reduces the developer's need for AOP tools, there are limits to such benefits. The reason is that although the EJB container is configurable, it is neither extensible nor programmable. Pichler et al. [22] refer to the combination of these two problems as *lack of tailorability*.

The container is *not extensible* in the sense that the set of services it offers is fixed. Kim and Clarke [6] explain why supporting logging in the framework would require scattered and tangled code. In general, J2EE lacks support for introducing new services for nonfunctional concerns which are not part of its specification. Among these concerns, we mention memoization, precondition testing, and profiling.

The container is *not programmable* in the sense that the implementation of each of its services cannot be easily modified by the application developer. For example, current implementations of CMP rely on a rigid model for mapping data objects to a relational database. The service is then useless in the case that data attributes of an object are drawn from several tables. Nor can it be used to define read-only beans that are mapped to a database view, rather than a table. The CMP service is also of no use when the persistent data is not stored in a relational database (e.g., when flat XML files are used).

Any variation on the functionality of CMP is therefore by *re-implementation* of object persistence, using what is called *Bean-Managed Persistence* (BMP). BMP support requires introducing callback methods (called *life-cycle methods* in EJB parlance) in each bean. Method `ejbLoad()` (`ejbStore()`), for example, is invoked whenever memory (store) should be updated.

The implication is that the pure business logic of EJB classes is contaminated with unrelated I/O code. For example, the tutorial code of Bodoff et al. [23, Chap. 5] demonstrates a mixup in the same bean of SQL queries and a JAVA implementation of functional concern. Conversely, we find that the code in charge of persistence is *scattered* across all entity bean classes, rather than being encapsulated in one cohesive module.

Worse, BMP may lead to code *tangling*. Suppose, for example, that persistence is optimized by introducing a "dirty" flag for the object's state. Then, each business logic method which modifies state is tangled with code to update this flag.

Similar scattering and tangling issues rise with modifications to any other J2EE service. In our financial software example, a security policy may restrict a client to transfer funds only out of his own accounts. The funds-transfer method, which is accessible for both clients and tellers, acts differently depending on user authentication. Such a policy cannot be done by setting configuration options, and the method code must explicitly refer to the nonfunctional concern of security.

To summarize, whenever the canned solutions provided by the J2EE platform are insufficient for our particular purpose, we find ourselves facing again the problems of scattered, tangled, and crosscutting implementation of nonfunctional concerns. As Duclos et al. [24] state: "*clearly, the 'component' technology introduced successfully by EJB for managing non-functional aspects reaches its limits*".

3.2 Marrying J2EE with AOP

Despite the limitations, the J2EE framework enjoys extensive market penetration, commanding a multibillion dollar market.[12] In contrast, AOP is only making its first steps into industry acceptance. It is natural to seek a reconciliation of the two approaches, in producing an aspect based, programmable and extensible middleware framework.

Release 4.0 of JBoss (discussed below in Sect. 4.1) is an open-source application server which implements the J2EE standard, and supports aspects with no language extensions.[13] Aspects are implemented in JBoss as JAVA classes which implement a designated interface, while pointcuts are defined in an XML syntax. However, JBoss does not implement J2EE services aspects.

We argue that a proper marriage of AOP and J2EE requires that each of J2EE's core services is expressed as an aspect. The collection of these services then forms the *core aspect library*, which relying on J2EE success, would not only be provably useful, but also highly customizable. Developers should be able to add their own services (e.g., logging) or modify existing ones, possibly using inheritance in order to re-use proven aspect code. The resulting aspects could then be viewed as stand-alone modules that can be re-used across projects.

Another implication is that not all aspects must come from a single vendor; in the current J2EE market, all J2EE-standard services are provided by the J2EE application server vendor. If developers can choose which aspects to apply, regardless of the application server used, then aspects implemented by different vendors (or by the developers themselves) can all be used in the same project.

Choi [7] was the first to demonstrate that an EJB container can be built from the ground up using AOP methodologies, while replacing services with aspects which exist independently of the container.

[12] http://www.serverwatch.com/news/article.php/1399361

[13] http://www.onjava.com/lpt/a/3878

Focal to all this prior work was the attempt to make an existing widespread framework more robust using AOP techniques. Our work here adopts a new approach to the successful marriage of J2EE and AOP, by using the notion of shakeins. More concretely, we propose a new AOP language ASPECTJ2EE, which in using shakeins, draws from the lessons of J2EE and its programming techniques.

The main issues in which the ASPECTJ2EE language differs from ASPECTJ are:

(1) *Aspect targets.* ASPECTJ can apply aspects to any class, whereas in ASPECTJ2EE aspects can be applied to *enterprise beans* only, i.e., those modules to which J2EE services are applied. (This selective application is made possible by the shakein semantics, which always have a designated target.)

   As demonstrated by the vast experience accumulated in J2EE, aspects have great efficacy precisely with these classes. We believe that the acceptance of aspects by the community may be improved by narrowing their domain of applicability, which should also benefit understandability and maintainability.

   It should be stressed, however, that this is not a limitation of the shakein concept but rather an ASPECTJ2EE design decision.

(2) *Weaving method.* Weaving the base class together with its aspects in ASPECTJ2EE relies on the same mechanisms employed by J2EE application servers to combine services with the business logic of beans. This is carried out entirely within the dominion of object-oriented programming, using the standard JAVA language, and an unmodified JVM. Again, this is made possible by the shakein semantics.

   In contrast, different versions of ASPECTJ used different weaving methods relying on preprocessing, specialized JVMs, and dedicated bytecode generators, all of which deviate from the standard object model.

(3) *Aspect parametrization.* Since the aspects in ASPECTJ2EE are shakeins, they take three kinds of parameters: pointcut definitions, types, and literal values. Parameterized aspects can be applied to EJBs by providing (in the EJBs deployment descriptor) a concrete value for each parameter, including concrete pointcut definitions. Pointcut parameters provide significant flexibility by removing undesired cohesion between aspects and their target beans, and enables the development of highly reusable aspects. It creates, in ASPECTJ2EE, the equivalent of Caesar's [25] much-touted separation between aspect implementation and aspect binding.

   Other aspect parameter types also greatly increase aspect reusability and broaden each aspect's applicability.

(4) *Support for tier-cutting concerns.* ASPECTJ2EE is uniquely positioned to enable the localization of concerns that cross not only program modules, but program tiers as well. Such concerns include, for example, encrypting or compressing the flow of information between the client and the server. Even with AOP, the handling of tier-cutting concerns requires scattering code across at least two distinct program modules. We show that using ASPECTJ2EE, many tier-cutting concerns can be localized into a single, coherent program module.

J2EE application servers offer the developer only minimal control over the generation of support classes. ASPECTJ2EE, however, gives a full AOP semantics to the deployment process. With deploy-time weaving, described next, the main code is unmodified, both at the source and the binary level. Further, the execution of this code is unchanged, and can be carried out on any standard JVM.

ASPECTJ2EE does not impose constraints on the base code, other than some of the dictations of the J2EE specification on what programmers must, and must not, do while defining EJBs. These dictations are that instances must be obtained via the home interface, rather than by directly invoking a constructor or any other user-defined method; business methods must not be `final` or `static`; and so forth.

4 Comparison with JBoss and Spring

Obviously, we were not the first to observe the case for using aspects in middleware applications in general, and in J2EE in particular. Indeed, there is a large body of previous work in which aspects are applied in the middleware domain: JAC [26], Lasagna [27], PROSE [28], JAsCo [29], and others.

Shakeins were designed with J2EE in mind—so in this section we compare shakeins with the two prime applications of AOP technology to J2EE. Section 4.1 compares shakeins with the dynamic aspects of *JBoss Application Server*.[14] A comparison with the AOP features of the *Spring Application Framework*[15] is the subject of Sect. 4.2. Unlike shakeins and other similar research, JBoss and Spring are industrial-strength software artifacts employed in production code. The lessons that these two teach are therefore valuable in appreciating the merits of shakeins.

4.1 JBoss AOP

Version 4.0 of JBoss was the first implementation of J2EE to integrate AOP support. For technical reasons, the *JBoss AOP* approach features advice- rather than aspect-level granularity, where each advice is encapsulated in what is called an *interceptor* class.

It is telling that JBoss AOP enhancements of the aspect notion are similar to those of shakeins, including: advice composition (*stacks* in the JBoss jargon), parameterized advice, selective and repeated application of advice, explicit and flexible (rather than global) ordering, and configuration parameters. We interpret this as a supporting empirical support to the claim that flat-oblivious aspects should be extended in certain ways.

Still, since the application of (standard) advice in JBoss is carried out in situ, destroying the original class, the JBoss approach suffers from the aspect/inheritance schism, instantiation complexity, etc. Perhaps in recognition of these difficulties, JBoss AOP also supports *dynamic AOP*—the ability to apply advice per instance rather than per class. A class must be "prepared" for such a dynamic application, by injecting into its code (at load time) hooks for the potential joinpoints. The class loader consults an XML configuration file for the list of classes to prepare, and the hook locations. It is then possible, at run time, to add or remove interceptors.

In this extent, JBoss's flexibility and repertoire of features is greater than that of shakeins. JBoss offers the ability to *un-apply* an advice at run time. This feature is missing in shakeins, but can be emulated by testing an on/off flag at the beginning of

[14] http://www.jboss.org
[15] http://www.springframework.org

every advice. Conversely, note that JBoss's flexibility has its inherent limits, e.g., since interceptors are applied to existing objects, advice cannot be applied to constructors.

The most major difference between JBoss AOP and shakeins is the approach taken for integration with the base technology. As explained in Sect. 2, shakeins are a language extension, which draws from principles of OO and genericity. In contrast, the variety of features in JBoss AOP is realized by a sophisticated combination of loaders, run-time libraries, and XML configuration files, and without any changes to the compiler (or the JVM). Thus, (probably in answer to Sun's J2EE certification requirements) JBoss AOP is an implementation of aspects through a software framework built over vanilla JAVA.[16]

Since JBoss aspects are not part of the language, a programmer who wishes to exploit aspect features is asked to master a variety of tools, while following the strict discipline dictated by the framework, with little if any compiler checking. For example, the run-time application of advice is achieved using a JAVA API; the compiler is unaware of the involved AOP semantics. As a result, many mistakes (e.g., an attempt to access an invalid method argument) can only be detected at run time, possibly leading to run-time errors. Other problems, such as a mistyped pointcut (which matches no joinpoints) will not be detected at all. Thus, in a sense, the offers of JBoss can be compared to assembly programming: immense flexibility but with greater risks. However, unlike assembly code, performance in JBoss is not at its peak.

The clockwork driving the JBoss framework is visible to the programmer. The programmer *must* understand this mechanism in order to be able to use it. This visibility has its advantages: the illusive issue of aspect instantiation in ASPECTJ is clarified by JBoss: since interceptors must be instantiated explicitly prior to their application, the semantics of aspect instance management is left up to the programmer.

To illustrate some of the issues of a framework-based implementation of aspects, consider Fig. 7, which demonstrates how the `Confined` shakein (Fig. 6) is implemented in JBoss. Figure 7a depicts a JAVA class which, by obeying the framework rules, can be used as an interceptor. The run-time content of argument `inv` to method `invoke` (lines 12–18) is the only information that the method has on the interception. The method assumes (line 13) that the joinpoint kind is a field-write. The newly assigned value is obtained (line 14) by downcasting an `Object` to the proper type. Both downcasts will fail if the interceptor is applied to an incorrect joinpoint.

Figure 7b is the XML code that directs the injection of the hooks intended for this interceptor into class `Point`. The interceptor is intimately coupled with the XML, in making tacit assumptions on the joinpoint kind and the argument type; violations of these assumptions are only detected at run time.

To create a shakein-typed instance of `Point`, one may write

```
Point p = Confined[0,1023,0,767]<Point>(); //Shakein version.
```

The JBoss equivalent is a bit longer:

```
Point p = new Point(); // JBoss dynamic AOP version.
((Advised)p)._getInstanceAdvisor().appendInterceptor(new Confined("x", 0, 1023));
((Advised)p)._getInstanceAdvisor().appendInterceptor(new Confined("y", 0, 767));
```

[16] Compliance with these requirements also explains why standard J2EE services are not implemented as aspects in JBoss, and are therefore not as flexible as they might be.

(a)
```
public class Confined implements Interceptor {
  private int min, max;
  private String fieldName;

  public Confined(String fieldName, int min, int max) {
     this.min = min; this.max = max;
     this.fieldName = fieldName;
  }

  public String getName() { return "Confined"; }

  public Object invoke(Invocation inv) throws Throwable {
     FieldWriteInvocation fwi = (FieldWriteInvocation)inv;
     int v = (Integer)(fwi.getValue());
     if (fwi.getField().getName().equals(fieldName))
        if (v < min || v > max) throw new IllegalArgumentException();
     return inv.invokeNext(); //proceed to subsequent interceptors/base code
  }
}
```

(b)
```
<aop>
  <prepare expr="set(int Point->x) OR set(int Point->y)" />
</aop>
```

Fig. 7. a `Confined` as a JBoss interceptor, and **b** the supporting configuration file

This demonstrates more intricacies of implementing aspects by a software framework: First, we see that advices are applied only after the object was constructed (no refinement of the constructors is possible). Second, since there is no explicit composition operator,[17] two interceptors must be manually applied, one per axis. Third, we see that `p` must be casted to the interface type `Advised`. This interface type is implemented by the modified (prepared) version of `Point`; yet the compiler is not aware of this change. If the class was not prepared (e.g., an inconsistency in the XML file), then this cast attempt will fail. Finally, again due to compiler obliviousness, field names are represented as string literals (here, as arguments to the interceptor's constructor). Any mistake in the field name (e.g., writing "`X`" instead of "`x`") will go undetected and result in silent failure. (By comparison, an empty pointcut argument to the auxiliary shakein from Fig. 6 triggers a compile-time warning.)

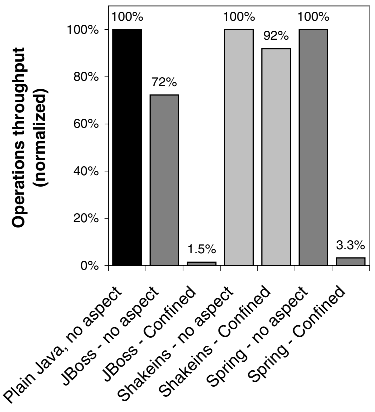


Fig. 8. Performance degradation of class `Point` with different strategies of applying a "confinement" aspect

Figure 8 compares the run-time performance of the JBoss and the shakein implementation of aspect `Confined`. The figure depicts the operations throughput of class `Point` in the base implementation and in the different aspectualized versions.[18] We see that the original class suffers no performance

[17] Stacks cannot be used here.

[18] Specifically we used the number of times the sequence of calls ⟨`setX`,`setY`,`moveBy`⟩ can be completed in a time unit.

penalty in the shakein version. The shakein-advised instance, generated by two subclassing operations, is about 8% slower. In contrast, while using JBoss AOP, instances of the original class suffer from a performance impact of about 28% before any advice is applied; this is the overhead introduced by the joinpoint hooks. Once applied, the reflection-based interceptors slow the JBoss version to 1.5% of the original throughput.

There are two main sources of performance degradation in the JBoss implementation. *Time wise*, hooks slow down code, and this slowdown occurs even if no advices are applied to the receiver instance. (In class `Point`, this slowdown was by 28%.) Moreover, even a nonprepared class may be slowed down, if its code matches, e.g., a joinpoint of a **`call`** to a prepared class.

Additional slowdown is caused by the advice having to use reflection-like objects in order to learn about the joinpoint. The invocation object must be downcast to the specific invocation type (`FieldWriteInvocation` in Fig. 7a), and arguments must be downcast from `Object` references to their specific types. (As we have seen, this slowdown was by more than an order of magnitude in `Point`. We expect a more modest relative performance degradation in classes that do more substantial computation.)

Space wise, the advice (whenever a joinpoint is reached) is reified in a run-time object. The memory consumed by this object must be managed, thereby leading to additional slowdown. The invocation itself is reified in a number of objects (metadata, argument wrappers, arguments array, etc.) which add to the space and time overheads of the implementation. (A quick inspection of Fig. 7a reveals that there are at least four objects generated in reifying the joinpoint.)

4.2 A Comparison with Spring AOP

The Spring Application Framework is an "inversion of control"[19] container, used for applying services to standard JAVA objects. It is often used in conjunction with a J2EE server for developing enterprise applications; however, Spring provides alternatives to many of the J2EE services, in a more flexible and developer-friendly manner.

Objects in Spring are "beans", all obtained via a centralized factory which is configured using an XML file. This XML file specifies what properties should be set and what services applied to each bean type. Developers can choose from a wide range of predefined services (e.g., Hibernate[20]-based persistence) or define their own. New services (as well as existing ones) are defined using Spring's AOP facilities.[21]

Much like shakeins, AOP in Spring is based on the generation of new classes. When advice is applied to a class, a new class is generated, which either implements the same interfaces as the base class or else extends it as a subclass. Thus, Spring enjoys several of the benefits of shakeins; most notably, there is no performance penalty to instances of the original class, which remains unmodified (see Fig. 8). However, beyond this similarity, there are several differences of note between the Spring and shakein approaches.

Advice in Spring is manifested as an interceptor class, which is invoked whenever an advised method is executed. Pointcuts are also manifested as classes, and interrogated at run time to find out which methods should be advised. Much as in JBoss, the

[19] http://www.martinfowler.com/articles/injection.html
[20] http://www.hibernate.org
[21] Spring also supports the integration of standard ASPECTJ aspects.

mechanism relies on a sophisticated combination of libraries and configuration files, with no changes to the language itself. Therefore, Spring AOP shares much of the tolls noted for JBoss AOP, including similar space and time complexities (with the exception of hooks-induced slowdowns). Additional performance penalties are caused by the need to evaluate pointcuts at run time, as well as the run-time generation of subclasses.

As a design decision, Spring AOP only supports method invocation joinpoints (and that, only for non-**`private`** methods). In our tests, the lack of support for field access joinpoints implied that the `Confined` aspect had to be made explicitly aware of each of the `Point` methods that can update the point's coordinates; in particular, the advice had to re-create the logic for the `moveBy` method. The Spring point of view contends that this would not have been needed, had `moveBy` relied on `setX` and `setY` to update the fields, rather than using direct access (recall Fig. 2a). But from this very claim we must conclude that the Spring aspect is fragile with respect to changes in the implementation of `Point`; should `moveBy` be updated to rely on the setter methods, the advice must be accordingly updated. A nonfragile aspect implementation must rely on examining the control-flow at run time, with a noticeable performance hit.

In our benchmarks (Fig. 8), the Spring-based `Confined` aspect (which was not created using composition, due to its asymmetry with regard to `moveBy`) was over twice as fast as the JBoss version, but still much slower than the shakeins-based version.

5 Weaving, Deployment, and Deploy-Time Weaving

Now that the theoretical foundation of the shakeins construct was established, and that we understand how and why it may be useful in the context of middleware frameworks, the time has come to combine the two. The first step is in describing how deployment, a basic technique of J2EE, can be generalized for the process of weaving aspects (specifically, shakeins) into an application.

Section 5.1 explains weaving. Deployment is the subject of Sect. 5.2. Weaving of shakeins onto EJBs is discussed in Sect. 5.3. Section 5.4 generalizes this process to arbitrary classes.

5.1 Weaving

Weaving is the process of inserting the relevant code from various aspects into designated locations, known as *joinpoints*, in the main program. In their original presentation of ASPECTJ [9], Kiczales et al. enumerate a number of weaving strategies: "*aspect weaving can be done by a special pre-processor, during compilation, by a post-compile processor, at load time, as part of the virtual machine, using residual runtime instructions, or using some combination of these approaches.*" Each of these weaving mechanisms was employed in at least one AOP language implementation. As mentioned before, our implementation of shakeins use its own peculiar *deploy-time weaving* strategy. In this section we motivate this strategy and explain it in greater detail.

We first note that the weaving strategies mentioned in the above quote transgress the boundaries of the standard object model. Patching binaries, preprocessing, dedicated loaders or virtual machines will confuse tools such as debuggers, profilers, and static analyzers, and may have other adverse effects on generality and portability.

Further, weaving introduces a major *conceptual* bottleneck. As early as 1998, Walker et al. [30] noted the disconcert of programmers when realizing that merely reading a unit's source code is not sufficient for understanding its run-time behavior.[22]

The remedy suggested by Constantinides et al. in the *Aspect Moderator* framework [31] was restricting weaving to the dominion of the OOP model. In their suggested framework, aspects and their weaving are realized using pure object-oriented constructs. Thus, every aspect-oriented program can be presented in terms of the familiar notions of inheritance, polymorphism, and dynamic binding. Indeed, as Walker et al. conclude: "*programmers may be better able to understand an aspect-oriented program when the effect of aspect code has a well-defined scope*".

Aspect Moderator relies on the PROXY design pattern [21] to create components that can be enriched by aspects. Each core class has a proxy which manages a list of operations to be taken before and after every method invocation. As a result, joinpoints are limited to method execution only, and only **`before`**`()` and **`after`**`()` advices can be offered. Another notable drawback of this weaving strategy is that it is *explicit*, in the sense that every advice has to be manually registered with the proxy. Registration is carried out by issuing a plain JAVA instruction—there are no external or non-JAVA elements that modify the program's behavior. Therefore, long, tiresome, and error-prone sequences of registration instructions are typical to Aspect Moderator programs.

A better strategy of implementing explicit weaving is that this code is generated by an automatic tool from a concise specification. The shakein weaving mechanism gives in essence this tool. However, rather than generate explicit weaving code for a proxy, it generates a woven version of the code in a *newly generated subclass*. By replacing the proxy pattern with the notion of subclassing, it also able to handle advice types other than **`before`**`()` and **`after`**`()`, and handle a richer gamut of joinpoint types, as detailed in Sect. 6.3.

Thus, shakeins do not use any of the obtrusive weaving strategies listed above. Instead, the mechanism employs a weaving strategy that *does not break* the object model. Instead of modifying binaries (directly, or by preprocessing the source code), the application of a shakein to a class results in an "under the hood" generation of a new class that inherits from, rather than replaces, the original. The new class provides an alternative realization, a re-implementation, of the same type; it does not introduce a new type, since there are no visible changes to the interface. This re-implementation is generated by advising the original one with the advice contained in the shakein.

Clearly, there are limitations to the approach of implementing shakeins as subclasses. The main such limitation is that Proposition 1 does not hold in the general case. Below we will show that in the particular case of EJBs in J2EE, this restriction does not arise, because access to EJBs is through interfaces.

5.2 Deployment

J2EE offers a unique opportunity for generating the subclasses required for the weaving of shakeins. Figure 9 compares the development cycle of traditional and J2EE application.

[22] Further, Laddad [8, p. 441] notes that in ASPECTJ the run-time behavior cannot be deduced even by reading *all* aspects, since their application to the main code is governed by the command by which the compiler was invoked.

We see that *deployment* is a new stage in the program development process, which occurs after compilation but prior to execution. It is unique in that although new code is generated, it is not part of the development, but rather of user installation.

Deployment is the magic by which J2EE *services*, such as security and transaction management, are welded to applications. The generation of subclasses and support classes is governed by *deployment descriptors*, which are XML configuration files.

The idea behind deploy-time weaving is to extend this magic, by placing the shakein semantics in government of this process. As shakeins are based on straightforward inheritance, this extension also simplifies the structure of and inter-relationships between the generated support classes.

Technically, *deployment* is the process by which an application is installed on a J2EE application server. Having received the application binaries, deployment involves generating, compiling, and adding additional support classes to the application. For example, the server generates *stub* and *tie* (skeleton) classes for all classes that can be remotely accessed, in a manner similar to, or even based on, the RMI compiler.[23] Even though some J2EE application servers generate support class binaries directly (without going through the source), these always conform to the standard object model.

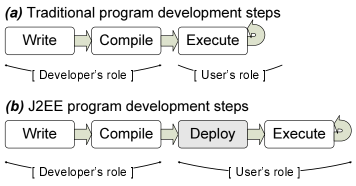


Fig. 9. Program development steps: **(a)** traditional, **(b)** J2EE

We must study some of the rather mundane details of deployment in order to understand how it can be generalized to do weaving. To do so, consider first Fig. 10, which shows the initial hierarchy associated with an ACCOUNT CMP bean (see Sect. 1.2).

At the right-hand side of the figure, we see interface `Account`, which inherits from `javax.ejb.EJBObject`. This interface is written by the developer in support of the remote interface to the bean.[24] This is where all client-accessible methods are declared. In the example, there are three such methods: `withdraw()`, `deposit()`, and `getBalance()`. Class `Account` resides at the *client side*.

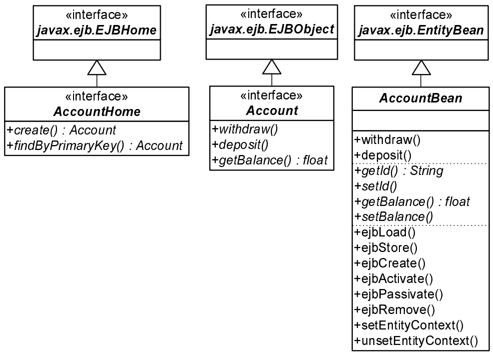


Fig. 10. Programmer-created classes for the ACCOUNT EJB

On the left-hand side of the figure, we see abstract class `AccountBean`, inheriting from `javax.ejb.EntityBean`. The J2EE developer's main effort is in coding this class, which will reside at the *server side*. There are three groups of methods in the bean class:

[23] http://java.sun.com/j2se/1.5.0/docs/guide/rmi/

[24] For the sake of simplicity, we assume that ACCOUNT has a remote interface only, even though beans can have either a local interface, a remote interface, or both.

(1) *Business Logic.* The first group of methods in this class consists of the implementation of business logic methods. These are `deposit()` and `withdraw()` in the example.
(2) *Accessors of Attributes.* EJB *attributes* are those fields of the class that will be governed by the persistence service in the J2EE server. Each attribute *`attr`* is represented by abstract setter and getter methods, called `set`*`Attr`*`()` and `get`*`Attr`*`()`, respectively. Attributes are not necessarily client accessible.
In the example, there are four such accessors, indicating that the bean ACCOUNT has two attributes: `id` (the primary key) and `balance`. Examining the `Account` interface we learn that `id` is invisible to the client, while `balance` is read-only accessible.
(3) *Lifecycle.* The third and last method group comprises a long list of mundane life-cycle methods, such as `ejbLoad()` and `ejbStore()`, most of which are normally empty. Even though sophisticated IDEs can produce a template implementation of these, they remain a developer's responsibility, contaminating the functional concern code. Later we shall see how deploy-time weaving can be used to remove this burden.

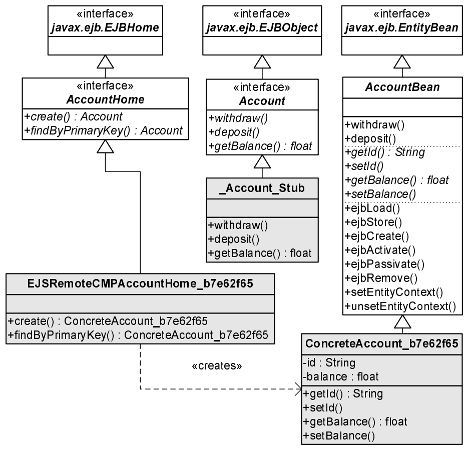


Fig. 11. ACCOUNT classes defined by the programmer, and support classes (in gray) generated by WAS 5.0 during deployment

Finally, at the center of Fig. 10, we see interface `AccountHome`, which declares a FACTORY [21] of this bean. Clients can only generate or obtain instances of the bean by using this interface.

Concrete classes to implement `AccountHome`, `Account`, and `AccountBean` are generated at deployment time. The specifics of these classes vary with the J2EE implementation. Figure 11 shows some of the classes generated by IBM's WebSphere Application Server (WAS)[25] version 5.0 when deploying this bean.

Examining the figure, we see that it is similar in structure to Fig 10, except for the classes, depicted in gray, that the deployment process created: `ConcreteAccount_b7e62f65` is the *concrete bean class*, implementing the abstract methods defined in `AccountBean` as setters and getters for the EJB attributes. Instances of this class are handed out by class `EJSRemoteCMPAccountHome_b7e62f65`, which implements the factory interface `AccountHome`.

Finally, `_Account_Stub`, residing at the client side, intercommunicates with `ConcreteAccount_b7e62f65` which resides at the server side.

In support of the ACCOUNT bean, WAS deployment generates several additional classes which are not depicted in the figure: a stub for the home interface, ties for both stubs, and more. Together, the deployment classes realize services that the container provides to the bean: persistence, security, transaction management, and so forth. However, as evident from the figure, all this support is provided within the standard object oriented programming model.

[25] http://www.ibm.com/software/websphere/

5.3 Deployment as a Weaving Process for EJBs

Having understood the process of deployment and the generation of classes in it, we can now explain how deployment can be used as a weaving process. Consider first the ordered application of four standard ASPECTJ2EE shakeins: `Lifecycle`, `Persistence`, `Security`, and `Transactions` to the bean ACCOUNT. (Such a case is easier than the more general case, in which the target class is not an EJB. We will discuss this issue below.)

Weaving by deployment generates, for each application of an aspect (or a shakein) to a class, a subclass of the target. This subclass is called an *advised class*, since its generation is governed by the advices given in the aspect. Accordingly, the sequence of applications under consideration will generate four advised classes.

Figure 12 shows the class hierarchy after the deployment tool generated these four class in support of the shakein application expression

```
Transactions<Security<Persistence<Lifecycle<Account>>>>.
```

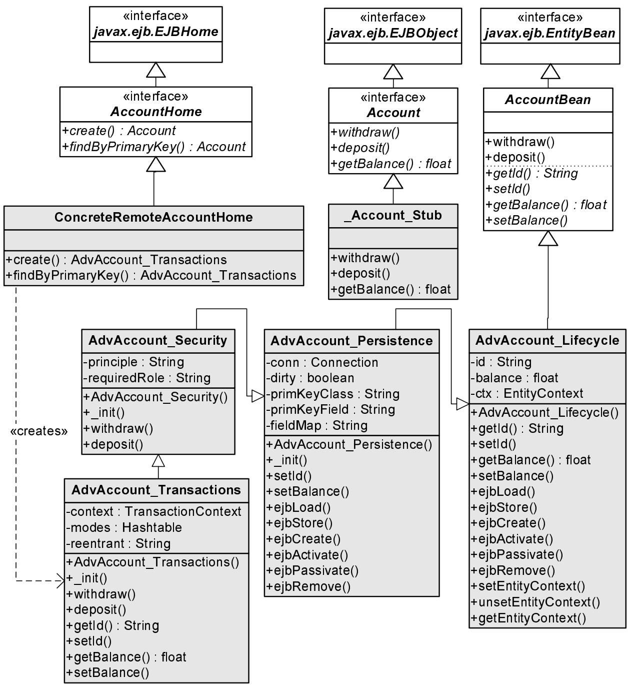


Fig. 12. ACCOUNT classes defined by the programmer, and support classes (in gray) generated by ASPECTJ2EE deployment

Comparing this figure to Fig. 11, we see first that the class `AccountBean` was shortened by moving the life-cycle methods to a newly defined class, `AdvAccount_Lifecycle` The shakein `Lifecycle` made it possible to eliminate the tiring writing of token (and not always empty) implementations of the life-cycle methods in each bean. All these are packaged together in a standard `Lifecycle` aspect.[26]

`AdvAccount_Lifecycle` is the advised class realizing the application of `Lifecycle` to ACCOUNT. There are three other advised classes in the figure, which correspond to the application of aspects `Persistence`, `Security`, and `Transactions` to ACCOUNT.

[26] The life-cycle methods are declared in the interface `javax.ejb.EntityBean`. Hence, implementing them in a shakein does not change the type of class `AccountBean`.

The sequence of aspect applications is translated into a chain of inheritance of advised classes, starting at the main bean class. The *root advised class* is the first class in this chain (`AdvAccount_Lifecycle` in the example), while the *terminal advised class* is the last (`AdvAccount_Transactions` in the example). Fields, methods, and inner classes defined in an aspect are copied to its advised class. *Advised methods* in this class are generated automatically based on the advices in the aspect.

We note that although all the advised classes are concrete, only instances of the terminal advised class are created by the bean factory (the generated EJB home). In the figure for example, class `ConcreteRemoteAccountHome` creates all ACCOUNTs, which are always instances of `AdvAccount_Transactions`.

It may be technically possible to construct instances of this bean in which fewer aspects are applied; there are, however, deep theoretical reasons for preventing this from happening. Suppose that a certain aspect applies to a software module such as a class or a routine, etc., in all but some exceptional incarnations of this module. Placing the tests for these exceptions at the point of incarnation (routine invocation or class instantiation) leads to scattered and tangled code, and defeats the very purpose of AOP. The bold statement that some accounts are exempt from security restrictions should be made right where it belongs—as part of the definition of the security aspect! Indeed, J2EE and other middleware frameworks do not support conditional application of services to the same business logic. A simple organization of classes in packages, together with JAVA accessibility rules, enforce this restriction and prevents clients from obtaining instances of nonterminal advised classes.

5.4 Deploy-Time Weaving for General Classes

We just saw that deploy-time weaving generates, at deployment time, an advised class for each application of an aspect of ASPECTJ2EE. Let us now consider the more general case, in which the target class is not an EJB.

It is instructive to compare the advising of EJBs (Fig. 12) with the general structure of shakein classes, as depicted in Fig. 3. We see that the diagrams are similar in making each aspect application into a JAVA class. However, Fig. 12 adds two factors to the picture: First, the generation of instances of ACCOUNT is controlled by an external *factory class*. Second is the fact that the class `Account` is abstract.

Together these two make one of the key properties of EJBs, namely the fact that an EJB does not have a *forge facet*. Instead, the framework imposes a requirement that all instances of the class are obtained from an external class, which follows the ABSTRACT FACTORY design pattern.

This property makes it possible to apply an aspect, a service, or a shakein to *all* instances of a certain class. When applying the deploy-time weaving technique to non-EJB classes, one may chose to degenerate the forge facet of the target class, as in EJBs, or in the case that this is not possible, make sure that the correct constructors are invoked in the code.

6 The ASPECTJ2EE Programming Language

Having described the shakeins construct and deploy-time weaving, we are ready to describe the ASPECTJ2EE language.

The syntax of ASPECTJ2EE is a variant of ASPECTJ. The semantics of ASPECTJ2EE-aspects is based on a (limited) implementation of the shakein concept. Hence, aspects in ASPECTJ2EE (unlike in ASPECTJ) do not have a global effect, and are woven into the application at deployment time (rather than compile time).

When compared to shakeins, the main limitation of ASPECTJ2EE-*aspects* (henceforth just "aspects", unless noted otherwise), is that their application to classes is governed by an external *deployment descriptor file*, written in XML. Accordingly, ASPECTJ2EE does not provide a syntax for explicitly applying aspects to classes. Consequently, the integration of aspects into ASPECTJ2EE is not complete. Indeed, ASPECTJ2EE suffers from two XML-JAVA coupling issues: (1) JAVA code using a class whose generation is governed by XML is coupled with this XML code. (2) XML file applying an aspect to a ASPECTJ2EE class is coupled with the ASPECTJ2EE names. However, in contrast with JBoss aspects, the detection of errors due to such coupling, i.e., using wrong class names or illegal or empty pointcut expressions, is not at run time, but rather at deployment time.

Comparing the ASPECTJ2EE version of shakeins with the theoretical description of the concept, we find that some of the benefits (see Sect. 2.4) are preserved, while others are not:

(1) *Selective application* is available; aspects are applied only to classes specified in the deployment descriptor.
(2) *Nondestructive application* is preserved. However, instances are obtained using Home objects (factories) only. Therefore, the programmer, wearing the hat of an *application assembler*, can dictate which combinations of aspect application are available. For example, it is possible to ensure that all instances of ACCOUNT are subjected to a security aspect.
(3) *Explicit and Flexible Ordering* is provided by the XML binding language.
(4) *Composition* is not supported; there is no syntax for composing two or more aspects.
(5) *Configuration parameters* are available; the deployment descriptor is used for argument passing.
(6) *Repeated application* is fully supported.
(7) *Parameter checking* is absent.

Section 6.1 presents the language syntax. The application of aspects through deployment descriptors is the subject of Sect. 6.2. Section 6.3 explains how deploy-time weaving can implement the various kinds of joinpoints. Finally, Sect. 6.4 gives a broad overview of the standard aspect library.

6.1 Language Syntax

The major difference between ASPECTJ2EE and ASPECTJ is that ASPECTJ2EE supports parameterized aspects. For example, Fig. 13 shows the definition of a role-based security shakein that accepts two parameters. The first parameter is a pointcut definition specifying which methods are subjected to a security check. The second is the user role-name that the check requires.

```
aspect Security[pointcut secured(), String requiredRole] {
   before(): secured() {
      if (!userInRole(requiredRole)) {
         throw new RuntimeException("Security Violation");
      }
   }

   private boolean userInRole(String roleName) {
      // Check if the currently active user has the given role...
   }
}
```

Fig. 13. The definition of a simple `Security` aspect in ASPECTJ2EE

Parameter values must be known at deploy time. Accordingly, there are four kinds of parameters for aspects:

- *Type parameters*, preceded by the keyword **`class`**. The type can be restricted (like type parameters in JAVA generics) using the **`implements`** and **`extends`** keywords.
- *Pointcut parameters*, preceded by the **`pointcut`** keyword.
- *String parameters* and *primitive type parameters*, preceded by the type name (`String`, **`int`**, **`boolean`**, etc.).

In contrast with ASPECTJ, the scope of a specific aspect application in ASPECTJ2EE is limited to its target class. Therefore, any pointcut that refers to joinpoints in other classes is meaningless. Accordingly, ASPECTJ2EE does not have a **`call`** joinpoint, since it refers to the calling point, rather than the execution point, of a method. (To apply advice to method execution, an **`execution`** joinpoint can be used.) This restriction is a direct result of the shakein semantic model, and it eliminates the confusion associated with the **`call`** joinpoint in relation to inheritance (see Sect. 2.2)

All other joinpoint kinds are supported, but with the understanding that their scope is limited to the target class; for example, a field-set joinpoint for a **`public`** field will not capture access to the field from outside its defining class. ASPECTJ2EE also introduces a new kind of joinpoint for handling remote invocation of methods.

Since the application of aspects in ASPECTJ2EE is explicit, it does not recognize the ASPECTJ statement **`declare precedence`**.

Finally, there is a subtle syntactical difference due to the "individual target class" semantics of ASPECTJ2EE aspects: The definition of a pointcut should not include the target class name as part of method, field, or constructor signatures. Only the member's name, type, access level, list of parameter types, etc., can be specified. For example, the signature matching any **`public void`** method accepting a single `String` argument is written ASPECTJ as `public void *.*(String)` . The same signature should be written as `public void *(String)` in ASPECTJ2EE. The ASPECTJ form applies to the methods with this signature in *all* classes, whereas the ASPECTJ2EE form applies only to such methods in the class to which the containing aspect is applied.

6.2 The Deployment Descriptor

In ASPECTJ, the application of aspects to classes is specified declaratively. Yet the process is not completely transparent: the application assembler must take explicit actions

```
<entity id="Account">
   <ejb-name>Account</ejb-name>
   <home>aspectj2ee.demo.AccountHome</home>
   <remote>aspectj2ee.demo.Account</remote>
   <ejb-class>aspectj2ee.demo.AccountBean</ejb-class>
   <apply>
      <aspect>aspectj2ee.core.Lifecycle</aspect>
   </apply>
   <apply>
      <aspect>aspectj2ee.core.Persistence</aspect>
      <parameter name="primKeyClass">java.lang.String</parameter>
      <parameter name="primKeyField">serialNumber</parameter>
      <parameter name="table">ACCOUNTS</parameter>
      <parameter name="fieldMap">serialNumber:SERIAL, balance:BALANCE</parameter>
   </apply>
   <apply>
      <aspect>aspectj2ee.core.Security</aspect>
      <parameter name="secured">execution(*(..))</parameter>
      <parameter name="requiredRole">User</parameter>
   </apply>
   <apply>
      <aspect>aspectj2ee.core.Transactions</aspect>
      <parameter name="reentrant">false</parameter>
      <parameter name="requiresnew">execution(deposit(..)) ||
                                   execution(withdraw(..))</parameter>
      <parameter name="required">execution(*(..)) && !requiresnew()</parameter>
   </apply>
</entity>
```

Fig. 14. A fragment of an EJB's deployment descriptor specifying the application of aspects to the ACCOUNT bean

to make sure that the specified aspect application actually takes place. In particular, he must remember to compile each core module with all the aspects that may apply to it. (Or else, an aspect with global applicability may not apply to certain classes if these classes were not compiled with it.)

The order of application of aspects in ASPECTJ is governed by **`declare precedence`** statements; without explicit declarations, the precedence of aspects in ASPECTJ is undefined. Also, ASPECTJ does not provide any means for passing parameters to the application of aspects to modules.

In contrast, the shakeins semantics in general, and ASPECTJ2EE in particular, require an explicit specification of each application of an aspect to a class, along with any configuration parameters. This specification could have been done as part of the programming language. But, following the conventions of J2EE, and in the sake of minimizing the syntactical differences between ASPECTJ2EE and ASPECTJ, we chose to place this specification in an *external* XML deployment descriptor.

In fact, we shall see that the XML specification is in essence the abstract syntax tree, which would have been generated from parsing the same specification if written inside the programming langauge. Figure 14 gives an example, showing the sequence of application of aspects to ACCOUNT which generated the classes in Fig. 12. Overall, four aspects are applied to the bean: `Lifecycle`, `Persistence`, `Security`, and `Transactions`. All of these are drawn from the `aspectj2ee.core` aspect library.

The figure shows the XML element describing bean ACCOUNT. (In general, the deployment descriptor contains such entities for each of the beans, along with other information.) We follow the J2EE convention, in that the bean is defined by the `<entity>` XML element.

Element `<entity>` has several internal elements. The first four: `<ejb-name>`, `<home>`, `<remote>`, and `<ejb-class>`, specify the JAVA names that make this bean. These are part of J2EE and will not concern us here.

Following are elements of type `<apply>`, which are an ASPECTJ2EE extension. Each of these specifies an application of an aspect to the bean.

An `<apply>` element has two kinds of internal elements:

(1) *`<aspect>`*, naming the aspect to apply to the bean.
For example, element `<aspect>aspectj2ee.core.Lifecycle</aspect>` in the figure specifies that `aspectj2ee.core.Lifecycle` is applied to ACCOUNT.
(2) *`<parameter>`*, specifying the configuration parameters passed to the aspect.
Consider for example the `Security` aspect (Fig. 13). In Fig. 14, we see that the actual value for the `secured` pointcut formal parameter is **`execution`**`(*(..))` (i.e., the execution of any method). Similarly, formal string parameter `required-Role` was actualized with value *`"User"`*.
Thus, the third `<apply>` element is tantamount to configuring the aspect with the following pseudo-syntax: `Security[`**`execution`**`(*(..)), "User"]`.

In support of *explicit and flexible ordering*, the order of `<apply>` elements specifies the order by which aspects are applied to the bean. Intra-aspect precedence (where several advices from the same aspect apply to a single joinpoint) is handled as in ASPECTJ, i.e., by order of appearance of advices.

We can generalize the example above to write the entire sequence of application of aspects to the bean, along with their parameters. In total, there are nine such parameters. These, together with the aspect names, would have made the programming language equivalent of the application sequence in Fig. 14 cumbersome and error prone. We found that the XML notation is a convenient replacement to developing syntax for dealing with this unwieldiness.

Note that ACCOUNT can be viewed as an entity bean with container-managed persistence (CMP EJB) simply because it relies on the core persistence aspect, which parallels the standard J2EE persistence service. Should the developer decide to use a different persistence technique, that persistence system would itself be defined as an ASPECTJ2EE aspect, and applied to ACCOUNT in the same manner. This is parallel to bean-managed persistence beans (BMP EJBs) in the sense that the persistence logic is provided by the application programmer, independent of the services offered by the application server. However, it is completely unlike BMP EJBs in that the persistence code would not be tangled with the business logic and scattered across several bean and utility classes. In this respect, ASPECTJ2EE completely dissolves the distinction between BMP and CMP entity beans.

6.3 Implementing Advice by Subclassing

ASPECTJ2EE supports each of the joinpoint kinds defined in ASPECTJ, except for **`call`**, since **`call`** advice is applied at the *client* (caller) site and not to the main class.

We next describe advice are woven into the entity bean code in each supported kind of joinpoint.

Execution Join Points. The **execution**(*methodSignature*) joinpoint is defined when a method is invoked and control transfers to the target method. ASPECTJ2EE captures **execution** joinpoints by generating advised methods in the advised class, overriding the inherited methods that match the execution joinpoint. Consider, for example, the advice in Fig. 15a, whose pointcut refers to the execution of the `deposit()` method. This is a **before**() advice which prepends a printout line to matched joinpoints. When applied to ACCOUNT, only one joinpoint, the execution of `deposit()`, will match the specified pointcut. Hence, in the advised class, the `deposit()` method will be overridden, and the advice code will be inserted prior to invoking the original code. The resulting implementation of `deposit()` in the advised class appears in Fig. 15b.

```
(a) before(float amount): execution(deposit(float)) && args(amount) {
        System.out.println("Depositing " + amount);
    }

(b) void deposit(float amount) {
        System.out.println("Depositing " + amount);
        super.deposit(amount);
    }
```

Fig. 15. a Sample advice for `deposit()` execution, and **b** the resulting advised method

Recall that only instances of the terminal advised class exist in the system, so every call to the advised method (`deposit()` in this example) would be intercepted by means of regular polymorphism. Overriding and refinement can be used to implement **before**(), **after**() (including **after**() **returning** and **after**() **throwing**), and **around**() advice. With **around**() advice, the **proceed** keyword would indicate the location of the call to the inherited implementation.

The example in Fig. 16 demonstrates the support for **after**() **throwing** advice. The advice, listed in part (a) of the figure, would generate a printout if the `withdraw()` method resulted in an `InsufficientFundsException`. The exception itself is rethrown, i.e., the advice does not swallow it. The resulting advised method appears in part (b) of the figure. It shows how **after**() **throwing** advice are implemented by encapsulating the original implementation in a **try/catch** block.

An **execution** joinpoint may refer to **private** methods. Since such methods cannot be overridden in subclasses, the ASPECTJ2EE weaver generates a new, advised version of the method—and then overrides any method that *invokes* the private method, so that the callers will use the newly generated version of the private callee rather than the original. The overriding version of the callers includes a complete re-implementation of each caller's code, rather than using refinement, so that only the new version of the callee will be used. The only exception is where a private method is invoked by a constructor, which cannot be replaced by an overriding version. ASPECTJ2EE will issue a warning in such cases.

```
(a) after() throwing (InsufficientFundsException ex)
     throws InsufficientFundsException:
     execution(withdraw(..)) {
        System.out.println("Withdrawal failed: " + ex.getMessage());
        throw ex;
     }

(b) void withdraw(float amount) throws InsufficientFundsException {
        try { super.withdraw(amount);
        } catch (InsufficientFundsException ex) {
           System.out.println("Withdrawal failed: " + ex.getMessage());
           throw ex;
        }
     }
```

Fig. 16. a Sample **after**() **throwing** advice, applied to a method execution join point, and **b** the resulting advised method

This technique is used not only with **execution** joinpoints, but whenever an advice apples to code inside a **private** method (e.g., when a field access joinpoint is matched by code inside one).

A similar problem occurs with **final** and **static** methods. However, such methods are disallowed by the J2EE specification and may not be included in EJB classes.

Constructor Execution Joinpoints. The constructor execution joinpoint in ASPECTJ is defined using the same keyword as regular method execution. The difference lies in the method signature, which uses the keyword **new** to indicate the class's constructor. For example, the pointcut **execution**(**new**(..)) would match the execution of any constructor in the target class.

Unlike regular methods, constructors are limited with regard to the location in the code where the inherited implementation (`super()`) must be invoked. The invocation must be the first statement of the constructor, and in particular it must occur before any field access or virtual method invocation. Hence, joinpoints that refer to constructor signatures can be advised, but any code that executes before the inherited constructor (**before**() advice, or parts of **around**() advice that appear prior to the invocation of **proceed**()) is invalid.

An **around**() advice for constructor execution that does not contain an invocation of **proceed**() would be the equivalent of a JAVA constructor that does not invoke **super**() (the inherited constructor). This is tantamount to having an implicit call to **super**().

Field Read and Write Access Joinpoints. Field access joinpoints match references to and assignments of fields. ASPECTJ2EE presents no limitations on advice that can be applied to these joinpoints. However, if a field is visible *outside* of the class (e.g., a **public** field), then any *external* access will bypass the advice. It is therefore recommended that field access will be restricted to **private** fields and EJB *attributes* only. Recall that attributes are not declared as fields; rather, they are indicated by the programmer using **abstract** getter and setter methods. These methods are then implemented in the concrete bean class (in J2EE) or in the root advised class (in ASPECTJ2EE).

```
(a) around(): remotecall(deposit(..)) {
        System.out.println("About to perform transaction.");
        proceed();
        System.out.println("Transaction completed.");
    }

(b) public void deposit(float arg0) {
        System.out.println("About to perform transaction.");
        // ... normal RMI/IIOP method invocation code ...
        System.out.println("Transaction completed.");
    }
```

Fig. 17. a Sample advice for a method's **`remotecall`** join point, and **b** the resulting `deposit()` method generated in the RMI stub class

If no advice is provided for a given attribute's read or write access, the respective method implementation in the root advised class would simply read or update the class field. The field itself is also defined in the root advised class. However, an attribute can be advised using **`before`**`()`, **`around`**`()`, and **`after`**`()` advice, which would affect the way the getter and setter method are implemented.

If an advice is applied to a field (which is not an attribute), all references to this field by method in the class itself are advised by generating overriding versions of these methods. However, since a **`private`** field is not visible even to subclasses, this might require generating a new version of the field, which *hides* [32, Sect. 8.3.3] the original declaration. In such cases, any method that accesses the field must be regenerated, even where the advice does not cause any code alteration, so that the overriding version will access the new field.

Exception Handler Joinpoints. The **`handler`** joinpoint can be used to introduce advice into **`catch`** blocks for specific exception types. Since the **`catch`** block per se cannot be overridden, advising such a joinpoint results in a new, overriding version of the entire advised method. Most of the code remains unchanged from the original, but the code inside the **`catch`** block is altered in accordance with the advice.

Remote Call Joinpoints. The **`remotecall`** joinpoint designator is a new keyword introduced in ASPECTJ2EE. Semantically, it is similar to ASPECTJ's **`call`** joinpoint designator, defining a joinpoint at a method invocation site. However, it only applies to remote calls to various methods; local calls are unaffected.

Remote call joinpoints are unique, in that their applied advice does not appear in the advised subclass. Rather, they are implemented by affecting the stub generated at deploy time for use by EJB clients (such as `_Account_Stub` in Fig. 12). For example, the **`around`**`()` advice from Fig. 17a adds printout code both before and after the remote invocation of `Account.deposit()`. The generated stub class would include a `deposit()` method like the one shown in part (b) of that figure. Since the advised code appears in the stub, rather than in a server-side class, the output in this example will be generated by the client program.

Remote call joinpoints can only refer to methods that are defined in the bean's remote interface. Advice using **`remotecall`** can be used to localize tier-cutting concerns, as detailed in Sect. 7.

Control-Flow Based Pointcuts. ASPECTJ includes two special keywords, **cflow** and **cflowbelow**, for specifying control-flow based limitations on pointcuts. Such limitations are used, for example, to prevent recursive application of advice. Both keywords are supported by ASPECTJ2EE.

The manner in which control-flow limitations are enforced relies on the fact that deployment can be done in a completely platform-specific manner, since at deploy time, the exact target platform (JVM implementation) is known. Different JVMs use different schemes for storing a stack snapshot in instances of the `java.lang.Throwable` class[27] (this information is used, for example, by the method `java.lang.Exception.printStackTrace()`). Such a stack snapshot (obtained via an instance of `Throwable`, or any other JVM-specific means) can be examined in order to test for **cflow/cflowbelow** conditions at runtime.

An alternative implementation scheme relies on `ThreadLocal`[28] objects. The advice application would result in a `ThreadLocal` flag that will be turned on or off as various methods are entered or exited. At the target joinpoint, the flag's value will be examined to determine if the **cflowbelow** condition holds, and the advice should be executed. The one-instance-per-thread nature of `ThreadLocal` objects ensures that this flag-based scheme will function properly in multithreaded application.

6.4 The Core Aspects Library

ASPECTJ2EE's definition includes a standard library of core aspects. Four of these aspects were used in the ACCOUNT example, as shown in Fig. 12. Here is a brief overview of these four, and their effect on the advised classes:

(1) The `aspectj2ee.core.Lifecycle` aspect (used to generated the root advised class) provides a default implementation to the J2EE life-cycle methods. The implementations of `setEntityContext()`, `unsetEntityContext`, and `getEntityContext()` maintain the entity context object; all other methods have an empty implementation. These easily available common defaults make the development of EJBs somewhat easier (compared to standard J2EE development); the user-provided `AccountBean` class is now shorter, and contains strictly business logic methods.[29]
(2) The `aspectj2ee.core.Persistence` aspect provides a CMP-like persistence service. The attribute-to-database mapping properties are detailed in the parameters passed to this aspect in the deployment descriptor. This aspect advises some of the life-cycle methods, as well as the attribute setters (for maintaining a "dirty" flag), hence these methods are all overridden in the advised class.
(3) The `aspectj2ee.core.Security` aspect can be used to limit the access to various methods based on user authentication. This is a generic security solution, on par with the standard J2EE security service. More detailed security decisions, such

[27] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Throwable.html
[28] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ThreadLocal.html
[29] The fact that the fields used to implement the attributes, and the concrete getter and setter method for these attributes, appear in `AdvAccount_Lifecycle` (in Fig. 12) stems from the fact that this is the root advised class, and is not related to the `Lifecycle` aspect per se.

as role-based variations on method behavior, can be defined using project-specific aspects without tangling security-related code with the functional concern code.

(4) Finally, the `aspectj2ee.core.Transactions` aspect is used to provide transaction management capabilities to all business-logic methods. The parameters passed to it dictate what transactional behavior will be applied to each method. Transactional behaviors supported by the J2EE platform include methods that must execute within a transaction context, and will create a new transaction if none exists; methods that must execute within an existing transaction context; methods that are neutral to the existence of a transaction context; and methods that will fail to run within a transaction context. The list of methods that belong to each group is specified with a pointcut parameter passed to this aspect.

7 Innovative Uses for AOP in Multitier Applications

The use of aspects in multitier enterprise applications can reduce the amount of cross-cutting concerns and tangled code. As discussed in Sect. 3, the core J2EE aspects were shown to be highly effective to this end, and the ability to define additional aspects (as well as alternative implementations to existing ones) increases this effectiveness and enables better program modularization.

But AspectJ2EE also allows developers to confront a different kind of cross-cutting nonfunctional concerns: aspects of the software that are implemented in part on the client and in part on the server. Here, the cross-cutting is extremely acute as the concern is implemented not just across several classes and modules, but literally across programs. We call these *tier-cutting concerns*. In the context of AspectJ2EE, tier-cutting concerns are applied to the business methods of EJBs.

The notion of remote pointcuts was independently discovered by Nishizawa et al. [33].

The remainder of this section shows that a number of several key tier-cutting concerns can be represented as single aspect by using the **`remotecall`** joinpoint designator. In each of these examples, the client code is unaffected; it is the RMI stub, which acts as a proxy for the remote object, which is being modified.

7.1 Client-Side Checking of Preconditions

Method preconditions [34] are commonly presented as a natural candidate for nonfunctional concerns being expressed cleanly and neatly in aspects. This allows preconditions to be specified without littering the core program, and further allows precondition testing to be easily disabled.

Preconditions should normally be checked at the method execution point, i.e., in the case of multitier applications, on the server. However, a precondition defines a contract that binds whoever invokes the method. Hence, by definition, precondition violations can be detected and flagged at the invocation point, i.e., on the client. In a normal program, this matters very little; but in a multitier application, trapping failed preconditions on the client can prevent the round-trip of a remote method invocation, which incurs a heavy overhead (including communications, parameter marshaling, and un-marshaling, etc.).

Figure 18 presents a simple precondition that can be applied to the ACCOUNT EJB: neither `withdraw()` nor `deposit()` are ever supposed to be called with a nonpositive amount as a parameter. If such an occurrence is detected, a `PreconditionFailed-Exception` is thrown. Using two named pointcut definitions, the test is applied both at the client and at the server.

In addition to providing a degree of safety, such aspects decrease the server load by blocking futile invocation attempts. In a trusted computing environment, if the preconditioned methods are invoked only by clients (and never by other server-side methods), the server load can be further reduced by completely disabling server-side tests.

```
public aspect EnsurePositiveAmounts {
   pointcut clientSide(float amount):
      (remotecall(public void deposit(float)) ||
       remotecall(public void withdraw(float))) && args(amount);

   pointcut serverSide(float amount):
      (execution(public void deposit(float)) ||
       execution(public void withdraw(float))) && args(amount);

   before(float amount): clientSide(amount) || serverSide(amount) {
      if (amount <= 0.0)
         throw new PreconditionFailedException("Non-positive amount: "+amount);
   }
}
```

Fig. 18. An aspect that can be used to apply precondition testing (both client- and server-side) to the ACCOUNT EJB

When using aspects to implement preconditions, always bear in mind that preconditions test for logically flawed states, rather than states that are unacceptable from a business process point of view. Thus, preventing the withdrawal of excessive amounts should be part of `withdraw()`'s implementation rather than a precondition.

7.2 Symmetrical Data Processing

By adding code both at the sending and receiving ends of remotely invoked methods, we are able to create what can be viewed as an additional layer in the communication stack. For example, we can add encryption at the stub and decryption at the remote tie; or we can apply a compression scheme (compressing information at the sender, decompressing it at the receiver); and so forth.

Consider an EJB representing a university course, with the method `register()` accepting a `Vector` of names of students (`String`s) to be registered to that course. The aspect in Fig. 19 shows how the remote invocation of this method can be made more effective by applying compression. Assume that the class `CompressedVector` represents a `Vector` in a compressed (space-efficient) manner. Applying this aspect to the COURSE EJB would result in a new method, `registerCompressed()`, added to the advised class. Unlike most nonpublic methods, this one would be represented in the class's RMI stub, since it is invoked by code that is included in the stub itself (that code would reside in the advised stub for the `register()` method).

```
public aspect CompressRegistrationList {
   around(Vector v): remotecall(public void register(Vector)) && args(v) {
      CompressedVector cv = new CompressedVector(v);
      registerCompressed(cv);
   }

   private void registerCompressed(CompressedVector cv) {
      Vector v = cv.decompress();
      register(v);
   }
}
```

Fig. 19. An aspect that can be used for sending a compressed version of an argument over the communications line, when applied to the COURSE EJB

Compression and encryption can be applied not only for arguments, but also for return values. In this case, the aspect should use **after**() **returning** advice for both the **remotecall** and **execution** joinpoints. Advice for **after**() **throwing** can be used for processing exceptions (which are often information laden, due to the embedded call stack, and would hence benefit greatly from compression).

7.3 Memoization

Memoization (the practice of caching method results) is another classic use for aspects. When applied to a multitier application, this should be done with care, since in many cases the client tier has no way to know when the cached data becomes stale and should

```
public aspect CacheExchangeRates {
   private static class CacheData { int year; int dayOfYear; float value; }

   private Hashtable cache = new Hashtable();

   pointcut clientSide(String currencyName):
      remotecall(public float getExchangeRate(String)) && args(currencyName);

   around(String currencyName): clientSide(currencyName) {
      Calendar now = Calendar.getInstance();
      int currentYear = now.get(Calendar.YEAR);
      int currentDayOfYear = now.get(Calendar.DAY_OF_YEAR);

      // First, try and find the value in the cache
      CacheData cacheData = (CacheData) cache.get(currencyName);
      if (cacheData != null) && currentYear = cacheData.year &&
         currentDayOfYear == cacheData.dayOfYear)
            return cacheData.value; // Value is valid; no remote invocation

      float result = proceed(currencyName); // Normally obtain the value

      // Cache the value for future reference
      cacheData = new CacheData();   cacheData.year = currentYear;
      cacheData.dayOfYear = currentDayOfYear; cacheData.value = result;
      cache.put(currencyName, cacheData);
   }
}
```

Fig. 20. An aspect for caching results from a currency exchange-rates EJB

be replaced. Still, it is often both possible and practical, and using AspectJ2EE it can be done without changing any part of the client program.

For example, consider a session EJB that reports international currency exchange rates. These rates are changed on a daily basis; for the sake of simplicity, assume that they are changed every midnight. The aspect presented in Fig. 20 can be used to enable client-side caching of rates.

8 Conclusions, Open Questions, and Future Work

Shakeins are a novel, aspect-like, programming construct, with three distinguishing characterization: explicit application semantics, configuration parameters, and restriction on the changes to a class to be re-implementation only. We argued that the shakein construct integrates well with the object model, by distinguishing the five facets of a class: type, forge, implementation, mold and mill, and explaining how these can be modified by shakeins. It was shown that if we adhere to the principle that no variables of shaked classes are allowed, then the construct can be implemented with current JVMs, and using the existing inheritance model of Java.

Shakeins enjoy the advantages of the parameterized notation of mixins, while offering a simple answer to the accidental overriding problem. Thanks to the pointcuts semantics of aspects, shakeins become more expressive than mixins in the sense that they can "examine" the internals of their target classes. Conversely, thanks to the parameterized semantics and the object model integration, shakeins simplify some of the more subtle issues of aspects, including aspect inheritance, instantiation, and abstract aspects.

As an important application and prime motivation for actual use of this construct, we presented AspectJ2EE, an AOP programming language, similar to AspectJ, whose aspects have a shakein semantics. AspectJ2EE shows that the shakeins semantics integrates well with the current architecture of J2EE servers. The language makes it possible to think of existing services as aspects, while unifying the deployment process of J2EE with aspect weaving as in AOP. Also, the language shows that the shakeins semantics allows existing services to be configured, and even applied multiple times. Such benefits are not possible with plain aspects.

By using deploy-time weaving, AspectJ2EE allows the programmer's code to be advised without being tampered with. Programmers can define methods that will provide business functionality while being oblivious to the various services (transaction management, security, etc.) applied to these methods. (It is the *code*, not the programmers, that is oblivious to the nonfunctional concerns—an important distinction, no doubt [35].) With the exception of the **`call`**, we showed that all joinpoint kinds can be implemented using deploy-time weaving.

To the existing repertoire of joinpoints AspectJ2EE adds a new joinpoint kind, **`remotecall`**. This joinpoint makes it possible to add to the familiar services provided by EJB containers, AspectJ2EE aspects can be used to unscatter and untangle tier-cutting concerns, which in many cases can improve an application server's performance. We discussed in particular interesting such services, including client side checking of preconditions, symmetrical data processing, and memoization.

In using the shakeins semantics, aspects in ASPECTJ2EE are less general, and have a more defined target, than their ASPECTJ counterparts. Also, even though the same aspect can be applied (possibly with different parameters) to several EJBs, each such application can only affect its specific EJB target. Therefore, we expect ASPECTJ2EE aspects should be more understandable, and the woven programs more maintainable.

We believe that ASPECTJ2EE opens a new world of possibilities to developers of J2EE applications, allowing them to extend, enhance, and replace the standard services provided by EJB containers with services of their own. EJB services can be distributed and used across several projects; libraries of services can be defined and reused.

ASPECTJ2EE does not encompass the shakeins semantics in full. In particular, aspect application is external to the language and is specified by an XML deployment descriptor file. The file format is such that shakeins can be applied to EJBs only, and that unshaked versions of such a bean are not available to clients.

This restriction demonstrates one of the open problems that this new construct opens, namely, *granularity of applicability*. We argued that explicit application of shakeins to classes contributes to the expressive power that programmers may need. We explained why a global, systemwide application of aspects may lead to undesired results.

Still, as evident by the ASPECTJ2EE experience, it is necessary at times to apply shakeins to a large number of classes. We need a mechanism that supports this need. The answer may lie with a syntax and semantics for applying a certain shakein, or even a family of shakeins, to an entire package, or to a class hierarchy.

Shakein also highlight the issue of using a class as a factory of its instances. It should be possible to change the class implementation, as shakeins do, without modifying client code. However, clients that use the class constructors directly to generate instances, resist such changes.

Although there are design patterns (e.g., ABSTRACT FACTORY and FACTORY METHOD), and framework rules (e.g., as in J2EE) for dealing with this predicament, we believe that a more systematic solution is possible. The appendix overviews *factories*, a mechanism which helps client code access only certain re-implementations of a class. Factories provide similar functionality to ASPECTJ2EE, but are implemented at the langauge level, rather than by using external XML files.

Another challenging topic is that of a type system for shakeins, including a type system for making and enforcing constraints on shakein parameters, and typing of shakein composition. Such a type system should deal with the case that some of the configuration parameters are specified at the time of composition.

References

[1] Cohen T., Gil J. AspectJ2EE = AOP + J2EE: towards an aspect based, programmable and extensible middleware framework. In: Odersky M. (ed.) *Proceedings of the Eighteenth European Conference on Object-Oriented Programming (ECOOP'04)*, pp. 219–243

[2] Cohen T., Gil J. Better construction with factories. To appear in journal of object technology, 2007

[3] Kiczales G., Lamping J., Menhdhekar A., Maeda C., Lopes C.V., Loingtier J.M., Irwin J. Aspect-oriented programming. In: Akşit M., Matsuoka S. (eds.) *Proceedings of the Eleventh European Conference on Object-Oriented Programming (ECOOP'97)*, pp. 220–242

[4] Soares S., Laureano E., Borba P. Implementing distribution and persistence aspects with AspectJ. In: *Proceedings of the Seventeenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, pp. 174–190

[5] Hao R., Boloni L., Jun K., Marinescu D.C. An aspect-oriented approach to distributed object security. In: *Proceedings of the Fourth IEEE Symposium on Computers and Communications*

[6] Kim H., Clarke S. The relevance of AOP to an applications programmer in an EJB environment. In: *Proceedings of the First International Conference on Aspect-Oriented Software Development (AOSD) Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2002

[7] Choi J.P. Aspect-oriented programming with Enterprise JavaBeans. In: *Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC 2000)*, pp. 252–261

[8] Laddad R. *AspectJ in action: practical aspect-oriented programming*. Manning, Greenwich, 2003

[9] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W.G. An overview of AspectJ. In: Knudsen J.L. (ed.) *Proceedings of the Fifteenth European Conference on Object-Oriented Programming (ECOOP'01)*, pp. 327–355

[10] Bracha G., Cook W.R. Mixin-based inheritance. In: Meyrowitz N.K. (ed.) *Proceedings of the Fifth Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming OOPSLA/ECOOP'90*, pp. 303–311

[11] Parnas D.L. Information distribution aspects of design methodology. In: Freiman C.V., Griffith J.E., Rosenfeld J.L. (eds.) *Proceedings of the IFIP Congress*, pp. 339–44

[12] Kiczales G., Mezini M. Aspect-oriented programming and modular reasoning. In: Gruia-Catalin Roman, William G. Griswold B.N. (ed.) *Proceedings of the Twenty Seventh International Conference on Software Engineering (ICSE'05)*, pp. 49–58

[13] Gradecki J.D., Lesiecki N. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, New York, 2003

[14] Schärli N., Ducasse S., Nierstrasz O., Black A.P. Traits: composable units of behavior. In: Cardelli L. (ed.) *Proceedings of the Seventeenth European Conference on Object-Oriented Programming (ECOOP'03)*, pp. 248–274

[15] Cardelli L., Wegner P. On understanding types, data abstractions, and polymorphism. ACM Computing Surveys 17:471–522, 1985

[16] Bracha G. The programming language jigsaw: mixins, modularity and multiple inheritance. PhD thesis, Department of Computer Science, University of Utah, 1992

[17] Allen E., Bannet J., Cartwright R. A first-class approach to genericity. [36] 96–114

[18] Ancona D., Lagorio G., Zucca E. Jam—a smooth extension of Java with mixins. In: Bertino E. (ed.) *Proceedings of the Fourteenth European Conference on Object-Oriented Programming (ECOOP'00)*, pp. 154–178

[19] McDirmid S., Flatt M., Hsieh W.C. Jiazzi: new-age components for old-fashioned Java. In: *Proceedings of the Sixteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pp. 211–222

[20] McDirmid S., Hsieh W.C. Aspect-oriented programming with Jiazzi. [37] 70–79

[21] Gamma E., Helm R., Johnson R.E., Vlissides J.M. *Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing series*. Addison-Wesley, Reading, 1995

[22] Pichler R., Ostermann K., Mezini M. On aspectualizing component models. Software—Practice and Experience 33:957–974, 2003

[23] Bodoff S., Green D., Haase K., Jendrock E., Pawlan M., Stearns B. *The J2EE Tutorial*. Addison-Wesley, Reading, 2002

[24] Duclos F., Estublier J., Morat P. Describing and using non functional aspects in component based applications. In: *Proceedings of the First International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pp. 22–26
[25] Mezini M., Ostermann K. Conquering aspects with Caesar. [37] 90–100
[26] Pawlak R., Seinturier L., Duchien L., Florin G., Legond-Aubry F., Martelli L. JAC: an aspect-based distributed dynamic framework. Softeware—Practice and Experience 34: 1119–1148, 2004
[27] Truyen E., Vanhaute B., Joosen W., Verbaeten P., Jørgensen B.N. Dynamic and selective combination of extensions in component-based applications. In: *Proceedings of the Twenty Third International Conference on Software Engineering (ICSE'01)*, pp. 233–242
[28] Popovici A., Gross T.R., Alonso G. Dynamic weaving for aspect-oriented programming. In: *Proceedings of the First International Conference on Aspect-Oriented Software Development (AOSD'02)*, pp. 141–147
[29] Suvée D., Vanderperren W., Jonckers V. JAsCo: an aspect-oriented approach tailored for component based software development. [37] 21–29
[30] Walker R.J., Baniassad E.L.A., Murphy G.C. An initial assessment of aspect-oriented programming. In: *Proceedings of the Twenty First International Conference on Software Engineering (ICSE'99)*, pp. 120–130
[31] Constantinides C.A., Elrad T., Fayad M.E. Extending the object model to provide explicit support for crosscutting concerns. Software—Practice and Experience 32:703–734, 2002
[32] Gosling J., Joy B., Steele G.L.J., Bracha G. *The Java Language Specification. 3rd edn.* Addison-Wesley, Reading, 2005
[33] Nishizawa M., Chiba S., Tatsubori M. Remote pointcut: a language construct for distributed AOP. In: *Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)*, pp. 7–15
[34] Meyer B. *Object-Oriented Software Construction. 2nd edn.* Prentice-Hall, Englewood Cliffs, 1997
[35] Filman R.E., Friedman D.P. Aspect-oriented programming is quantification and obliviousness. In: *OOPSLA 2000 Workshop on Advanced Separation of Concerns*. ACM, Minneapolis, 2000
[36] Crocker R., Jr. G.L.S. (eds.) In: *Proceedings of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, California, USA, ACM SIGPLAN Notices 38 (11) (2003)
[37] *Proceedings of the Second International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, USA, ACM Press, New York, 2003
[38] Fähndrich M., Leino K.R.M. Declaring and checking non-null types in an object-oriented language. [36] 302–312
[39] Grand M. *Patterns in Java*, vol. 1. Wiley, New York, 1998

Appendix 1 Beyond ASPECTJ2EE: Factories

The ASPECTJ2EE experiment indicates that the shakeins semantics integrates well with current J2EE server architectures. However, the same experiment also led us to the conclusion that program modules should be provided with a better mechanism for controlling object instantiation. We therefore present *factories* as a new language construct, providing each class with complete control over its own instance-creation process.

Factories are methods which return new class instances. Syntactically, a factory is a method which overloads the **`new`** operator with respect to a certain class. The language extension requires no changes to the JVM.

The need for factories stems directly from anomalies that can be found in constructors (in JAVA, C++, etc.). First, constructors are simultaneously static and dynamic: static, since they are invocable without an instance; and dynamic, since they work on a specific object. Second, it is mundane to see that constructors obey a static binding scheme, and it takes just a bit of pondering to understand the difficulties that this scheme brings about. If a class C' inherits from C, then C' should be always substitutable for C. An annoying exception is made by constructor invocation sites in client code; these have to be manually fixed in switching from C to C'.[30]

The confusion between static and dynamic binding penetrates even into the constructor code itself, i.e., into the mill. Method invocation from the mill follows a static binding scheme in C++[31]; in JAVA and C$^{\#}$, however, dynamic binding is used. Neither approach is without fault. Static binding can lead to illegal invocation of pure virtual methods. Dynamic binding prevents methods, invoked from within the mill, from assuming that all fields were properly initialized. This leads, among other things, to difficulties in implementing non-nullable types, as described by Fähndrich and Leino [38]: during construction, fields of non-null types may contain null values.

Unlike constructors, factories can be defined as either static or dynamic.

In studying constructors further, we can identify three steps in instances birth process: (1) *creation*, in which the object's actual type is selected, memory is allocated and structured by the mold; (2) *initialization*, in which fields are set to their initial values; and (3) *setup*, in which the mill is executed.

JAVA (as well as C++, C$^{\#}$ and other languages) does not provide the programmer with control over the creation step.[32]Yet, as the classical creational design patterns (e.g., ABSTRACT FACTORY, FACTORY METHOD, SINGLETON [21], and OBJECT POOL [39]) demonstrate, elaborate software systems require intervention in this step. The use of *home interfaces* in J2EE, or a centralized Factory object in Spring, further underlines this point: since the middleware environment requires control over the object instantiation process, it forces a particular mechanism for obtaining instances, which makes bean client code cumbersome. Creational patterns and home interfaces grant the programmer control over the creation step, by replacing constructor signatures from the forge facet with a different, statically bound, common method (e.g., `getInstance`).

However, in contrast to most other patterns, the creational patterns cannot be implemented in OO languages without revealing implementation details to the client. If class `T` is implemented as a SINGLETON, then clients of this class cannot write **`new`** `T()` and expect the correct instance to be returned; rather, they must be explicitly aware of the nonstandard creation mechanism. Further, to generalize object instantiation, middleware frameworks often place arguments to the creation mechanism in a configuration file. This extra-lingual resource is a source for potential errors that will only be detected

[30] Interestingly, the Gang of Four [21, p. 24] place this predicament first in their list of causes for redesign, saying: "*Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface*".

[31] Even for **`virtual`** methods.

[32] Overloading **`new`** in C++ grants the programmer control over memory allocation, but not over the kind of object to be created, nor the decision if an object has to be created at all.

at run time. Factories enable a clear-cut separation between creation and initialization and setup, and make the use of external configuration files entirely optional.

1.1 Supplier-Side Factories

Class `Demo` in Fig. 21 realizes the SINGLETON design pattern, by overloading **new** with the factory defined in lines 4–8.

This factory is invoked whenever the expression **new** `Demo()` is evaluated, in class `Demo` or any of its clients. Note that the factory is declared **static**, which stresses that it binds statically, and that (unlike constructors) it has no implicit **this** parameter. Examining the factory body we see that it always returns the same instance of the class. Thus, the clients cannot know that `Demo` is a singleton, and will not be affected if this implementation decision is changed.

```
1class Demo {
2  private static Demo instance = null;

4  public static new() {
5    if (instance == null)
6      instance = this();
7    return instance;
8  }

10  Demo() {
11    // ... setup code
12  }
13}
```

Fig. 21. A Singleton defined using a factory

In general, a factory must either return a valid object of the class (instances of subclasses are ok), or throw an exception.[33] A constructor call is required to generate such a new object; but simply writing **new** `Demo()` would recurse infinitely. Instead, the factory invokes the class constructor directly (line 6) with the expression **this**`()`.

Like constructors, factories are not inherited. Had class C' inherited a factory **new**`()` from its superclass C, then the expression **new** C'`()` might yield an instance of C, contrary to common sense. (In contrast, the expression **new** C`()` *can* yield an instance of the C', since a subclass is always substitutable for its superclass.)

Existing techniques for controlling the creation step, such as the `getInstance` method of a SINGLETON, are not protected from inheritance. Therefore, if C is an old-fashioned singleton, then the expression C'`.getInstance()` is valid—but returns an instance of C! This happens because `getInstance` is technically part of the type, while conceptually being part of the forge.

1.2 Client-Side Factories

All examples so far defined factories in the same class on which the overload takes place. Factories of this sort are called *supplier-side factories*. It is also possible to define *client-side factories*, as demonstrated in Fig. 22.

Line 2 in the figure starts the definition of a factory. Unlike the previous examples, this definition specifies the returned type. The semantics is that the definition overloads **new** when used for creating `Account` objects from within class `Bank`. It is invoked in the evaluation of an expression of the form **new** `Account(c)` (where `c` is of type `Client` or any of its subclasses) in this context. This factory chooses (lines 3–7) an appropriate kind of `Account` depending on the particular business rules used by the enclosing class.

[33] Should the returned expression evaluate to **null**, a `NullPointerException` is thrown.

Unlike supplier-side factories, client-side factories *are* inherited by subclasses. Therefore, the factory from Fig. 22 will also be used for evaluating expressions of the form **new** `Account(c)` in subclasses of `Bank`.

```
class Bank {
 public static new Account(Client c) {
  if (c.hasBadHistory())
   return new Secure<LimitedAccount>(c);
   // LimitedAccount is a subclass of Account

  return new Secure<Account>(c);
 }
 // ... rest of the class
}
```

Fig. 22. A shakein-using factory

It is easy to see how middleware applications using shakeins can benefit from factories. In Fig. 22, the client-side factory always applies the `Secure` shakein to the selected type before generating an instance. Thus, any `Account` created by a `Bank` (or a subclass thereof) will be a `Secure`d one. Should the developers desire that a given shakein S must be applied to all instances of some class c, then the supplier-side factory of c can be made to return only instances of $S\langle c\rangle$. The supplier-side factories can choose the particulars of which shakeins to apply based on configuration values set, e.g., in deployment descriptors. This completely alleviates the need for home interfaces for EJBs, while allowing EJB clients to obtain instances as simply as invoking **new**.

Run-Time and Atomic Weaving of Distributed Aspects

Eddy Truyen and Wouter Joosen

Dept. of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, 3001 Leuven, Belgium
{Eddy.Truyen, Wouter.Joosen}@cs.kuleuven.be

Abstract. Run-time weaving of distributed aspects, if performed without any support for atomicity, endangers the global behavioral integrity of the application. Existing aspect-oriented middleware supports run-time weaving of distributed aspects, without addressing this problem. This inherently limits the type of behavioral changes that can be performed at run time. This paper presents a model and an architecture for middleware, named Lasagne, that supports run-time weaving of distributed aspects in an atomic way. The paper makes the case that run-time weaving of distributed aspects is well suited for supporting dynamic and behavioral adaptations that are cross-component, cross-node or cross-layer. Adding support for atomic weaving ensures that such system-wide adaptations are performed in a safe and coordinated way.

Keywords: aspect-oriented middleware, run-time weaving.

1 Introduction

The environments in which distributed software applications must execute have become very dynamic and heterogeneous. As a result, software must be dynamically adapted or even be able to adapt itself. This is for instance especially the case in ubiquitous computing environments and environments in which the availability of resources is variable and unanticipated. The types of adaptations that should be supported are very broad. Many of these adaptations require intrusive modification of multiple classes and components if one has to rely purely on state-of-the-art software development technologies, i.e., object-oriented and component based software engineering.

Aspect-oriented software development (AOSD) tackles this shortcoming by focusing on the systematic identification, modularization, representation and composition of (often nonfunctional) concerns—requirements—throughout the entire software development process. Three core concepts in AOSD are *concerns*, *aspects* and *weaving*. In the context of this paper, it is sufficient to introduce these concepts at the level of aspect-oriented programming (AOP) [24]. Concerns are similar to requirements in a broad sense of the word, ranging from high-level requirements that are articulated in an early stage of the software project,[1] to

[1] For instance, "the application should ensure confidentiality when information is exchanged between two parties".

additional—often detailed—requirements that are generated when performing detailed design and implementation.[2] A concern corresponds in that sense with a feature, capability or quality-of-service (QoS) level that is important for a stakeholder in the software project. At the programming level, an aspect is a modular unit that implements a concern. An aspect definition contains (a) behavior (code that must be executed) which is called *advice* and (b) a specification that expresses when, where and how to invoke the advice; this specification is called a *pointcut*. A pointcut is conceptually defined as a predicate that evaluates over *joinpoints*. A joinpoint is a well-defined place in the structure or execution flow of a program where additional behavior can be attached. Finally, *weaving* is the process of composing core functionality modules (typically application components) with aspects, thereby yielding a working system.

We observe that AOSD—when successfully supported—can simplify adaptations that emerge from changing concerns/requirements. This observation will be clarified in a motivation section (Sect. 2). Based on this view, we consider the dynamic adaptation process of a distributed software application as a continuous process in which aspects are woven to (and unwoven from) the distributed application at run time in order to accommodate evolving requirements.

Many—if not most—nonfunctional concerns in distributed applications are distributed aspects. A distributed aspect is an aspect of which the behavior is woven (into multiple application components that are deployed) on different computer nodes of the distributed system. Aspect-oriented middleware (AOM) (e.g., [15, 39, 42]) supports the creation, deployment and execution of aspects for a distributed environment.

Weaving or unweaving of a distributed aspect at run time implies adding (or removing) mutually dependent functionalities at different nodes of a distributed system. A coordination mechanism is required to preserve the behavioral integrity of the distributed application while the weaving is in progress. In fact run-time (un)weaving must be performed *atomically* from the perspective of the unaffected parts of the distributed application. Atomicity means that the mutually dependent functionalities must be added or removed in an all or nothing fashion.

Existing AOM lacks appropriate support for atomic weaving of distributed aspects. While some systems such as Prose [41] provide support for atomic weaving of aspects within the boundaries of a single computer node, there exists to our knowledge no system that supports atomic weaving of distributed aspects. A naive approach to circumvent this problem is to suspend the execution of the distributed application until the weaving of the aspect code is completed. This approach is nontrivial, however, and incurs a service disruption which is undesirable for mission-critical applications that must permanently be available.

The main contribution of this paper is a model and an architecture for middleware, named Lasagne, that supports atomic weaving without requiring any service disruption. The design of the Lasagne middleware is essentially based on two guiding principles. First, weaving of distributed aspects is performed in

[2] For instance, "decrypted messages should never be cached".

two phases: a *deployment* phase and an *activation* phase. Secondly, we create a solution that preserves the essence of an aspect-oriented execution environment: the atomic weaving mechanism enables a coordinated manipulation of control flow across the boundaries of different computer nodes. We will argue that the application of these two key principles provides the basis for an elegant solution that supports run-time and atomic weaving of distributed aspects.

Previous publications about Lasagne focussed on a generic model for run-time composition of behavioral extensions in component-based applications, but these publications did not integrate that model with aspect-oriented constructs. In [52], the composition model has been presented in the context of a wrapper-based programming model [5, 16], while the focus in [51] was to illustrate the benefits of the composition model in a service-oriented context. This paper integrates the composition model with aspect-oriented constructs, such as pointcut and advice, yielding a *run-time aspect weaving model.* Furthermore it discusses how this run-time weaving model is realized in state-of-the-art AOM, and how the resulting mechanisms can be usefully applied in a more general middleware context. Also, the previous publications addressed system support that allows different concurrent customizations of the application to co-exist, whereas this paper focusses on support for atomicity.

The rest of this paper is structured as follows. In Sect. 2 we articulate the scope and problem statement of this paper. Section 3 describes the run-time weaving model of the Lasagne middleware. Section 4 presents the architecture and design of the Lasagne middleware. We further illustrate the value of our solution in Sect. 5 by discussing the implementation, the applicability, performance trade-offs of Lasagne and integration of Lasagne in standard middleware. We describe related work in Sect. 6 and we conclude in Sect. 7.

2 Scope and Motivation

This section first illustrates the range of adaptations that Lasagne aims to support and motivates why support for run-time aspect weaving will become an important feature of the future middleware. Subsequently the atomic weaving problem is revisited in more detail.

2.1 Scope of Supported Adaptations

Distributed software applications must be dynamically adapted and be able to self-adapt. We give an example to illustrate the versatility of the spectrum of adaptations. Consider an e-finance application that initially supports a limited service of delivering statements of the customer's checking and savings accounts.[3] An initial level of security for this straightforward and minimal application

[3] We will sketch the evolution of this minimal service towards a fully fledged e-banking service. This illustration is based on our experience with real life experience with that type of applications—in that sense, the sketched evolution matches the real history of an application. More info can be found in [27].

could deliver confidentiality by sending the messages over an encrypted network layer—the encryption or decryption facilities will obviously be implemented in the lower layer of the middleware architecture. Security risks will increase when online transactions are supported by an extended set of services. This will be even more the case when private banking services are added, for instance to support the e-trading of securities. The security measures that will be added to the application are no longer embedded in the middleware, in fact specific security measures will be added to the application code, to use advanced authorization for financial transactions, and to add a nonrepudiation feature for specific types of transactions (typically with a higher risk level). Both the financial institution and the clients may wish to have a solid audit trail that effectively enables to prove that certain transactions have been ordered by the client and executed by the bank. As the service becomes more successful and the state of the art of security technologies evolves, the bank may have to adapt the cryptographic building blocks that produce signatures (for instance when certain types of hashing functions become too weak to avoid the algorithms to be broken). Also, authentication tokens do evolve; sometimes different user types may choose for different tokens (smart cards that store X.509 certificates vs. popular generators of one-time passwords, etc.). This picture includes a lot of extensions and alternatives for an evolving security solution. It therefore illustrates that adaptations can be

- *Embedded in the middleware or tightly coupled to application* code (e.g., IP level encryption vs. application level authorization for transactions);
- *Extending* the existing functionality (e.g., adding signature verification through interaction with the certificate revocation list of a public key infrastructure) or *replacing* certain parts (e.g., when updating the signature generation techniques);
- *Context sensitive*: user type and possibly the type of application functionality that is requested may determine the set (or one specific instance) of authentication techniques that are required.

We clearly envision a broad spectrum of adaptations. The common denominator for all these adaptations is that they are all behavioral, i.e., the adaptations require functional extensions and/or replacements, and they all target service-oriented applications. We do not address the management of globally consistent state. Web services, for example, make as many tiers as possible state-less, and incorporate the complete state in a back-end database or in an XML document that is part of the message. It is therefore assumed that a distributed application operates according to a state-less session model, i.e., no external state dependencies towards invoking clients are engaged. Neither do we focus on adaptations that are limited to the modification of configuration parameters (e.g., QoS parameters).

We aim for a generic solution that can cover a broad spectrum of adaptations as introduced above. We believe that the concept of aspects is well suited for modeling entities that are subject to such complex behavioral adaptations. More importantly, the need for a behavioral adaptation occurs not only at load-time but also at run time, hence the need for run-time weaving. For example, for signature verification multiple alternative algorithms exist and each algorithm can be implemented as a separate aspect. Which algorithm is to be applied cannot always be determined in advance. For example, when a drop of computing resources in the operating system layer occurs, the signature verification aspect in the system may have to be replaced with an alternative aspect that reduces the use of computing resources (at the price of some quality levels). Run-time weaving is required as this kind of decisions can clearly only be taken at run time.

In order to give more safety to the run-time weaving process, our work assumes an *aspect–component model* where aspects have to respect the well-defined interfaces of the application components. In this way we avoid undocument coupling between aspects and application components. As such, pointcuts may only evaluate over *noninvasive* joinpoints: joinpoints which are visible in the interface of the components (typically the declared required and provided operations and the invocation/execution of these operations). Furthermore, an aspect–component model typically keeps the behavior of an aspect reusable, by having pointcuts separately specified in an *aspect binding*. Consequently, behavioral adaptations can be represented by aspects that have the properties of components, enabling independent reuse.

2.2 Distributed Aspects and Atomic Weaving

Run-time weaving or unweaving of distributed aspects may endanger the global behavioral integrity of the system. A pedagogical illustration can be based on a simple example of encryption/decryption. Suppose a distributed producer–consumer application, in which a producer object produces data items that are sent over the network to multiple consumer objects that process the data. Now suppose that at a particular point of time, at the node of the producer the need for encrypting the messages is detected due to environmental changes.[4] As such, the advice for encrypting messages must be dynamically woven into the producer object. It is clear that this single, local weaving is not enough because the encryption advice depends on the associated decryption advice that is to be composed at the nodes of the consumers. Encryption and decryption must therefore be woven in an atomic fashion to preserve behavioral integrity. Notice that we have not yet discussed atomic aspect replacement. This is a reconfiguration operation that dynamically replaces a specific aspect with another aspect by unweaving the former and weaving the latter. Obviously, such a run-time replacement must be performed in an atomic manner as well.

Appropriate coordination mechanisms for atomic weaving, unweaving and replacement are nontrivial in a distributed system. For example, in the encryption/decryption example, atomic weaving does not mean that encryption and

[4] This can for instance be triggered by an intrusion detection system—techniques for detecting environmental changes and correlating them into adaptation decisions are obviously out of the scope of this paper.

decryption advices must be injected at exactly the same point of time. At the point of time when the encryption advice is executed for the first time at the producer node, there may still be nonencrypted messages circulating somewhere in the network that still have to be processed by consumer objects. If an attempt to decrypt a nonencrypted message will result in an error, then the decryption advice at the consumer's nodes cannot be woven at the same time as the encryption advice at the producer's node. This atomicity problem becomes even worse when the need and types of encryption changes frequently: this is typically the case when context-sensitive adaptations have to be supported.

3 A Model of Run-Time and Atomic Aspect Weaving

Lasagne is based on a run-time weaving model that is divided into two phases. First, aspects are deployed into the distributed application. Once deployed, aspects can be dynamically activated on a per message basis. We refer to the former mechanism as *deployment* and to the latter as *activation*. The above separation between deployment and activation allows us to apply the "divide and conquer" principle to the complex problem of atomic weaving.

The remainder of this section is structured as follows. A high-level overview of the run-time weaving model is presented in Sect. 3.1. Then, the models of deployment and activation are respectively presented in Sect. 3.2 and Sect. 3.3. Subsequently, Sect. 3.4 summarizes our solution approach to the atomic weaving problem.

3.1 Roles and Scenarios

We give an overview of what are the main human actors involved and the scenarios in which these human actors interact with the Lasagne middleware.

Deployment is performed by a human operator, referred to as *application deployer*, who is trusted within the organization and system infrastructure boundaries of the distributed application. The application deployer uses a deployment tool to deploy, un-deploy or replace aspects, at load-time or run time. How an aspect must be bound with a particular application, is specified by the application deployer in an aspect binding. This aspect binding basically encapsulates one or more pointcuts for the advices of the aspect. Figure 1 gives an example of an aspect binding in the context of the aforementioned producer–consumer application and the encryption–decryption aspect.

In the spirit of context-sensitive adaptation, activation (or deactivation) of an aspect is requested by a *contextual actor*. Contextual actors are software programs or devices, that often are based on interaction with human actors such as end users, system administrators, and domain experts (e.g., security experts who decide on the policies for access control). We describe two simple pedagogical examples of contextual actors and explain how contextual actors are able to activate aspect-based adaptations.

A first example of a contextual actor is a client application or integrated device that uses the services of the aforementioned e-finance application. The

```
aspectspecification encryption-decryption {
  classPath = "....";
  around advice encrypt(byte[] data);
  around advice decrypt(byte[] data);
}

aspectbinding encryption-decryption {
  pointcut messageSend(data): call Consumer.consume(byte[]
    data) within Producer.produce() at distrinet.cs.kuleuven.be;

  pointcut messageReceipt(data): execution Consumer.consume(byte[]
    data) at hostname.cs.kuleuven.be;

  connect {
    messageSend(byte[])<- encrypt(byte[]);
    messageReceipt(byte[]) <- decrypt(byte[]);
  }

  dominates <one or more aspect identifiers>;
}
```

Fig. 1. The specification of the encryption–decryption aspect and its binding to the producer–consumer application is shown. The aspect specification describes the signatures of the advice methods of the aspect and contains the location where the aspect code can be downloaded. The aspect binding encapsulates one or more pointcuts and connects these pointcuts to the advices. For example, pointcut `messageSend()` connects the `encrypt()` advice to a call of the `consume()` operation from within any `Producer` object that is located on host `distrinet`. Optionally, ordering constraints with respect to other aspects can be specified using a **dominates** construct, very similarly to Aspect/J.

client uses activation as a mechanism to customize the service to its individual preferences. Simply by tagging so called *aspect identifiers* to its client requests, the client is able to activate aspects that contribute functional or nonfunctional extensions to the e-finance application that are desired by that client. Another example of a contextual actor, is a resource monitor system component, that monitors the availability of resources in the underlying distributed system infrastructure of the e-finance application. The resource monitor uses activation as a mechanism to adapt the e-finance application to undesired fluctuations in the resource availability or to system overload. For example, during system overload, the resource monitor could deactivate a resource-intensive aspect, and instead activate a cheaper variant of the aspect by tagging all client requests, that arrive at the server-side of the e-finance application, with the aspect identifier of the cheaper aspect.

3.2 Deployment

Figure 2 gives an overview of the deployment model of Lasagne by showing the sequence of steps that are performed to deploy a particular aspect. First, the

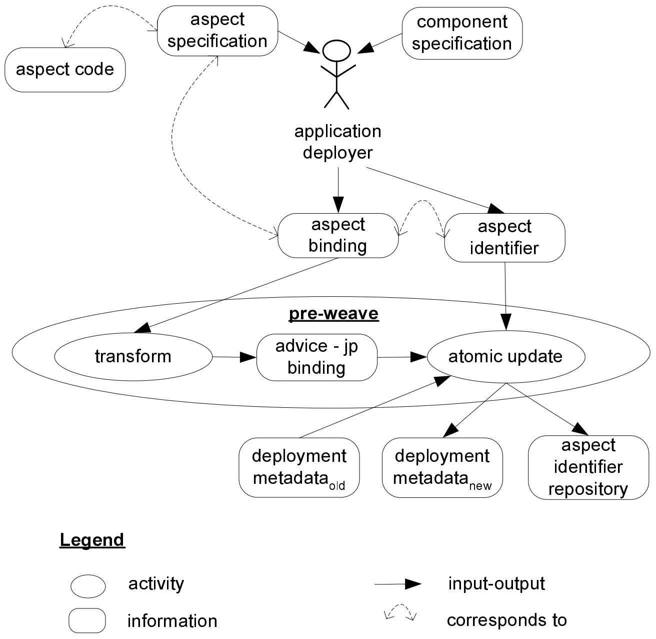


Fig. 2. Deployment data flow. The application deployer gives to the deployment tool an aspect identifier and an aspect binding. The deployment tool first transforms the aspect binding into lower-level advice-to-joinpoints bindings and then atomically updates the deployment metadata with these bindings and the corresponding aspect identifier. The aspect identifier is stored in a publicly accessible aspect identifier repository.

application deployer must of course specify an aspect binding. The application deployer uses subsequently the Deployment API of the middleware to deploy the aspect. For example, the encryption–decryption binding of Fig. 1 is deployed as follows:

```
lasagne.core.Deployment.deploy(
    "encryption-decryption",
    new AspectIdentifier("MessageConfidentiality"));
```

The application deployer must assign a unique name to the aspect. This name, referred to as *aspect identifier*, is supposed to associate with the aspect a meaning that makes sense to contextual actors. The aspect identifier may for instance describe a capability, feature or QoS level that is of interest to contextual actors. In the above example, the application deployer maps the encryption–decryption aspect to the aspect identifier “MessageConfidentiality” to indicate that this aspect offers a specific security property when transmitting messages.

Subsequently, the Lasagne middleware interprets the aspect binding to effectuate a so-called *preweaving* of the aspect with the application. This means that the aspect is not woven directly into the application code, but it is woven into a metalevel representation of the application code, which is called *deployment* *metadata*. Deployment metadata describes a lower-level configuration of the aspect bindings in terms of which advices of aspects must be woven at which joinpoints of application components. As shown in Fig. 2, the process of preweaving is a consecutive process of transforming the aspect binding into a sequence of so called *advice-to-joinpoint bindings*, and atomically updating the deployment metadata with these bindings. An advice-to-joinpoint binding connects a specific advice to a specific joinpoint. More specifically, it is a tuple of four members: the signature of the advice method, the aspect identifier associated with that advice method, the joinpoint where the advice must be executed and the order of execution (relative to other advices at that joinpoint).

Finally, the aspect identifier is stored in a public *aspect identifier repository* for future reference on behalf of contextual actors: they need it in order to activate the aspect.

3.3 Activation

Deployment of an aspect does not involve "turning the switch" that effectively injects the aspect code into execution space. Instead, turning the switch is under control of flow-dependent activation that allows to initiate the execution of aspects by tagging aspect identifiers to messages.

To support multiple concurrent adaptations [49, Chap. 2] by contextual actors, activation of an aspect will only affect the current execution context (i.e., the current message flow or invocation path in which the contextual actor is involved) and will not interfere with other ongoing flows of messages. It is therefore also called *flow-dependent* activation.

Figure 3 gives an overview of the activation model. To request the activation of an aspect at a certain execution point, a contextual actor intercepts the application-level message that corresponds with this execution point and tags it with the desired aspect identifier. Of course, the contextual actor can tag multiple aspect identifiers to a message. Multiple aspects are thus combined and activated for the same message. Once the aspect identifiers are tagged with the message, they *propagate* with the invocation path of messages, succeeding the original message. As such, the aspect activation logic travels (as a set of aspect identifiers) with the message flow. The run-time weaver of the Lasagne middleware consults the tagged aspect identifiers in order to know which advices to execute for a given message. When an aspect is activated for the first time, and therefore, the code of the aspect is not yet woven into the application code, the run-time weaver has to consult the deployment metadata in order to know which advice must be woven at which joinpoints.

Figure 4 presents a message sequence diagram that illustrates the flow-dependent activation of the encryption–decryption aspect in the producer–consumer example. Here a client application, using the service of the producer–consumer application, plays the role of contextual actor. Message exchange between the client application and the frontend of the producer–consumer application is based on a well-defined *collaboration* (i.e., invocation of an object-based

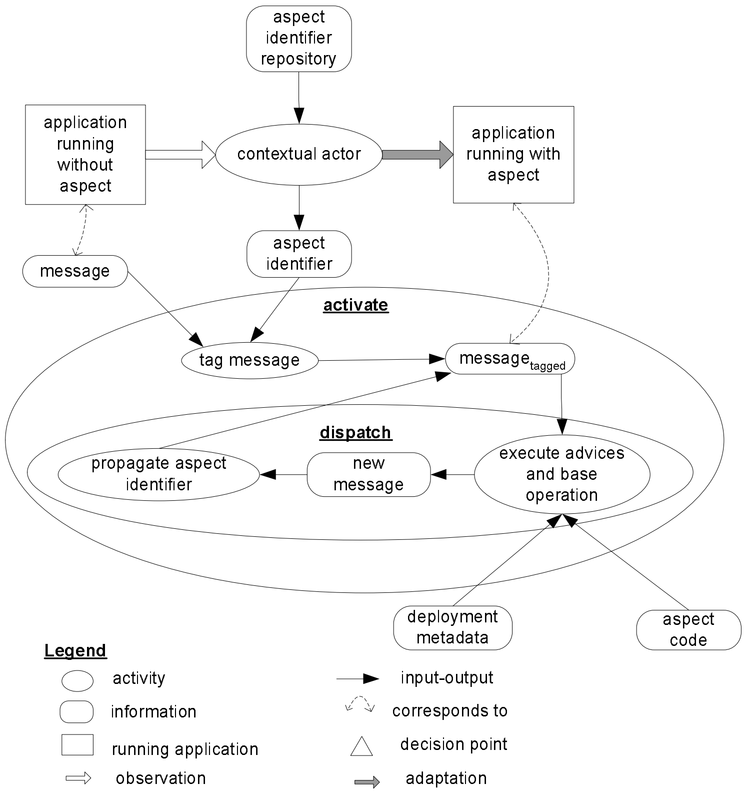


Fig. 3. Activation data flow. The contextual actor observes directly or indirectly the behavior of the application and decides at a certain point in time to adapt the application by activating one or more aspects. To activate an aspect, the contextual actor dynamically tags aspect identifiers to messages which are on their way to the application. When processing a message, the run-time weaver consults the tagged aspect identifiers in order to know which advices to execute. If some aspect is not yet injected into the application, the run-time weaver first loads the aspect's code, and updates its internal data structures, based on the deployment metadata. New messages, resulting from the original message's execution, are tagged with the same set of aspect identifiers by the run-time weaver. As such, aspect identifiers propagate.

interface).[5] Client application and frontend typically perform multiple *collaboration instances* sequentially. For example in Fig. 4 one collaboration instance has completed its execution, and a second one is running till completion. The figure illustrates that once the client application has tagged the aspect identifier "MessageConfidentiality" to the message that starts the second collaboration instance, the aspect identifier is propagated with the subsequent message flow

[5] Packet-based communication using a transport protocol is another example of a collaboration that is situated at the level of the lowest middleware layer.

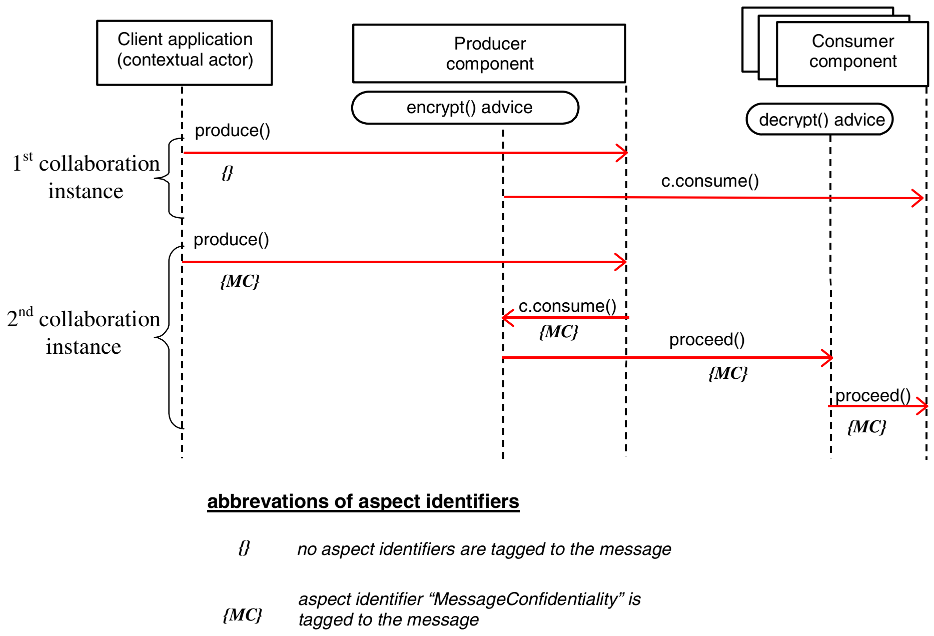

Client application (contextual actor)
Producer component
Consumer component
encrypt() advice
decrypt() advice
1st collaboration instance
2nd collaboration instance
produce()
{}
c.consume()
produce()
{MC}
c.consume()
{MC}
proceed()
{MC}
proceed()
{MC}
abbrevations of aspect identifiers
{} no aspect identifiers are tagged to the message
{MC} aspect identifier "MessageConfidentiality" is tagged to the message

Fig. 4. Flow-dependent activation of aspects

across the producer–consumer application such that the run-time weaver knows it has to invoke the advices of the corresponding aspect along this message flow.

To manage flow-dependent activations in an effective way, it is useful to group them into *sessions*. A session is defined as the group of messages that are exchanged between a client and a service as part of one or more collaboration instances. The *activation policy* of a session, or *session policy* in short, is defined as the set of aspect identifiers that is activated for that particular session. In this view a flow-dependent activation can be considered as an update of a particular session policy. Typically, the session initiator defines a particular activation policy and this policy remains the same during the whole session. Concurrently running sessions can have different activation policies. Finally, it is possible to dynamically update the activation policy of an ongoing session provided the application architecture conforms to a well-defined structural property. This property will be explained in more detail in Sect. 4.1.

3.4 Summary of Atomic Weaving

The division of the weaving process into a deployment and an activation phase enables us to decompose the problem of atomic weaving into three subproblems that are easier to solve on their own.

Complete and correct deployment. Lasagne ensures that an update to the deployment metadata due the preweaving of an aspect is correctly and completely distributed across all computer nodes on top of which the distributed application executes.

Consistency in the activation of aspects. During flow-dependent activation, Lasagne guarantees the consistency in the activation of aspects: once a combination of aspects has been activated for a collaboration instance by tagging aspect identifiers to the message that starts that collaboration instance, this decision is consistently coordinated all over the subsequent execution of the collaboration instance by means of propagating the aspect identifiers with the message flow.

Coordination between deployment and activation. An important correlation between deployment and activation is that Lasagne only permits the activation of an aspect after the aspect is completely and correctly deployed. Thus, a successful completion of the deployment protocol is a prerequisite for initiating the activation of the aspect. The underlying rationale is of course that when the binding of an aspect is incompletely represented in deployment metadata, the execution of the aspect will also be incomplete, and thus results into inconsistencies in the systemwide behavior of the application. This phased transition from deployment and activation is realized by means of a coordination protocol.

In the next section we describe the architecture and design of the Lasagne middleware with a focus on how to achieve the above three requirements. Before proceeding, we would like to point out that the division into deployment and activation also helps to improve other important properties including security, availability, scalability, etc. With respect to security, the division implies a division of authority: only the trusted application deployer is allowed to deploy new aspects. This is a first line of defense against malicious contextual actors [51]. Furthermore, as already argued, without a separate deployment phase, one would have to tolerate a temporary service disruption. Also, a separate deployment phase makes the weaving process scalable towards complex collaborations and federated applications. Suppose one wants to atomically weave multiple mutually depending subaspects into a large-scale federated application, where each subaspect is to be woven in a specific subsystem of the application. To do this, one first deploys the subaspects with the *same* aspect identifier, then one activates the whole configuration of subaspects using that unique aspect identifier. A more detailed treatment of these issues can be found in [49, Chaps. 2, 5].

4 Architecture of the Lasagne Middleware

In this section we present the architecture and design of the Lasagne middleware. First we give an overview of the distributed systems architecture of Lasagne. Then we elaborate upon the approach taken to achieve atomic weaving. Finally we focus on the design of Lasagne's run-time weaver which is the core of the Lasagne middleware.

4.1 Distributed System Architecture

The Lasagne middleware is compatible with a broad range of application architectures and platform architectures. There are only a few assumptions. With respect to the application architecture, it is assumed that the components of the distributed application are grouped in several operating system processes that are possibly allocated on different computing nodes of the distributed system. With respect to the platform, we assume that intra-process as well as inter-process communication between components is based on messages.

Figure 5 gives an overview of the distributed system architecture of the Lasagne middleware. It is conceptually divided into a *deployment system* and a *run-time system*. The deployment system implements the **preweave** activity (see Fig. 2), while the *run-time system* implements the **activate** activity (see Fig. 3). The deployment metadata, which is stored in a *deployment metadata repository*, is shared info between the deployment system and the run-time system. The deployment metadata repository may be centralized or replicated, depending on how local one wants to keep the deployment metadata to the run-time system instances, and depending on the size and topology of the distributed system.

The architecture of the deployment system consists of a *deployment tool* and a *data tier*. The deployment tool offers the functionality for transforming an aspect binding into advice-to-joinpoint bindings and for atomically updating the deployment metadata repository with these bindings. The data tier conceptually consists of three kinds of data stores. These are the *aspect identifier repository* (introduced in Sect. 3.2), a *code repository* that stores all the binaries of aspect and component code, and a *specification repository* that acts as a backend for the deployment tool by storing aspect bindings, and aspect and component specifications. Similar to the deployment metadata repository, several alternative architectures are possible for the data tier.

The architecture of the run-time system consists of a *run-time weaver* and an *activator*. A separate instance of the run-time weaver is running within each process.[6] Its main responsibility is to support the **dispatch** activity, as sketched in Fig. 3. The activator component is a plug-in for contextual actors that provides support for the **tag message** activity in Fig. 3.

Messages in Lasagne have an associated *invocation context*, in which properties about the ongoing computation can be stored. This invocation context is also used to tag aspect identifiers to messages. The activator component provides an interface to contextual actors for setting aspect identifiers to this invocation context field. Propagation of the aspect identifiers with the message flow is simply achieved by copying the invocation context from one message to another. The aspect identifiers are used by the run-time weaver to dynamically dispatch messages to a selected subset of advices. How this is implemented will be further clarified in Sect. 4.3.

[6] For example, in Java-based applications [3], there would be a single instance of the run-time system per Java virtual machine.

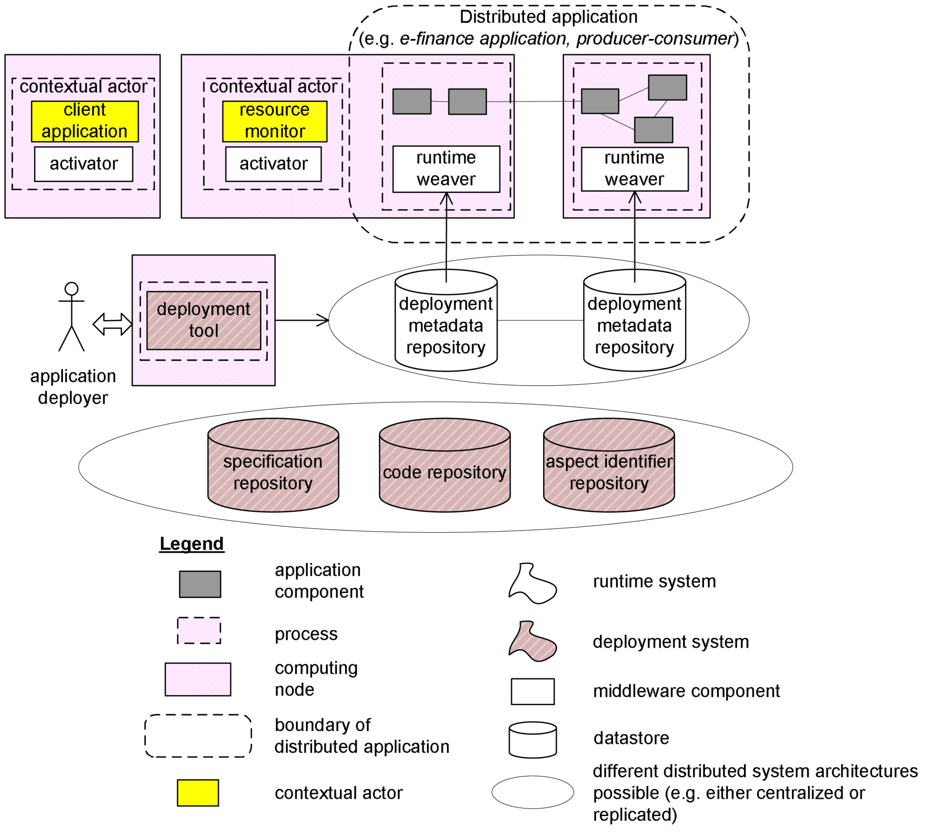


Fig. 5. Distributed system architecture of the Lasagne middleware

Finally, we relate the application architecture to the notion of session policy. To ensure consistency in the activation of aspects, the activation policy of an ongoing session may only be dynamically updated at the start of a new collaboration instance. As will be explained in Sect. 4.2, the Lasagne middleware has the capacity to enforce this constraint for application architectures where an exterior client process (e.g., the client of the e-finance application) or any intermediate proxies (e.g., the resource monitor) require dynamic adaptations of the interior server processes (the e-finance application itself). On the other side, for application architectures where the boundary between the exterior and interior components cannot be drawn clearly,[7] the activation policy of a session must remain the same for the lifetime of the session. Notice that in these applications, it still is possible for concurrent sessions to have different activation policies.

[7] One can argue that such applications have a weak architecture, but this is out of the scope of our discussion.

4.2 Support for Atomic Weaving

This section presents the detailed approach of how the Lasagne middleware supports atomic weaving. As stated in Sect. 3.4, we have divided this complex problem into three subproblems. We describe the solution for each of these problems.

Correct and complete deployment. In order to enable correct and complete deployment of an aspect, atomicity and concurrency control must be provided at the level of updates to deployment metadata. The use of a conventional transaction protocol is the best way to implement this. Hence, we refer to the process of updating the deployment metadata of a distributed application as a *deployment transaction.* How to implement the deployment transaction depends on whether the deployment metadata repository is replicated or not.

A centralized repository is simply implemented as a transactional file service or database that employs a two-phase locking protocol to serialize concurrent updates.

In a decentralized approach, deployment metadata is replicated at multiple nodes of the distributed application. Here, an update of deployment metadata must be coordinated by means of a two-phase commit protocol. Concurrency control within each replica is again provided by implementing it as a transactional file service. The advantage of the decentralized approach is that the Lasagne run-time weaver can consult the deployment metadata locally.

Finally, a hierarchical approach is useful to make deployment scalable towards complex collaborations in federated applications. This hierarchical approach implies a delegation of deployment authority: different subsystems of a federated application are controlled by separate application deployers who are themselves organized in a hierarchy. Correct and complete deployment of mutually depending aspects into these different subsystems is then coordinated at the root of this hierarchy by means of a nested two-phase commit protocol.

Consistency in the activation of aspects. As explained in Sect. 3.3, the core mechanism for ensuring consistency is the propagation of aspect identifiers with the message flow. As a result, within a distributed application, the run-time weaver instances in different processes are implicitly coordinated by the tagged aspect identifiers. In other words, when two run-time weaver instances are requested to dispatch messages which logically belong to the same message flow, then these messages will be consistently dispatched to the same combination of aspects.

A second element related to activation consistency implies that flow-dependent activations are only permitted for messages that start a new collaboration instance. Since we assume a state-less session model and an application architecture where the boundary between the exterior and interior components can be drawn clearly, this means that flow-dependent activations are only permitted for messages that enter the application architecture.

This constraint can be enforced in two ways: by rigorous design of the application architecture, or by run-time checks at the level of the middleware. At the design level of the application architecture, one can enforce the constraint by placing contextual actors at the right places of the application architecture. For example, every application that provides a service to client applications has some process that acts as a frontend. It is clear that tagging of aspect identifiers may only occur on messages that are exchanged between the client applications and this frontend (either at the client-side or at the server-side).

The second approach is to ensure that the underlying middleware infrastructure can automatically detect if messages enter the application architecture. We introduce the concept of an *application identifier* which is a specialization of the concept of aspect identifier. An application identifier is defined as a name that identifies the components that are part of the application architecture. For example, the components of the e-finance application could all be tagged with the application identifier"E-Finance", whereas the client application could be tagged with the application identifier "Client". Thus the concept of a message entering the application architecture is modeled as a message of which the sender and receiver component are associated to different application identifiers.[8]

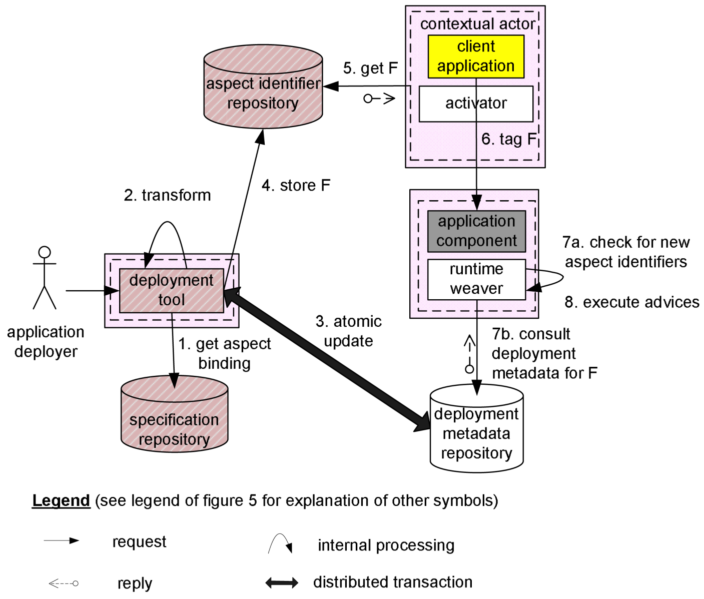
Coordination between deployment and activation. We have separated weaving into a **preweave** and an **activate** activity to cope with the complexity of atomic weaving. We now bring these two activities into synchrony to effectively gain the desired atomicity property. To achieve this synchrony, a coordination protocol must be defined between the different system components of the Lasagne middleware.

Three different coordination protocols are actually required, respectively for weaving, unweaving and replacement of aspects at run time. We now discuss the coordination protocol for aspect weaving by means of a concrete adaptation scenario.

Suppose, a specific client application, which is already using the services of the distributed application, desires a feature that is not yet deployed in the application. To accommodate this request, the application deployer should deploy at run time a new aspect that implements the feature as required by the client.[9] The application deployer requests the deployment tool to deploy the aspect and hereby he associates a new aspect identifier, say F to the aspect. After that, the coordination protocol for governing the actual run-time weaving of the aspect begins. Figure 6 indicates, by means of numbered arrows, what are the steps involved in this coordination protocol.

[8] In practice, application identifiers are marshaled into the sender and receiver fields of the wired messages such that the activator component can verify this property.

[9] We assume there is an off-line meeting about the requirements of the feature between the stakeholders of the client application and the distributed application. After that, the application deployer seeks to realize the feature by means of a suitable aspect implementation and specifies the binding of the aspect with the distributed application by means of an aspect binding.



Scenario: the application deployer requests the deployment tool to deploy a specific aspect with aspect identifier F and, after that, a client application wants to activate this aspect.

Fig. 6. Coordination protocol for aspect weaving at run time

The coordination protocol consists of a deployment phase and an activation phase. The deployment phase consists of four steps:

(1) The deployment tool retrieves the aspect binding of the aspect from the specification repository.
(2) The deployment tool transforms the aspect binding into advice-to-joinpoint bindings.
(3) The deployment tool updates the deployment metadata repository with the advice-to-jointpoint bindings in a deployment transaction.
(4) The deployment tool inserts the aspect identifier F into the aspect identifier repository.

The activation phase consists of four steps:

(5) The client application retrieves the aspect identifier F from the aspect identifier repository.
(6) The client application tags aspect identifier F to its subsequent client requests.
(7) The run-time weaver notices the new aspect identifier F on the first arrived message and consults the deployment metadata.
(8) The run-time weaver updates its internal data structures to manage the new aspect and executes the appropriate advices, related to F.

Obviously, the coordination protocol is very simple. This is the result of the fact that deployment and activation are quite orthogonal operations, and that the Lasagne run-time system already incorporates dedicated support for maintaining consistency in the activation of aspects.

The details of the coordination protocol for aspect unweaving and replacement can be respectively found in Appendices 1 and 2.

4.3 Design of the Run-Time Weaver

The run-time weaver is designed as a message-based dispatcher. What is required is (a) the aspect behavior needs to be supported in the dispatch (method) tables of all application objects involved and (b) the aspect behavior needs to be executed whenever the message (that needs to be processed) expresses the need for that particular behavior. This is nontrivial compared to a centralized application where this behavior is localized in one context: then inconsistency cannot occur. In a distributed application, however, the distributed context needs to be updated without the risk of inconsistencies to occur.

We do need a mechanism as lightweight as possible. Different from most AO middleware systems, we want to support per-instance aspect-composition: this means that either dispatch tables will be created per object (which is very expensive) or dispatch tables will be shared by objects in a single process. This is a more economical approach. The behavior in such a table then is a superset of what can be executed in a specific collaboration. In this implementation context, deployment means binding specific advice behavior to distributed objects, activation means selectively executing that advice behavior. Thus, per instance run-time aspect weaving in a distributed context will be performed by inserting the required advices into the dispatch table and by executing (activating) that behavior on a per-message basis.

The internal structure of the run-time weaver is exactly decomposed according to this division: as shown in Fig. 7 it consists of a *deployment manager* component that manages the behavior (and state) of the aspects, and an *advice combiner* that selectively executes that behavior. As indicated above, the deployment manager maintains a dispatch table that has an entry for each provided and required operation of the application components. Each entry stores an ordered list of pointers to advice methods. The deployment manager also creates and manages aspect instances. Since a full treatment of Lasagne's aspect state management is out of the scope of this paper, it suffices to say that Lasagne supports a per-object aspect instantiation scope. More details about Lasagne's instance management can be found in [49, Chap. 5].

Figure 7 illustrates in more detail how the deployment manager and the advice combiner interact among each other to perform the actual weaving. An activator, shown at the top left corner of the figure, tags one or more aspect identifiers to a particular client request. The message flow, that represents the processing of this client request, propagates through the application. The advice combiner interprets the tagged aspect identifiers of each message and will only

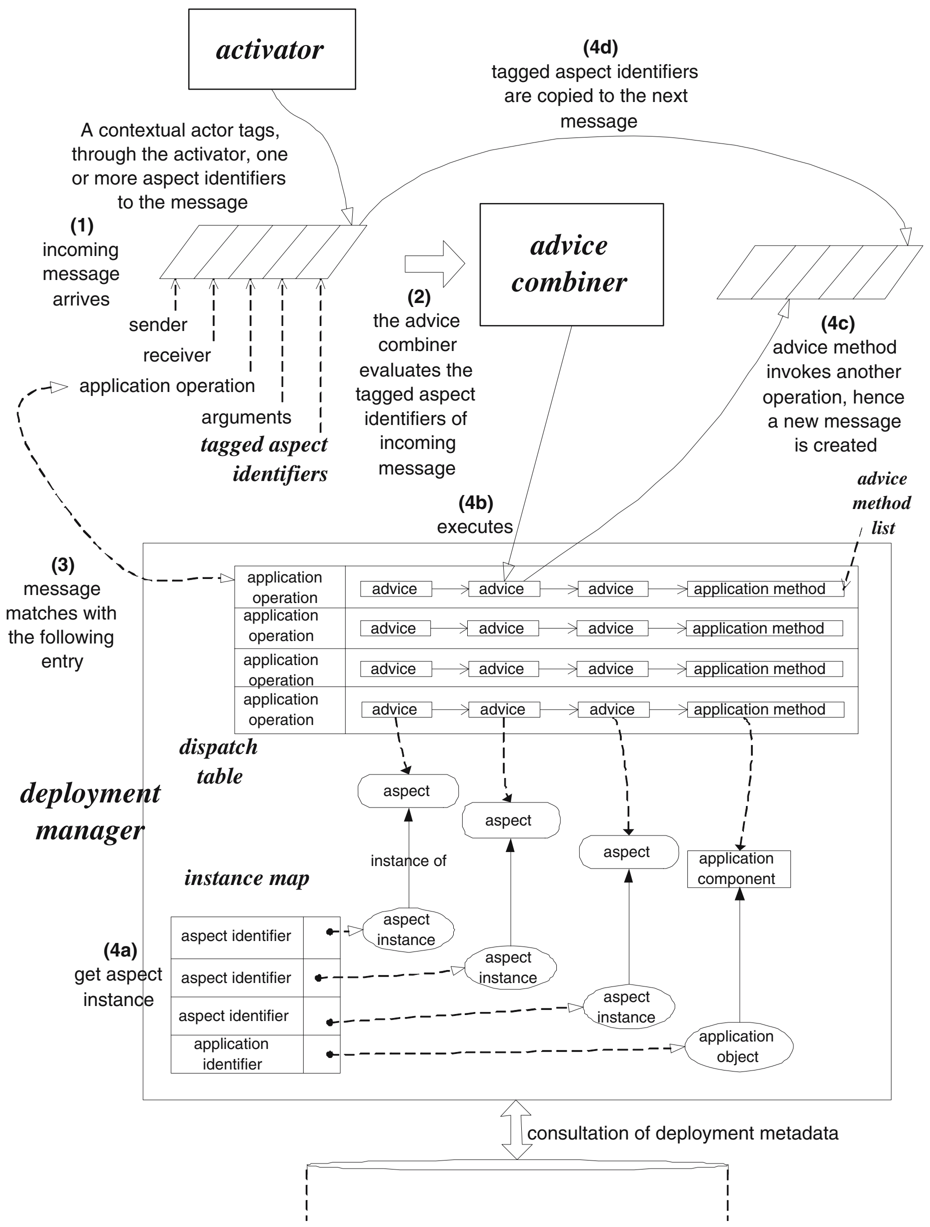

activator
A contextual actor tags, through the activator, one or more aspect identifiers to the message
(4d) tagged aspect identifiers are copied to the next message
(1) incoming message arrives
sender
receiver
application operation
arguments
tagged aspect identifiers
advice combiner
(2) the advice combiner evaluates the tagged aspect identifiers of incoming message
(4c) advice method invokes another operation, hence a new message is created
advice method list
(4b) executes
(3) message matches with the following entry
application operation
advice
application method
dispatch table
deployment manager
aspect
application component
instance of
instance map
aspect identifier
application identifier
aspect instance
application object
(4a) get aspect instance
consultation of deployment metadata

Fig. 7. The design of the run-time weaver: interplay between the advice combiner and the deployment manager

execute the advices associated to these aspect identifiers. For every message, the advice combiner thus selectively combines advices from the superset in the dispatch table.

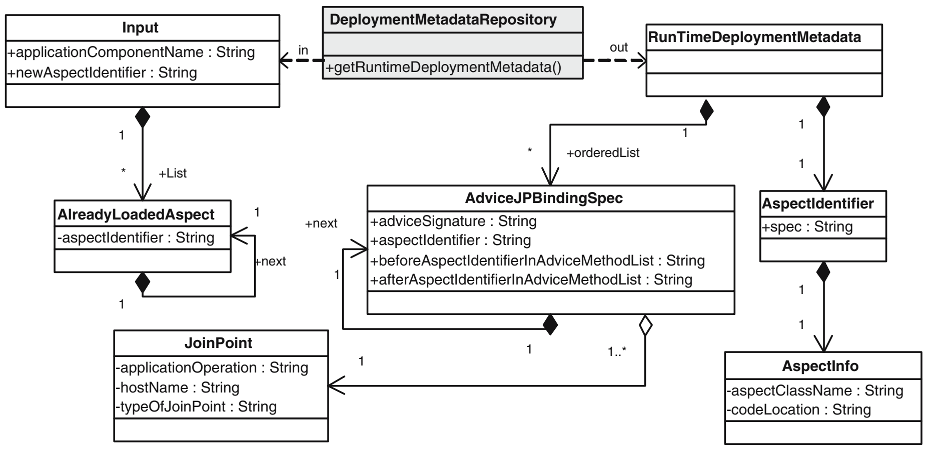

DeploymentMetadataRepository
+getRuntimeDeploymentMetadata()
in
out
Input
+applicationComponentName : String
+newAspectIdentifier : String
RunTimeDeploymentMetadata
1
*
+List
+orderedList
AlreadyLoadedAspect
-aspectIdentifier : String
+next
AdviceJPBindingSpec
+adviceSignature : String
+aspectIdentifier : String
+beforeAspectIdentifierInAdviceMethodList : String
+afterAspectIdentifierInAdviceMethodList : String
AspectIdentifier
+spec : String
JoinPoint
-applicationOperation : String
-hostName : String
-typeOfJoinPoint : String
1..*
AspectInfo
-aspectClassName : String
-codeLocation : String

Fig. 8. Deployment metadata repository interface for consulting deployment metadata at run time

Possibly, the behavior related to a requested aspect identifier is not yet supported in the dispatch table. In this case, the deployment manager component first consults the deployment metadata repository for this aspect identifier. The interface of the deployment metadata repository that supports this consultation operation is shown in Fig. 8. The returned deployment metadata is essentially a list of advice-to-joinpoint bindings and the aspect's code base location. Based on this information, the deployment manager is able to load the aspect code and insert advice method pointers into the dispatch table.

Of course, statically deployed aspects[10] will be processed by the deployment manager component at load-time, using another interface of the deployment metadata repository (see Appendix 3).

5 Discussion

The discussion of this work will focus on four topics: the techniques used for implementing Lasagne, the integration of Lasagne on top of standard middleware platforms, the performance trade-offs in the current design of Lasagne and the applicability of Lasagne.

5.1 Implementation

To validate our run-time weaving model we have implemented a prototype of the Lasagne middleware on top of the programming language Correlate [43].

[10] In other words, the application deployer also has the option to deploy aspects before the application is started up.

Correlate is a concurrent object-oriented language with an execution environment that makes it easier to develop distributed applications. The execution environment is designed as a metalevel architecture that offers a so-called *metaobject-protocol* (MOP) [23]. Correlate's MOP essentially offers several metaobjects that allow to inspect and modify the internal semantics of the language and execution environment. Correlate is implemented as an extension of the Java programming language [3].

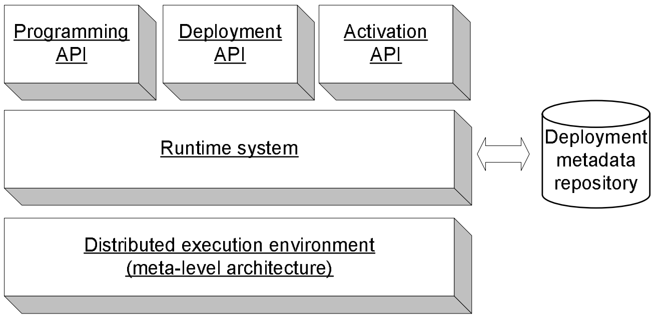

Programming API
Deployment API
Activation API
Runtime system
Deployment metadata repository
Distributed execution environment (meta-level architecture)

Fig. 9. Overview of the Lasagne middleware

Figure 9 illustrates the implementation structure of the Lasagne middleware. As shown in the figure it consists of three layers: the API layer that offers three separate programming interfaces for programming, deploying and activating aspects; the run-time system (the core of the Lasagne middleware); and a distributed execution environment that comes with Correlate.

The run-time system of Lasagne is implemented as a modular plug-in of Correlate, using Correlate's MOP. The design of the run-time weaver has already been presented in Sect. 4. The activator component is implemented using Correlate's MOP as well, by relying on the particular metaobject that allows to intercept method invocations and executions as first-class message objects. The interface of the Activator component corresponds to the Activation API. The deployment metadata repository is implemented as a set of files that are managed in a distributed file system.[11] Each file stores the deployment metadata of a given application component.

The Programming and Deployment API provide basic support for programming and deploying aspects. We do not require any extensions to state-of-the-art pointcut constructs. Run-time deployment is currently supported by manually updating the deployment metadata files. As outlined in Sect. 4.2, we only need to rely on basic transaction protocols to ensure atomicity and concurrency control when updating the deployment metadata files.

[11] UNIX Sun OS 5.8 implementation of the Sun network file system [44].

5.2 Integration in State-of-the-Art Platforms

We compare Lasagne middleware to standards middleware such as OMG's CORBA, Enterprise Java Beans (EJB), and Microsoft .NET, and discuss how Lasagne can be integrated in these platforms.

The Lasagne run-time weaver is situated at the same level as the Portable Object Adapter(POA) component in the OMG CORBA ORB model [53], or at the same level as the Container architecture in the Enterprise Java Beans (EJB) specification [34] and the OMG CORBA Component model [54]. An important difference is however, that in traditional object-based middleware, message dispatch (or similar mechanism in container-based middleware) is only driven by the receiver's object identity that is stored in the message. In Lasagne, message dispatch is (in addition) driven by aspect identifiers.

The tagging of aspect identifiers can be commonly achieved in CORBA by means of portable interceptors [37] or in .NET by means of a special API that gives access to the logical call context [28].

In Lasagne, aspect identifiers are propagated with the message flow by storing the identifiers as part of the reified messages, and copying them along the invocation path. Another approach is storing the aspect identifiers in a thread local variable. We have not followed this approach in order to circumvent the problem of losing *logical thread identity* when control crosses system boundaries, or when control is asynchronously handed over between two threads as in the active object model [36]. Some of the standard middleware platforms already provide a mechanism to cope with this problem such as distributed threads [10] in real-time CORBA or the LogicalCallContext class in .NET [28].

Finally, the aspect identifier repository can be compared with a standard component repository, with the difference that it enables lookup of (distributed) aspects, whereas a standard component service enables lookup of (distributed) components.

5.3 On Performance Trade-Offs

The proposed model and architecture has clearly been focused on providing generic support for a broad range of behavioral adaptations that have to be performed in a distributed system. Notice that we aimed for a solution that enables different context-dependent versions of the application to coexist, and that should enable distributed service offering without any disruption because of adaptations. It therefore seems obvious that we have created an architecture which generates quite some overhead in achieving the proposed objectives. We analyze the costs by addressing (1) cost in terms of service availability, (2) overhead in terms of the cost of processing interactions, (3) overhead in terms of memory consumption.

(1) Our solution offers consistent behavioral adaptation of distributed entities without causing unavailability of any service or application feature.

(2) We have chosen to obtain the value of our solution at the cost of an increased processing time for invocations. The reason is threefold:

(1) We want a solution that guarantees no service disruption when an (complex) adaptation is in progress. This is an inherently solid basis for mission-critical systems that must be permanently on-line.
(2) We anticipate that a broad range of network applications (e.g., the typical case of Web services) is relatively insensitive to processing delays as communication delays often dominate the response times in executing distributed services.
(3) From a methodological viewpoint, we believe that it is a sound strategy to ensure that (a) a first (base level) architecture is offering a robust solution with absolute guarantees (no service disruption and atomicity) and that (b) optimizations—probably weakening other qualities (e.g., flexibility)—are pursued thereafter. This is exactly the scope of our ongoing and future work. For instance, Lasagne has already an option to limit deployment of aspects to load-time only. Flexibility cuts are minor (aspects can still be dynamically activated and deactivated on a per message basis), while the performance overhead due to the coordination between deployment and activation is eliminated.

(3) Our architecture is extremely economical when it comes to memory cost. Deploying distributed aspects implies the storage of a logical reference (aspect identifier) in the nodes where a specific aspect must be executed. Pro-actively loading aspects would lead to substantially larger memory footprints, as well as extra delays. In the current solution, aspect code is loaded on demand, i.e., when the first activation of the required behavior occurs.

5.4 Applicability

For several application areas, it has become increasingly more important that software can be dynamically adapted to evolving requirements of a given context. Our discussion will focus on three such application areas: application service provisioning, autonomic computing, and adaptive QoS. The proliferation of Web services, and recent initiatives such as Microsoft's *Office Live* [11] indicate that software applications are gradually evolving from a desktop model to a *service provider* model. For example, Microsoft Office Live will offer expert business management applications as online services without that their customers need Office or Windows installed on their desktop computers. Moreover, the underlying system infrastructure is not only targeted at remote application execution, but also at the uniform deployment and integration of new applications. In other words, the vision [48], that system software such as compilers will some day in the future be offered as distributed service objects, seems to emerge. This evolution creates a completely new meaning for the notion of a client request. When popular application services begin to have a large number of client requests on a daily basis, it becomes economically profitable for the service provider to make such application services more dynamic and customizable to the needs of individual customers. Existing online (Web) services are already personalized. Such personalization is however limited to the user interface and data that is offered to the customer. An additional opportunity is the supporting of customization scenarios in which customers can dynamically adapt the behavior of the application service itself. Needless to say that Lasagne is very well suited for providing this type of client-specific customizations.

Lasagne can also provide a complementary advantage to middleware support for autonomic computing and self-healing [7, 17], and adaptive QoS [4]. Research in these areas is targeted towards extending middleware with policies and techniques for context monitoring and dynamic selection of adaptation strategies. To our knowledge, these middleware extensions do not automate the task of coordinating adaptation operations across multiple processes and computer nodes; the coordination must be explicitly managed by the programmer. After all, these middleware extensions are integrated with an object-based platform, thereby constraining the representation of a single adaptation strategy as inherently localized. We believe that Lasagne will be useful for automating cross-node coordination requirements in QoS control and autonomic computing systems as Lasagne offers a solid service-enabling platform for the management and coordinated selection of cross-process and cross-node adaptation strategies.

6 Related Work

Five categories of related work are considered: recent AOM advances, network software, dynamic software architecture, coordinated adaptation in distributed systems and reflective middleware.

AOM. The general relation to other AOM platforms has already been sketched in the motivation of this paper. To our knowledge, no other AOM exists to date that provides support for atomic weaving of distributed aspects. The most important difference between existing AOMs and Lasagne is that their run-time weaving model does not support the distinction between deployment and activation.

DADO [55] (distributed aspects for distributed objects) middleware offers an innovative IDL languages for programming distributed aspects in heterogeneous environments. DADO has a mechanism, called Remote Multiple Contextual Invocation (RMCI), that provides a form of dynamic per-invocation selection of aspects as in Lasagne. However this mechanism is restricted within the scope of a single client–server interaction and, therefore, does not provide any guarantees with respect to systemwide consistency in the activation of aspects.

CAM/DAOP [15, 40] is a component and aspect based approach that combines the benefits of both CBSD and AOSD disciplines. An innovative aspect of CAM/DAOP is that it specifies the composition of components and aspects using an architectural description language (ADL) [45], named DAOP-ADL. This ADL-based approach provides an interesting complement to Lasagne (or any run-time weaving approach for that matter). After all, using DAOP-ADL, application deployers are able to comprehend the overall aspect-component composition, facilitating a better understanding and easier verification of the application *as a whole*. This is of course an important software engineering quality that improves the safety and robustness of deploying aspects at run time.

Lasagne would, in the same vain, be nicely complemented by model-driven middleware (e.g., [18]) and aspect-oriented domain modeling (e.g., [20]). Combining these approaches leads to a very powerful concept. Design models of aspects and applications can be specified, composed and possibly verified. Once composed, these models can be automatically synthesized to deployment metadata for a specific middleware platform of choice.

Referring to our discussion on the applicability of Lasagne to adaptive QoS, Duzan et al. [14] have built an AOM platform, called QuO, that allows to separate adaptive QoS concerns from the application. The middleware also provides support for coordinating adaptation operations across multiple processes and computer nodes. Lasagne could be useful for automating that coordination support.

Dynamic architectures for network protocols. Some of the problems related to ensuring consistency in the activation of aspects (see Sect. 4.2) have already been addressed, to a certain extent, in networking protocols. It is common for messages to have dynamically computed headers in order for peers to process the message correctly by dispatching it through a series of protocols. Also at the client-side, the shift towards dynamic protocol stack composition architectures (e.g., [32, 35]) makes that the protocol stack at the server-side must have the intelligence to deal with this correctly.

There are two important differences between Lasagne and network protocols however. First, Lasagne is more generic because it is aimed to support adaptations at different layers of software, including the application layer [52], middleware layer [50] and network layer [22]. Secondly, Lasagne's support is reusable across multiple application domains. In protocol stacks the support for message dispatching is wired inside the protocol stack architecture itself, whereas in Lasagne all support is externalized in a separate run-time system that can be applied to any family of applications.

Another related technology in this field is programmable networking. Programmable networking stands for a whole spectrum of networking technologies that aim to introduce more openness and adaptability into today's networks by making network routers programmable. Historically there have been two main approaches to the provision of programmability in networks [13]: First, in the *active networking paradigm* [47] so called 'active packets' carry programs that execute on 'active' nodes. Secondly in the *open signaling approach* (e.g., [8]), routers export 'control interfaces' through which they can be remotely (re)configured by out-of-band, application-specific, signaling protocols.

Overall, the active networking approach is the most dynamic since it enables application-specific adaptations across the network in a very fine-grained scale. However, it is perceived as more vulnerable to security treats from malicious applications, and there is also the problem of domain mismatch [31]: application programmers, who wish to customize the underlying network to their preferences, are suddenly confronted with a domain (in this case, active networks) that is completely different from their familiar application domain. The open signaling approach does not suffer from these problems since it supports adaptations that are programmed by a trusted and informed third party. However, open signaling is an order of magnitude less dynamic.

We position Lasagne in the middle of these two approaches:

- Similar to the open signaling approach, adaptations are programmed by a trusted application deployer who is an expert at his application domain (in this case, network protocols).
- Similar to the active network approach, once adaptations are deployed, they can be dynamically activated on a per message basis by contextual actors (which in this case act as the (client) applications wanting to customize the underlying network).

Adaptations in programmable networking can be more complex compared to the range of adaptations that are supported in Lasagne, but this also leaves the door open for more intrusive changes that endanger safety. A second important difference is that Lasagne is more generic. The application area of programmable networking is more like a niche, and of course, the approach is somewhat optimized for that niche.

Dynamic software architectures. In Regis [30] and ArchStudio [38], distributed applications are constructed from a number of components and connectors that encapsulate the interactions between these components. Here, the term “component” corresponds with the traditional view of component-based development.

The emphasis of these works is on the description, dynamic reconfiguration and evolution of the application’s architecture. Connectors help to decouple components from one another and the systems use configuration languages to describe the configuration of the components. The run-time structure of the application is altered by applying a program written in the configuration language to the current architecture, thus generating a different arrangement of components and connectors. Our goal differs from these works in that Lasagne focusses on adapting an application to context-specific needs, instead of coping with run-time software evolution in general. Furthermore, Lasagne’s adaptation process is based on a systemwide additive refinement of existing application components with crosscutting aspects, rather than replacing existing components with new ones and switching connectors.

Coordinated adaptation in distributed systems. In the context of the above component-based reconfiguration approaches there is a whole field of research (e.g., [1, 2, 9, 19, 21, 26, 38]) that looks at the issue of how to achieve coordinated adaptation in a distributed system: when a reconfiguration involves replacing multiple components, atomicity problems appear that are very similar to those that have been studied in this paper. The existing approaches to coordinated adaptation in a component-based setting can be classified according to

- Their impact on service disruption as either *disruptive* (e.g., [2]) or *non-disruptive* (e.g., [1]) approaches
- Their ability to replace an existing component by a new version or to let both be used simultaneously as either *single-configuration-managed* (e.g., [2]) or *multiple-configurations-managed* (e.g., [19]) approaches

Lasagne clearly belongs to the multiple-configurations-managed and non-disruptive category.[12] Remarkably, we have not found any coordinated adaptation approach in a component-based setting that combines nondisruptive and multiple-configurations-management. Admittedly, the problem of coordinated adaptation has been mostly studied in a setting where adaptation is triggered out-of-band, rather than in-band as Lasagne supports.

Another important difference is that some of the aforementioned approaches also employ techniques and protocols for preserving global state consistency [33] and application state invariants [2]. As stated in Sect. 2, Lasagne has not been designed for addressing global state consistency.

Reflective middleware. Other related work are projects (e.g., [6, 25]) that use reflection [29] to achieve adaptive middleware. Generally, the use of reflective techniques allows for more powerful[13] (i.e., intrusive) adaptations in middleware and distributed applications. However, employing reflection for context-sensitive adaptations yields often a too complex system that cannot be understood by the typical contextual actor (or more specifically, the human operators involved). In this respect, Lasagne offers a very simple adaptation model to the contextual actor. For example, by simply tagging aspect identifiers to its messages, (client) applications can dynamically adapt the underlying middleware to its preferences.

More recent research on reflective middleware [12] has focussed on combining reflection with component framework technology [46] in order to preserve behavioral integrity. Much of this work has focussed on enforcing semantically compatible compositions of middleware components. Lasagne could be useful for implementing some of these integrity requirements, especially those that relate to maintaining semantic compatibility across multiple nodes.

7 Conclusion

This paper presents a model and an architecture for middleware that supports run-time and atomic weaving of distributed aspects without causing any disruption in the online availability of the distributed application. We have argued that the concept of aspects is well suited for modeling complex behavioral adaptations, i.e., not limited to the modification of configuration parameters. Furthermore, we assume that these adaptations do not introduce external state dependencies towards invoking clients.

[12] The first point may require some explanation. Lasagne offers a multiple-configuration-managed approach because the flow-dependent activation mechanism implies that an application may be running in different versions simultaneously.

[13] Relative to the support of typical AOM.

The run-time weaving process has been divided into a deployment phase and a subsequent activation phase: an aspect is first prewoven into so-called deployment metadata, and then the aspect is activated on a per message basis. The architecture of Lasagne documents that this division between deployment and activation creates a solution for the atomic weaving problem which includes three elements. (1) Aspects must be completely and correctly prewoven into deployment metadata. (2) When an aspect is activated for a particular collaboration instance this decision must be consistently propagated within the subsequent message flow. (3) A simple coordination protocol acts as a synchronization barrier to ensure that activation is only possible for aspects that have been completely and correctly deployed. This approach is not only a guarantee for atomicity; the division into deployment and activation makes the run-time weaving process also more secure and scalable.

In our future work we aim to develop optimizations of the Lasagne model. We will also address atomic weaving techniques that target out-of-band adaptations and application architectures with state-full sessions.

Acknowledgments. This work has been supported by the Institute for the promotion of innovation by science and technology in Flanders (IWT). We would like to thank the anonymous reviewers and Mario Südholt for their useful comments on previous drafts. Many thanks to Nico Janssens, Bert Lagaisse and Frans Sanen for the fruitful discussions and for their feedback.

References

[1] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: how to upgrade distributed systems. In: *Ninth Workshop on Hot Topic in Operating Systems (HotOS-IX)*, 2003

[2] J.P.A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for corba. In: *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, IEEE Computer Society, pp. 197–207, 2001

[3] K. Arnold and J. Gosling. *The Java Programming Language.* The Java Series, Addison-Wesley, New York, 1996

[4] K. Nahrstedt and B. Li. A control-based middleware framework for quality of service adaptations. *IEEE Journal of Selected Areas in Communication, Special Issue on Service Enabling Platforms*, 17(9):1632–1650, 1999

[5] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. Role object. In: *Pattern Language of Program Design 4*, Addison-Wesley, New York, pp. 15–32, 2000

[6] G.S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In: *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Springer, Berlin Heidelberg New York, pp. 191–206, 1998

[7] G.S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in openorb. In: *WOSS '02: Proceedings of the First Workshop on Self-Healing Systems*, ACM, New York, pp. 9–14, 2002

[8] P.R. Chandra, A. Fisher, C. Kosak, T. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: customizable resource management for value-added network services. In: *ICNP*, pp. 177–188, 1998
[9] W.-K. Chen, M.A. Hiltunen, and R.D. Schlichting. Constructing adaptive software in distributed systems. In: *Proceedings—the 21st IEEE International Conference on Distributed Computing Systems*, IEEE Computer Society, pp. 635–643, 2001
[10] R. Clark, D.E. Jensen, and F.D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In: *Proceedings of the USENIX Workshop on Microkernel and Other Kernel Architectures*, April 1992
[11] 2006 Microsoft Corporation. Microsoft office live. http://www.microsoft.com/office/officelive/default.mspx
[12] G. Coulson, G.S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002
[13] G. Coulson, G.S. Blair, D. Hutchison, A. Joolia, K. Lee, J. Ueyama, A.T.A. Gomes, and Y. Ye. NETKIT: a software component-based approach to programmable networking. *Computer Communication Review*, 33(5):55–66, 2003
[14] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky. Building adaptive distributed applications with middleware and aspects. In: *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, ACM, New York, pp. 66–73, 2004
[15] L. Fuentes, M. Pinto, and P. Sánchez. Dynamic weaving in cam/daop: an application architecture driven approach. In: *Proceedings of the Dynamic Aspect Workshop in Conjunction with AOSD 2005*, March 2005
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, New York, 1995
[17] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM System Journal*, 42(1):5–18, 2003
[18] A. Gokhale, D. Schmidt, B. Natarajan, J. Gray, and N. Wang. Model-driven middleware, 2004
[19] J. Gouveia, G. Koutsoukos, L. Andrade, and J. Fiadeiro. Tool support for coordination-based software evolution. In: *TOOLS Europe 2001*, IEEE Computer Society Press, Los Alamitos, 2001
[20] J. Gray, T. Bapty, S. Neema, D.C. Schmidt, A. Gokhale, and B. Natarajan. An approach for supporting aspect-oriented domain modeling. In: *Proceedings of the Second International Conference on Generative Programming and Component Engineering*, Springer, Berlin Heidelberg New York, pp. 151–168, 2003
[21] N. Janssens, W. Joosen, and P. Verbaeten. NeCoMan: middleware for safe distributed-service adaptation in programmable networks. *IEEE Distributed Systems Online*, 6(7), July 2005
[22] B.N. Jørgensen, E. Truyen, F. Matthijs, and W. Joosen. Customization of object request brokers by application specific policies. In: *Middleware 2000, LNCS vol. 1795*, Springer, Berlin Heidelberg New York, pp. 144–163, 2000
[23] G. Kiczales, J. des Riviers, and D. Bobrow. *The Art of the Meta-Object Protocol*, MIT Press, Cambridge, 1991
[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. In: *ECOOP2001—Object-oriented Programming, LNCS vol. 2072*, Springer, Berlin Heidelberg New York, pp. 327–353, 2001

[25] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L.C. Magalhãs, and R. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In: *Middleware 2000*, *LNCS vol 1795*, Springer, Berlin Heidelberg New York, pp. 121–143, 2000

[26] J. Kramer and J. Magee. The evolving philosopher problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990

[27] B. Lagaisse, B. De Win, and W. Joosen. SoBeNet: financial case study—part 1: requirements and analysis. Report CW 404, Department of Computer Science, K.U.Leuven, Leuven, Belgium. http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW404.abs.html, 2005

[28] J. Lowy. Contexts in .NET: decouple components by injecting custom services into your object's interception chain. *MSDN Magazine*, March 2003

[29] P. Maes. Concepts and experiments in computational reflection. In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices 22(12), pp. 147–155, 1987

[30] J. Magee, N. Dulay, and J. Kramer. Regis: a constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5), 1994

[31] F. Matthijs, N. Janssens, and P. Verbaeten. Automatic service composition: a case for active networks usability. Report CW 356, Department of Computer Science, K.U.Leuven, Leuven, Belgium. http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW356.abs.html, January 2003

[32] S. Michiels, N. Janssens, L. Desmet, T. Mahieu, W. Joosen, and P. Verbaeten. A component platform for flexible protocol stacks. In: *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, *LNCS vol. 3778/2005*, Springer, Berlin Heidelberg New York, pp. 185–208, November 2005

[33] K. Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*, PhD thesis, Imperial College, March 1999

[34] R. Monson-Haefel. *Enterprise JavaBeans, 3rd edn.* O'Reilly, USA, September 2001

[35] R. Morris, E. Kohler, J. Jannotti, and M. Frans Kaashoek. The click modular router. In: *SOSP*, pp. 217–231, 1999

[36] O. Nierstrasz. Composing active objects—the next 700 concurrent object-oriented languages. In: *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, Cambridge, pp. 151–171, 1993

[37] Object Management Group. *The Common Object Request Broker: Architecture and Specification—Version 3.0.* http://doc.ece.uci.edu/CORBA/formal/02-06-33.pdf, July 2003

[38] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In: *Proceedings of the 1998 International Conference on Software Engineering*, IEEE Computer Society Press/ACM, New York, pp. 177–186, 1998

[39] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: a flexible solution for aspect-oriented programming in Java. In: *Reflection 2001*, *LNCS vol. 2192*, Springer, Berlin Heidelberg New York, pp. 1–24, 2001

[40] M. Pinto, L. Fuentes, and J.M. Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In: *Proceedings of the Second International Conference on GPCE*, *LNCS vol. 2830*, Springer, Berlin Heidelberg New York, pp. 118-137, September 2003

[41] A. Popovici, G. Alonso, and T. Gross. Just in time aspects: efficient dynamic weaving for Java. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, ACM, New York, pp. 100–109, 2003

[42] A. Popovici, A. Frei, and G. Alonso. A proactive middleware platform for mobile computing. In: *Middleware 2003*, *LNCS vol. 2672*, Springer, Berlin Heidelberg New York, pp. 455–473, 2003

[43] B. Robben. *Language Technology and Metalevel Architectures for Distributed Objects*, PhD thesis, Katholieke Universiteit Leuven, 1999

[44] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. *Innovations in Internetworking:* 379–390, 1988

[45] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, New York, 1996

[46] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM/ Addison-Wesley, New York, 1998

[47] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEECM*, 35(1):80–86, 1997

[48] M. Tokoro. The society of objects. *OOPS Messenger*, 5(2):3–12, 1994

[49] E. Truyen. *Dynamic and Context-Sensitive Composition in Distributed Systems*, PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven, 2004

[50] E. Truyen, B. Nørregaard Jørgensen, and W. Joosen. Customization of component-based object request brokers through dynamic reconfiguration. In: *Technology of Object-Oriented Languages and Systems—TOOLS 33*, IEEE Computer Society, pp. 181–194, 2000

[51] E. Truyen, B. Vanhaute, W. Joosen, and P. Verbaeten. Consistency management in the presence of simultaneous client-specific views. In: *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, IEEE Computer Society, pp. 501–510, 2002

[52] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and selective combination of extensions in component-based applications. In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, IEEE Computer Society, pp. 233–242, 2001

[53] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997

[54] N. Wang, D.C. Schmidt, and C. O'Ryan. Overview of the corba component model. In: *Component-Based Software Engineering: Putting the Pieces Together*, pp. 557–571, 2001

[55] E. Wohlstadter, S. Jackson, and P. Devanbu. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In: *ICSE 2003*, pp. 174–186, 2003

Appendix 1 Coordination Protocol for Aspect Unweaving

Figure 10 presents the coordination protocol for unweaving an aspect at run time. Aspect unweaving is useful for adaptation scenarios where the stakeholders of the distributed application want to remove a certain aspect from the application because of various reasons, e.g., the aspect is not often activated by the contextual actors, it is malfunctioning, or it is too heavy in terms of resource usage. In any case, these adaptation scenarios always involve a request from the application deployer to unweave an aspect with, say, aspect identifier F.

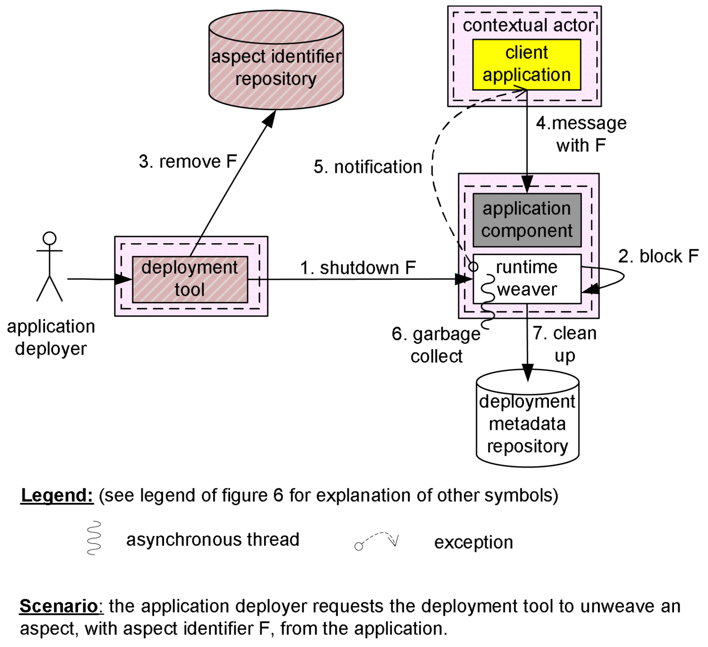

aspect identifier repository
contextual actor
client application
4.message with F
3. remove F
5. notification
application component
2. block F
application deployer
deployment tool
1. shutdown F
runtime weaver
6. garbage collect
7. clean up
deployment metadata repository
Legend: (see legend of figure 6 for explanation of other symbols)
asynchronous thread
exception
Scenario: the application deployer requests the deployment tool to unweave an aspect, with aspect identifier F, from the application.

Fig. 10. Coordination protocol for aspect unweaving at run time

The coordination protocol consists of a deactivation phase and an un-deployment phase. The deactivation phase consist of five steps (see Fig. 10):

(1) The deployment tool requests all run-time weaver instances to shutdown the execution of aspect F.
(2) Each run-time weaver instance configures itself such that it will block new messages from the outside (i.e., messages that enter the application architecture).
(3) The deployment tool removes aspect identifier F from the aspect identifier repository.
(4) A client application sends a client request carrying the aspect identifier F.
(5) The client application receives a run-time exception from the run-time weaver that aspect identifier F is not available anymore.

The un-deployment phase consists of two steps:

(6) When all ongoing collaboration instances for which aspect identifier F is activated have completed their execution, an asynchronously running thread

will update the internal data structures of each run-time weaver instance such that all advice code and aspect instances, associated to F, can be released from memory,

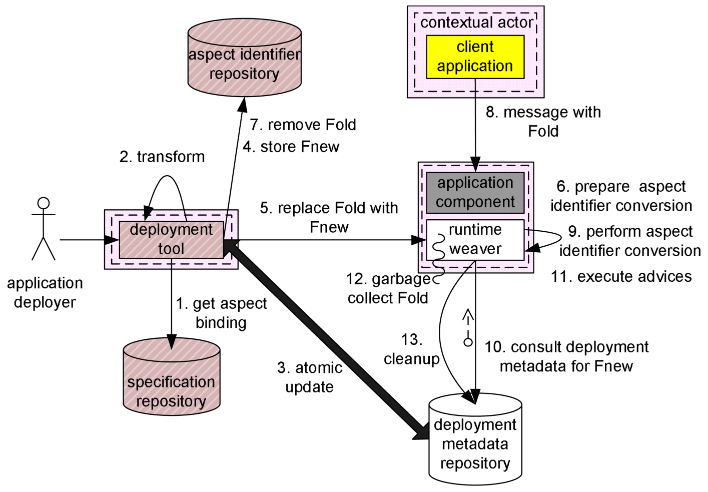
(7) Each run-time weaver notifies the local deployment metadata repository, such that the latter can remove all relevant advice-to-joinpoint bindings from the deployment metadata.

Appendix 2 Coordination Protocol for Run-Time Aspect Replacement

Figure 11 presents the coordination protocol for replacing an aspect at run time. Replacement means that the application deployer requests the deployment tool to replace an aspect, having aspect identifier F_{old}, with a new aspect identifier, having aspect identifier F_{new}.

An important property of the coordination protocol is that replacement is performed transparent for the contextual actors that have activated the old aspect. To achieve this transparency, the second phase employs automatic aspect identifier conversion.

The coordination protocol is a combination of the coordination protocols for weaving and unweaving, and therefore consists of three phases: deploying the new aspect, deactivating the old aspect and activating the new aspect, and



Legend: see legend of figure 10 for explanation of symbols

Scenario: the application deployer requests the deployment tool to replace an aspect, having aspect identifier Fold, with a new aspect, having aspect identifier Fnew. This should be performed transparent for the clients who have activated F_old.

Fig. 11. Coordination protocol for aspect replacement at run time

un-deploying the old aspect. Notice that deactivation and activation occurs in parallel. This is possible because we target a state-less session model (i.e., aspects do not engage external state dependencies with invoking clients).

Deploying the new aspect. This phase consists of five steps (see Fig. 11):

(1) The deployment tools retrieves the aspect binding of the new aspect.
(2) The deployment tool transforms the aspect binding into advice-to-joinpoint bindings.
(3) The deployment tool updates the deployment metadata repository with the advice-to-jointpoint bindings in a deployment transaction.
(4) The deployment tool inserts the aspect identifier F_{new} into the aspect identifier repository.
(5) The deployment tool requests all run-time weaver instances to replace aspect F_{old} with aspect F_{new}.

Deactivating the old aspect; activating the new aspect. This phase consists of six steps:

(6) Each run-time weaver instance configures itself such that it will transform new messages from the outside (i.e., messages that enter the application architecture) that carry the aspect identifier F_{old} into carrying F_{new}.
(7) The deployment tool removes aspect identifier F_{old} from the aspect identifier repository.
(8) A client request carries the aspect identifier F_{old}.
(9) The client request is converted by the run-time weaver such that F_{old} is detached from the client request and instead F_{new} is tagged to it.
(10) The run-time weaver consults the deployment metadata for F_{new}.
(11) The run-time weaver instance updates its internal data structures and executes the advices that are associated to F_{new}.

Un-deploying the old aspect. This phase consists of two steps:

(12) When all ongoing collaboration instances for which aspect identifier F_{old} is activated have completed their execution, an asynchronously running thread will update the internal data structures of each run-time weaver instance such that all advice code and aspect instances, associated to F_{old}, can be released from memory,
(13) Each run-time weaver instance notifies the local deployment metadata repository, so that the latter can remove all relevant advice-to-joinpoint bindings, associated to F_{old}, from the deployment metadata.

Appendix 3 Load-Time Consultation of Deployment Metadata

The specification of this operation is presented in Fig. 12 as an UML model. As shown in the figure, the deployment manager receives a record structure of two elements. First, information related to where the aspects' implementations can be downloaded. Second, for every application operation, an ordered list of advice-to-joinpoint bindings is returned.

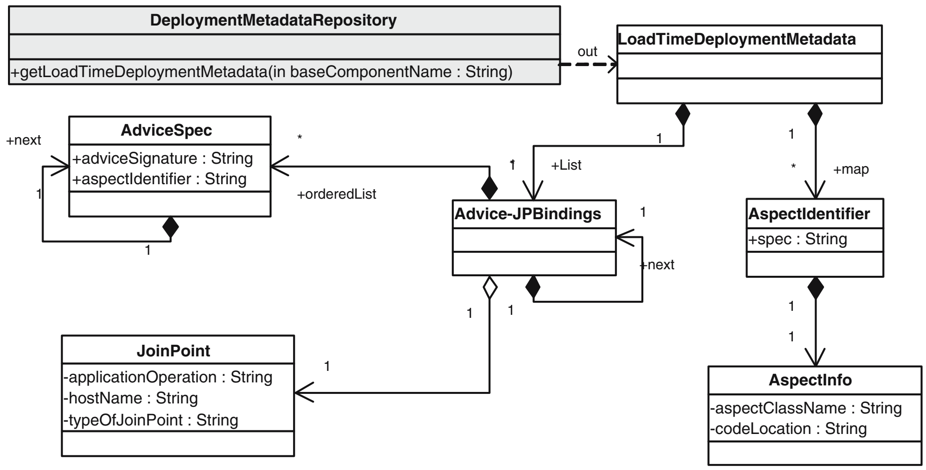

DeploymentMetadataRepository
+getLoadTimeDeploymentMetadata(in baseComponentName : String)
out
LoadTimeDeploymentMetadata
AdviceSpec
+adviceSignature : String
+aspectIdentifier : String
+next
1
1
*
+orderedList
1
+List
1
Advice-JPBindings
1
+next
1
1
1
+map
*
AspectIdentifier
+spec : String
1
1
JoinPoint
-applicationOperation : String
-hostName : String
-typeOfJoinPoint : String
1
AspectInfo
-aspectClassName : String
-codeLocation : String

Fig. 12. Deployment metadata repository interface for consulting deployment metadata at load-time

TOSKANA: A Toolkit for Operating System Kernel Aspects

Michael Engel and Bernd Freisleben

Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Str., 35032 Marburg, Germany
{engel, freisleb}@informatik.uni-marburg.de

Abstract. The development process for operating system kernels in Unix-like systems is becoming increasingly complex. The simple, easily understandable kernels of the 1970s have evolved into giant software systems consisting of hundreds of modules implemented in millions of lines of code, resulting in greatly increased resource and time overhead for bug fixes as well as functional improvements. Many code modifications and additions in kernel code are crosscutting, since they exhibit *cross-module* or *cross-layer* functionality, thus an aspect-oriented programming (AOP) approach is well suited for solving these problems. Operating system code implicitly involves dynamic behavior due to various threads in kernel mode that run in different process or hardware contexts. Methods to solve the crosscutting problems have to be able to adapt dynamically to this changing environment; as a result, dynamic AOP inside the kernel is required to implement required functionality.

This paper analyzes the feasibility of using dynamic AOP in kernel code. Using the TOSKANA system, it is shown that implementing dynamic AOP features is not only possible in operating system kernels, but also realizable with a justifiable overhead. First, the cross-module and cross-layer properties inherent in kernel code are presented, with special consideration of crosscutting concerns in procedural C kernel code. Then, TOSKANA—our toolkit for deploying dynamic aspects into an operating system kernel—is introduced. TOSKANA provides *before*, *after* and *around* advice for in-kernel functions and supports the specification of pointcuts as well as the implementation of aspects themselves as dynamically exchangeable kernel modules. The use of TOSKANA is demonstrated by several examples of cross-module as well as cross-layer problems and their solutions using dynamic AOP functionality inside the kernel. Performance results are presented to characterize the aspect deployment overhead incurred by using TOSKANA.

1 Introduction

Unix kernels are complex software systems that evolved from a simple implementation of anticipatory concepts into a software system typically spanning millions of lines of code. The flexible architecture of Unix has permitted the system to acquire new functionality along the way, like layered architectural frameworks for network and file system implementation, which nowadays seem like a natural fit for the Unix structure. Compared to application software, a kernel has to handle many more tasks concurrently, while always being alert to asynchronous, external events that change the flow of control in the system. Much of the data kept inside the kernel depends on changing conditions and thus is expected to dynamically change over time. The evolution of the system as well as the asynchronous nature of control flow in a kernel have implications for the development of kernel-mode code so that writing and evolving kernel code is a demanding task compared to application-level development.

In many cases, this involves modification of a large portion of code scattered throughout the system, since the related concerns are either crossing layers inside the operating system kernel or have to be implemented in a set of components that realize similar functionality (e.g., diverse file system implementations or supported binary executable formats) simultaneously. Thus, in many cases, kernel functionality consists of a set of crosscutting concerns. Consequently, an aspect-oriented programming (AOP) approach is the obvious way to handle the complex interactions required to implement kernel extensions as well as bug fixes.

Kernel code, however, is mostly not run on behalf of itself, but it rather acts according to the requirements from user applications or the hardware under its control. While applications request kernel functionality using a well-defined system called API (often wrapped in special libraries) in a synchronous way, hardware interaction often takes place in asynchronous ways, e.g., by reacting to hardware interrupts or processor exceptions. As a result, kernel code is always executed in one of several, dynamically changing *contexts*—accordingly, the code executed on behalf of an application or hardware device that has to adapt dynamically to context switches. Thus, pure static approaches to in-kernel AOP do not suffice in all circumstances; rather, a dynamic AOP approach is required that not only allows loading aspect code into kernel space on demand, but also permits pointcut descriptions to depend on in-kernel contexts like process states and interrupt contexts. Being able to implement AOP functionality inside the kernel without recompiling the source code also implies that this functionality can be added and removed at run time, thus eliminating the need for time-consuming and work-disrupting reboots.

After discussing the peculiarities of cross-module and cross-layer interaction in a typical Unix-like kernel, the paper presents TOSKANA—our toolkit for operating system kernel aspects with nice applications [1]. It provides the first set of tools, macros and libraries for developing and deploying dynamic aspects in the kernel space of NetBSD, an open source system derived from UC Berkeley's 4.4BSD[1] operating system [2]. The paper illustrates how kernel programming based on TOSKANA can be performed and presents details of the toolkit's design and implementation. In addition, an analysis of crosscutting concerns in procedural code is presented. Finally, novel applications (compared to [1]) solving cross-layer as well as context-dependent problems using TOSKANA are demonstrated and an evaluation of TOSKANA's performance is presented.

[1] BSD = Berkeley system distribution.

The paper is organized as follows. Section 2 discusses crosscutting properties in the NetBSD kernel code. Section 3 presents the design of TOSKANA and illustrates how kernel programming using TOSKANA is performed. Section 4 shows application examples from the areas of network optimization and Grid computing. Section 5 discusses the implementation of TOSKANA. Section 6 presents an evaluation of the performance characteristics of a TOSKANA-enhanced system. Section 7 summarizes related work. Section 8 concludes the paper and outlines areas for further research.

2 Crosscutting Concerns in NetBSD

Crosscutting concerns can be defined as a set of related functionality that, when implemented in a software system, defies the existing modularization structure of the system. Traditional languages and modularization mechanisms suffer from a limitation—the program can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another. To gain an improved insight into the characteristics of crosscutting concerns in NetBSD, structural as well as implementation-language-specific facets of crosscutting concerns are described in the following sections.

2.1 Crosscutting Properties in the NetBSD System Structure

To identify crosscutting concerns in a code base, the modularization structure of the system has to be examined. A typical operating system kernel has to perform various different tasks, many of which are dependent on functionality provided by other functional units of the kernel. For example, file system operations depend on user and process management to provide access control as well as on the virtual memory management system to manage buffer space and memory-mapped files. This complex and variable set of tasks is reflected in the source code structure of a kernel which represents a high-level modularization of the kernel source code. Functional subsystems of the kernel are contained in separate source code directories under the `/usr/src/sys/` top-level source code hierarchy. This main modularization structure of the kernel code source tree is further subdivided in several directories, e.g., each architecture supported is contained in a separate directory inside `arch` and several small file system implementations are contained in the `fs` and `miscfs` directories.

2.1.1 System Structure

The NetBSD kernel is written almost completely in ANSI C; only about 9% of the code is written in assembler—of which in turn 90% are machine-dependent parts. Figure 1 summarizes the distribution of functionality in a snapshot of the current development version.

Consisting of about 6,000 source files containing nearly 3 million lines of C and assembler source code (including comments and empty lines), the NetBSD kernel is a rather complex system.

Part of NetBSD	Nr. of C files	Lines of C code	Asm. files	Lines of Assem.
Basic kernel code	98	87229	0	0
Device Drivers	1157	838157	0	0
File System Code	133	105408	0	0
Networking	268	239051	0	0
Libraries	154	25718	205	19163
Arch. independent	2278	1490932	212	24268
Arch. dependent	3039	1215513	412	217082
Complete system	5317	2706445	624	241350

Fig. 1. Statistics of NetBSD 2.0 kernel code structure

Of interest for the application of software technology tools is the overall code complexity. Analyses [3] have shown that the number of source code lines, the number of files and the size of the kernel source code grew exponentially over the last 10 years, whereas the number of developers grew mainly in a linear fashion. Further analysis of release cycles of the NetBSD kernel [3] showed that the number of lines of code between successive major releases of NetBSD as well as the time between these releases was growing exponentially for the last 5 years, whereas the overall number of developers contributing to the code at best grew linearly, resulting in an ever-growing overhead of code that a single developer has to create and maintain.

An informal survey of other open source systems like the Linux and FreeBSD kernels and the Apache web server showed similar results, indicating that the time to implement additional features (indicated by the temporal distance between successive major releases of the respective system) is growing remarkably. While a thorough analysis of the reasons for the increase in complexity is certainly required, at least some part of the complexity increase of the systems may be attributed to the growing number of modules and, consequentially, the increasing number of crosscutting interdependencies between these systems. Thus, aspect-oriented software development technologies can provide part of a solution to speed up system-level code development.

2.1.2 Layers

Several subsystems of the kernel provide functionality that is well suited to be further subdivided into smaller layers, each implementing a strictly defined set of functions relying on services from the lower levels and providing functionality to upper layers using well-defined APIs.

Since issues concerning functionality that spans several layers are important sources for crosscutting concerns, the largest layered structure inside the NetBSD kernel—the network protocol stack—is discussed below.

Layered Networking Structure. One well-known example for layered software design is the ISO/OSI network protocol stack [4] which decomposes the tasks of network protocol software into seven layers, each implementing a set of functionality based on functions provided by lower layers. Real-world implementations of network protocols, however, mostly employ a more simple method of organization.

The NetBSD TCP/IP network stack follows a layered model without strict adherence to the OSI model. On the hardware level, device drivers take care of the physical layer and part of the data link layer. Addressing at the data link layer is handled by the ARP module, while data encapsulation, authentication and service functions of the data link layer reside in various protocol-specific modules.

The network subsystem is logically divided into three layers. These layers manage three tasks: interprocess data transport, internetwork addressing and message routing and transmission–media support. The first two layers are made up of modules that implement communication protocols; the software in the third layer generally includes a protocol sublayer, as well as one or more network device drivers.

The layering described here is a *logical layering*. Software that implements network services may use more or fewer communication protocols according to the design of the network architecture being supported. For example, raw sockets often use a null implementation at one or more layers. At the opposite extreme, *tunneling* of one protocol through another uses one network protocol to encapsulate and deliver packets for another protocol and involves multiple instances of some layers. Certain properties of the NetBSD network functionality, however, cannot fit into this strict layer model. Examples for this crosscutting functionality are quality of service (QoS) control and packet filtering.

Quality of service is a property that has to be provided in response to the requirements of an application. One common example for QoS is a guaranteed bitrate for a certain application, e.g., voice over IP or video streaming. To maintain the required throughput, the network stack has to ensure priorities at every layer of the model, potentially delaying or even discarding packets generated by other applications to give "urgent" packets precedence. In addition, the networking code has to coordinate several applications that simultaneously require QoS guarantees on the same host and detect unsatisfiable conditions.

Packet filtering has to be performed in various layers in the network stack, depending on the filtering criteria provided to the NetBSD kernel. Such criteria can be MAC-level addresses, IP addresses, source or destination port numbers, protocol types, etc. Filtering potentially can occur in every layer, thus a network-filtering infrastructure is crosscutting in nature. This is obvious when observing that the implementation of the Berkeley packet filter (BPF, [5]) uses a large number of hooks throughout the layers of the network code in order to enable filtering.

Another critical issue in the design of network stacks is to provide as much throughput as possible. Traditionally, in the BSD network stack, code in one layer provided the adjacent layer with a copy of the relevant parts of a packet in transit, thus requiring a large amount of memory copy operations. On fast networks, this severely constrains network throughput. As a counter-measure, most systems nowadays implement a zero-copy policy in their network stack. Here, a large buffer is allocated for each packet and code in each layer only accesses the layer-specific information, thereby avoiding the copying of packet data throughout the layers. The coordination of buffer space for packets, detection of the ways packets are to be routed through the layers and the necessary calculations to ensure a sufficiently large buffer for each packet is allocated are all crosscutting concerns.

A different approach to solving crosscutting as well as performance problems is to replace the traditional network stack architecture by a network heap [6]. This solves some common performance problems and eliminates crosscutting problems by changing the modularization structure of the system at the expense of more complex programming. A TOSKANA-based approach, however, preserves the basic layered network stack structure while providing mechanisms for implementing crosscutting concerns.

2.1.3 Contexts

In addition to handling various different tasks, each of which has to be provided in several varieties, kernel code also faces another dependency. While traditional, single-threaded user-mode applications rely on the notion of running their code in a single, well-defined context, operating system code usually does not run on behalf of itself, but rather in reaction to asynchronous events which are generated from the hardware as well as synchronous calls from user-mode processes running on top of it. This has implications for crosscutting concerns, since a crosscut now is not only related to locations in the source code, but rather extends into an additional dimension—the process or hardware context in which the code containing the crosscutting concerns is executed.

Process Contexts. The basic operation model of a Unix kernel implies that code in the kernel itself is always executed in a certain *context* and thus has to keep different state, accordingly. The most important contexts are process contexts and hardware contexts.

When a process executes a system call, control is transferred from user- to kernel-mode using a special machine code instruction like `trap` or related mechanisms like software interrupts or call gates. This mechanism interrupts the code flow of the current process, changes the execution mode of the processor to a privileged state and transfers control to a location in kernel virtual memory corresponding to the address of the *system call handler*. This handler analyzes the system call number that is passed on the stack or in a processor register and further dispatches the flow of control to the appropriate in-kernel system call function. This system call code—and all functions called on behalf of this code at this moment—is said to be running in the *context* of the process that executed the system call.

Since Unix is a multitasking system, several user-mode processes are permitted to run (semi)concurrently and it is not uncommon for two unrelated processes to be executing the kernel code of a common system call (like `read`) at the same time, since processes might also be interrupted or rescheduled while running in kernel mode. Due to the ubiquity of context changes many pieces of code in the kernel have to deal with, a mechanism is required to handle code adaptation to a new context. As a consequence, code that modifies the execution of instructions has to be activated or deactivated depending on the context. Thus, a purely static approach to handling in-kernel crosscutting concerns can only provide the required functionality with increased overhead by introducing additional if-clauses or similar constructs into the code flow. This effect also has to be considered in multithreaded user-mode programs, of course. In kernel code, due to the inherent concurrency, this behavior is a common case and has to be considered when developing a dynamic aspect system for kernel code.

Exceptions are another common case of control transfer from user to kernel mode initiated by the actions of a running process. Some exceptions occur regularly, like page faults indicating to the virtual memory subsystem of the kernel to retrieve further pages from the on-disk executable or the swap space area of the hard disk. Other exceptions, like arithmetic exceptions (division by zero, overflow, etc.), occur unwantedly and usually indicate a programming error in the user-mode process that has to be handled by the kernel—usually, by terminating the operation of the faulting process. These exception handlers, too, run in the context of the corresponding user-mode process.

Asynchronous (Hardware) Contexts. Some parts of the kernel do not operate synchronously in response to user-mode actions, but rather have to react asynchronously to unpredictable external events, usually signaled by devices through some sort of interrupt mechanism.

Interrupt handler code running in response to external interrupts is running in a special *interrupt context* and usually runs uninterruptible by temporarily turning off all interrupt handling (like in Linux) or adjusting the interrupt level of the processor so that only higher-privileged interrupts may be processed. Current trends, however, try to minimize the time spent in an interrupt handler and relegate most of the traditional bottom-level work of the handler to special in-kernel *interrupt handler threads* that run independently of the interrupt context.

2.2 Summary

Working with kernel code creates a specific set of problems when handling crosscutting concerns. These problems can be summarized as follows:

– Kernel code is mostly written in procedural languages like C, which provides only poor modularization facilities.

– The layered structure of many kernel components requires cross-layer interaction for functionality relevant to more than one layer; this in turn creates new crosscutting concerns.
– Running kernel code in several, changing process and hardware contexts requires dynamic behavior of components that implement crosscutting concerns.

Compared to application-level code, restarting kernel code is usually not possible. In order to reach acceptable system uptimes, dynamic insertion of aspects into a running system is required. In addition, it should be possible to activate and deactivate a deployed aspect on demand.

As a consequence, dynamic AOP proves to be a valuable tool for kernel code development. Our toolkit for dynamic AOP in kernel code, TOSKANA, is presented in the next section.

3 Kernel Programming with TOSKANA

Kernel programming tools should be minimally invasive and should not try to extensively change the typical approaches to code development and the skills required with regard to learning new language features or programming concepts. This is one of the main considerations in the development of TOSKANA—the methods to extend the kernel mode programming model should integrate nicely with the existing conventions and the tools provided should be able to be integrated into the existing toolset collection easily while still providing a large amount of new features and flexibility.

One of the tasks that reoccur on a regular basis is to implement functionality that represents a crosscutting concern—e.g., code that crosses layer boundaries, performs whole-system optimizations or provide extensive introspection capabilities. In addition, many of the changes to kernel code may only be relevant to a certain context or develop while a system is in operation and a restart is impractical. Thus, the aim was to develop a dynamic aspect-oriented approach that is integrated with common C programming methods as tightly as possible.

The TOSKANA toolkit [1] provides tools for weaving and unweaving aspects on demand into a running NetBSD kernel enabling the use of dynamic AOP in kernel space. The underlying NetBSD kernel source code does not need any modification to run TOSKANA so that dynamic AOP can be deployed on demand in a production system without having to apply refactoring methods or manual changes to the original code base.

In the current implementation, support for *before*, *after* and *around* advice at C function execution joinpoints is available. Advice code itself is also implemented as C code that has access to a set of macros that call functions from an in-kernel support library.

The following paragraphs describe typical tasks of a kernel programmer and the problems showing up when implementing these tasks that are related to crosscutting concerns.

3.1 Kernel Code Development Approaches

Many programmers are reluctant to approach kernel code development. Compared to application-level development, programming on the system level bears the stigma of being overly complex. Tool support for structuring and debugging kernel code is not as advanced as recent application-level tools and kernel code is generally hard to work with as well as error-prone due to the required direct hardware interaction and the inherent parallelism in the kernel.

All these points are valid to a certain extent. While the problems of parallelism and hardware interaction have been approached in the past by various solutions like microkernel-based systems that run most of the kernel code in user mode and virtualization features, approaching this problem is beyond the focus of this paper. Tool support, however, can be improved on various levels. The analysis of code and the build systems that are in use in current kernels like Linux and the various BSD-based open source systems show that kernel code development is basically still very similar to the methods used by the inventors of Unix in the 1970s. Discussions among kernel developers[2] show that established procedures and habits of kernel programmers are often hard to change, leading to a stagnation of the state of kernel code development tools. Experiments like introducing object orientation into the kernel, e.g., using C++ code in the Linux kernel around version 0.97, were met with harsh criticism and related code was, in turn, removed from the kernel rapidly.

Two of the—arguably valid—reasons cited for this rather conservative approach to kernel mode programming are code stability and performance, both of which are of paramount importance in kernel code. Tools that change the way kernel programming takes places should therefore not obstruct the programmer's view on kernel details, yet enable the developer to perform a set of tasks more efficiently.

However, other open source projects like the L4 microkernel [7] and the eCos embedded operating system have shown that less traditional approaches to kernel code development—both projects use a subset of C++ as their implementation language—is accepted by a developer community, provided that a reasonably simple and stable set of tools is available. In the case of eCos, this has already lead to an analysis of aspects in the C++ kernel base and their implementation using AspectC++ [8].

3.2 Problems in Kernel Code Development

Typical kernel-level programming tasks can roughly be divided into three classes.

The first class is driver development. As seen in Fig. 1, a disproportionally large amount of kernel code is concerned with controlling peripheral devices. This is especially prevalent in systems running on commodity PC hardware, since these have to support a large amount of third-party hardware devices.

The second class of tasks is the development of fixes for deficiencies in kernel code—these may be actual errors in the programming logic as well as security-

[2] For typical examples, see the Linux kernel mailing list at http://www.lkml.org

related fixes like removing buffer overflow vulnerabilities or timing problems that may lead to race conditions in the code.

Finally, the third class is the development of new subsystems for the kernel and the exchange or feature improvements of existing kernel components. Examples for these are the development of new scheduling approaches, improvements to code scalability (e.g., with respect to multiprocessor systems), implementing a new network protocol, support for a new bus system infrastructure or changes to the virtual memory management system to support larger physical address spaces.

Most of the device drivers that make up the *first class* of programming tasks are easily implementable using a standardized in-kernel API to handle buffer management, interrupt control, etc. Recent hardware developments in the PC area like offloading of software functions to separate CPUs or custom-integrated circuits on extension boards[3] have also demonstrated crosscutting concerns within device drivers. One example is the offloading of network processing tasks like packet checksumming, traffic filtering and shaping to intelligent network processors [9]. To support “primitive” network interface cards that only handle low-level frame reception and sending, the kernel nevertheless has to supply an implementation of all required functionality in software. In the nonhardware-accelerated case, a packet coming in from the network travels from the MAC layer of the TCP/IP stack all the way up to the socket layer where it is delivered to the application. Depending on the amount of capabilities of the networking hardware, however, some of the processing steps the kernel code performs are now implemented in separate hardware. For example, a driver for a network card that performs hardware checksumming has to insert the checked packet in the network stack’s data stream above the MAC layer. These cross-layer dependencies lead to driver code in which crosscutting concerns have to be implemented.

The *second class*, development of code to handle modifications of the logic behavior of the kernel and security problems in the system’s implementation, more obviously shows crosscutting concerns. Problems with the logic of control flow in the kernel often are a result of an incomplete understanding of the overall structure. A small change to a piece of kernel code may result in unexpected behavior of a different, but related, component of the kernel. One example is the addition of variables to in-kernel structures that are in turn mapped to hardware devices. A developer writing code that handles data in one of these structures wants to store additional status data in this structure and inserts a variable in the middle of it. Another unrelated piece of kernel code accesses this structure and, due to performance optimization, does not access the structure’s variables with their compiler-provided offsets but rather copies the structure as a whole using a `memcopy` operation on the provided pointer to the structure. This results

[3] Some of these concepts are in use for many years in mainframe environments—mainframe systems implement I/O offloading to the so-called “channel processors” for decades.

in data corruption, since the target of the copy operation is not aware of the changed source data and will incorrectly interpret the copied data.

Security problems are also often crosscutting. One of the most common causes of security problems is the exploitation of buffer-overrun problems in C code, which may result in an attack that overwrites pieces of the heap or stack, e.g., in order to modify stored function return addresses to code provided in the overflowed string value by the attacker. These buffer overflow problems are a result of *coding practice* and *implementation language deficits*, not of a specific implementation of code.

The *third class of kernel programming tasks*—developing new functionality—often has to implement changes to the overall behavior of a system. This behavior is defined by the actions and interactions of various system components, so in many cases having to handle crosscutting concerns will be unavoidable. The adaptation of an existing kernel to a multiprocessor environment requires the implementation of many crosscutting concerns. One of these concerns is the locking behavior used in kernel code. This is crosscutting since locks now may indicate that code in the kernel may be executed in parallel on all CPUs, thus it must be ensured that some different code in critical sections of the kernel is only being executed on a single CPU at any given instance of time.

The following sections present a top-down view of TOSKANA's structure and show how a programmer facing kernel programming can solve problems with the help of TOSKANA.

3.3 A Top-Down Overview of TOSKANA

TOSKANA consists of several user- and kernel-mode components that work together to create a toolset for dynamic in-kernel AOP. The user-mode tools provide functionality to dynamically load and unload aspect kernel modules and provide hooks (via macros defined in a set of header files) to the in-kernel library to implement run-time functionality (weaving/unweaving, pointcut matching etc.). In kernel mode, then, a special module supports the required run-time functionality for TOSKANA.

A programmer who wishes to use TOSKANA, however, does not have to use the relatively low-level macros provided by the TOSKANA headers. Rather, (s)he can use an adapted version of AspectC, implemented as a precompiler that generates regular ANSI C code augmented with TOSKANA macros, in order to develop aspect-oriented NetBSD extensions.

A typical AspectC-like implementation that implements part of an aspect to handle hardware network processing offloading is shown in listing 1.1. Here, two pointcuts are defined that capture sending and receiving of packets on the IP and TCP level and, depending on the configuration of the interface (which tells the advice code whether the interface supports features like hardware checksumming), calculates the checksum in software or hands a nonchecksummed packet to the network card for sending.

```
/* Device driver interaction for hardware offloading */
aspect hardware_offload {
  pointcut packet_from_to_tcp(struct tcpcb *tp):
     cflow(tcp_output)
     && calls(int tcp_build_datapkt(struct tcpcb *tp, ..);

  pointcut packet_from_to_mac(mbuf *m0):
     cflow(ip_output) && ( calls(int in_cksum(mbuf *m, ..))
                        || calls(int in_delayed_cksum(mbuf *m)) );

  before ( packet_from_to_ip ):
  {
    struct hw_netif *if;
    int status;
    uint32_t cksum;

    status = _get_current_hwif(if);  /* Get interface info */
    if (!status) {
       return ENOENT;
    }

    /* If hardware IP checksumming supported, send unchecked pkt */
    /* via function pointer registered by the driver */
    if (if->hw_offload & IP_CKSUM == IP_CKSUM) {
       cksum = 0;
       if->send_unchecked_packet(m);
    }
    /* otherwise, calculate checksum in software and send */
    /* checksummed packet out */
    else {
       cksum = ip_calculate_checksum(m);
       if->send_packet(m);
    }
  }

  around ( packet_from_to_tcp ):
  { /* ... */ }
}
```

Listing 1.1. A sample aspect in AspectC

This, in turn, is compiled by the TOSKANA precompiler into the ANSI C code shown in listing 1.2, with interspersed TOSKANA macros calling the in-kernel support library when the aspect has been deployed.

```
#include <sys/aspects.h>

ASPECT_NAME("hardware_offload");

void aspect_init(void) {
    PC *p1, *p2;
    p1 = POINTCUT("cflow(tcp_output)␣&&" \\
                  "calls(int␣tcp_build_datapkt(struct␣tcpcb␣*tp,␣..)");
    p2 = POINTCUT("cflow(ip_output)␣&&␣" \\
                  "(␣calls(int␣in_cksum(mbuf␣*m,␣..))" \\
                  "||␣calls(int␣in_delayed_cksum(mbuf␣*m))␣)");

    BEFORE(p1, hardware_offload_advice_1);
    AROUND(p2, hardware_offload_advice_2);
}

ADVICE hardware_offload_advice_1(struct tcpcb *tp) {
    ...
}

ADVICE hardware_offload_advice_2(mbuf *m) {
    struct hw_netif *if;
    int status;

    status = _get_current_hwif(if);  /* Get interface info */
}
```

Listing 1.2. A sample aspect with TOSKANA macros

The pointcut definitions are resolved at weaving time against the kernel symbol table, thus the compiler has no knowledge about whether a pointcut description is semantically valid or not. The task of performing validity checks is delegated to the weaver. This has the advantage that symbols in modules can also be used to construct pointcuts, with the tradeoff that the weaving process takes longer to execute since all referenced symbols have to be dereferenced at weaving time. As a consequence, pointcuts are not able to reference symbols in modules loaded past its own weaving time.

The C code generated by the AspectC compiler is then compiled into a loadable kernel module using the standard kernel methods provided by NetBSD. This module can then be loaded into the kernel manually using the *weave* command-line tool, which performs the kernel symbol lookups necessary to validate the pointcuts and, on success, uses the NetBSD kernel module loader to insert the aspect module into the kernel address space. The aspect is then automatically activated. However, separate functions in the in-kernel TOSKANA support library exist to activate and deactivate a loaded aspect.

The methods used to insert calls to advice into the kernel code are presented in the next section.

3.4 Aspect Semantics in TOSKANA

A taxonomy for joinpoints and pointcuts is presented in [10]. The aspect semantics described below is derived from the terminology introduced in that paper.

Deploying several aspects into the same kernel poses some interesting problems. If none of the pointcuts involved share any joinpoints, the weaving process for each of the aspects can occur independent of all other weaving. As soon as two pointcuts both match one joinpoint, both pieces of advice have to be executed.

TOSKANA currently handles this case in a strict first-come, first-served manner. A deployed aspect first executes its advice prior to all other aspects concerning the specific joinpoint that have been woven into the system at a later point in time. Thus, TOSKANA ensures a consistent execution order of aspects; however, support for specifying a particular execution order of aspects is currently not available.

Currently, no means are provided by TOSKANA to specify aspect precedence. Implementing precedence implies an analysis of all aspects already deployed in order to insert the new advice functions in the correct order at each joinpoint. This may involve code duplication in some cases, since a daisy-chained order of advice code for one joinpoint is possibly not the correct order for a different joinpoint. However, support for aspect precedence is planned for future versions of TOSKANA.

Since uninitialized pointers in binary code will crash the kernel, in order to deploy an aspect into the kernel, all references to external symbols inside the aspect's code must be referenced at aspect weave time. Thus, it is not possible to weave an aspect that references symbols contained in modules that are not loaded yet. Since TOSKANA aspects are implemented as loadable kernel modules, this implies that an aspect is not able to reference symbols of an aspect woven at a later point in time. This generally means that aspects should not reference other aspects' symbols if their weaving order cannot be predetermined.

4 Application Examples

The applications presented in this section cover several examples of crosscutting problems in kernel code: the first example illustrates cross-layer networking issues in wired networks, demonstrated by an improvement of a congestion control mechanism. The second example shows how TCP throughput for wireless networks can be optimized by exploiting cross-layer information. The third example, grid confinement, is an application for aspects that not only crosscuts all system call entry and exit points in the kernel, but in addition is a use case for the novel notion of process context joinpoints.

4.1 Cross-Layer Congestion Control

The TCP protocol uses two algorithms for increasing the congestion window. During steady state, TCP uses the congestion avoidance algorithm to linearly increase the value of the offered congestion window size, `cwnd`, shown in Fig. 2. At the beginning of a transfer, after a retransmission timeout or after a long idle period (in some implementations), TCP uses the slow start algorithm to increase `cwnd`, exponentially. According to RFC 2581, slow start bases the `cwnd` increase on the number of incoming acknowledgments. During congestion avoidance, RFC 2581 allows more latitude in increasing `cwnd`, but traditionally implementations have based the increase on the number of arriving ACKs.

"Appropriate byte counting" (ABC), defined in RFC 3465 [11], is a set of modifications to these algorithms to increase TCP's congestion window (`cwnd`) based on the number of bytes being acknowledged by each arriving ACK, rather than by the number of ACKs that arrive. The ABC algorithm improves performance by mitigating the impact of delayed ACKs on the growth of the window size. Implementing ABC improves performance of the network stack as well as security.

Simultaneously, the ABC algorithm provides window size growth in direct relation to the probed capacity of a network path, resulting in a more appropriate response to ACKs that only cover small amounts of data, i.e., ACKs of less than a full segment size, compared to the standard ACK counting method.

This more appropriate `cwnd` growth can improve both performance and can prevent inappropriate `cwnd` growth in response to a misbehaving receiver. In some cases, however, the modified `cwnd` growth algorithm causes larger bursts

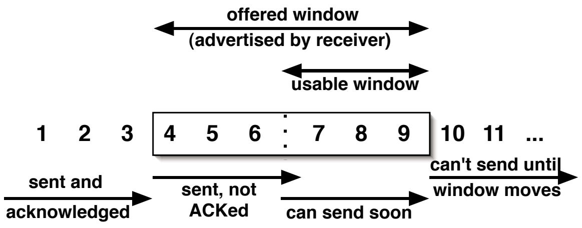


Fig. 2. TCP congestion window in operation

of segments to be sent into the network. This can, in extreme situations, lead to a nonnegligible increase in the drop rate and reduced performance.

Crosscutting Properties. To implement congestion window management, the original BSD TCP/IP stack implementation counts the number of incoming as well as outgoing packets. The counting takes place in the routines that handle TCP input and output and in functions that handle packet timeouts and buffer space management (e.g., in case of overflows). The packet counting is scattered over these functions and tangled with the respective code. A new implementation that implements ABC instead of simple packet counting affects the same locations. In detail, the implementation of ABC affects the following kernel source files:

– **src/sys/netinet/tcp_input.c** – 5 locations
  The function affected most is the tcp_input function itself, which handles the reception of TCP packets. Here, the behavior of the `cwnd` calculation is adapted.
  The other locations where the ABC implementation crosscuts the functionality set the variable `tp->snd_wacked` which counts the number of acknowledged packets, according to the number of correctly or incorrectly received packets.
– **src/sys/netinet/tcp_output.c** –1 location
  In this file, only the function `tcp_output` is affected. Here, at the end of the function, the variable `tp->snd_wacked` has to be set according to the number of ACK packets sent.
– **src/sys/netinet/tcp_subr.c** – 1 location
  Here, the `tcp_quench` function also has to keep count of the number of acknowledged packets.
– **src/sys/netinet/tcp_timer.c** – 6 locations
  In `tcp_timer.c`, the function `tcp_revert_congestion_state` resets the congestion state indicators after a predefined timeout. Here, statistics about the number of acknowledged packets are kept and the counter is reset after a timeout.

AOP Implementation. The main functionality is to appropriately adjust the byte count. Listing 1.3 shows the relevant excerpt from the advice function that calculates the window size:

```
/*
 * Open the congestion window.  When in slow-start,
 * open exponentially: maxseg per packet.  Otherwise,
 * open linearly: maxseg per window.
 */
if (tp->snd_cwnd <= tp->snd_ssthresh) {
    u_int abc_sslimit =
        (SEQ_LT(tp->snd_nxt, tp->snd_max) ?
         tp->t_maxseg : 2 * tp->t_maxseg);

    /* slow-start */
    tp->snd_cwnd += tcp_do_abc ?
        min(acked, abc_sslimit) : tp->t_maxseg;
} else {
    /* linear increase */
    tp->snd_wacked += tp->t_maxseg;
    if (tp->snd_wacked >= tp->snd_cwnd) {
        tp->snd_wacked -= tp->snd_cwnd;
        tp->snd_cwnd += tp->t_maxseg;
    }
}
tp->snd_cwnd = min(tp->snd_cwnd,
              TCP_MAXWIN << tp->snd_scale);
```

Listing 1.3. Advice code implementing ABC

The functionality most scattered throughout the source files, however, is the code to reset the variable counting the number of acknowledged packets. Resetting the `tp->snd_cwnd` counter is handled by a very simple advice function (listing 1.4):

```
ADVICE reset_wacked (struct tcpcb *tp) {
    tp->snd_wacked = 0;
}
```

Listing 1.4. Resetting snd_wacked

Since ABC counting can adversely affect the performance of TCP transfers, it is important that the ABC algorithm can be activated and deactivated on demand. This activation mechanism makes use of TOSKANA's dynamic aspects. In the current version, however, activation and deactivation has to be performed manually. A functionality that determines the appropriate time to switch the behavior would have to observe TCP input and output performance and activate the aspect accordingly. This would extend the pointcut descriptions used in this application to be dependent on the current TCP stack input queue context.

4.2 Optimizing TCP Throughput for Wireless Networks

The TCP/IP protocol suite, though ubiquitous, is not optimized for the use in wireless hop-to-hop networks, where an underlying broadcast medium is present. Here, the protection of the network from excessive load is required to ascertain a working system for all nodes. One way to enable this is to modify the congestion control provided by TCP/IP. Compared to the previous example, however, this approach is not universally applicable to hosts implementing TCP/IP. Rather, it is an optimization only useful on low-bandwidth shared broadcast media like IEEE 802.11-based WLANs. The end-to-end congestion control provided by TCP, therefore, is not an ideal protocol for wireless hop-to-hop networks. A more appropriate approach is to implement a hop-to-hop congestion control strategy, as described below.

The cooperative crosslayer congestion control (CXCC) approach, described in [12], uses implicit feedback from the next node along a route, obtained by overhearing this node's transmissions. Hence, it does not require additional overhead. The implicit feedback is combined with a tight queue length limitation at each node, which yields a congestion control method that not only avoids explicit feedback wherever possible, but additionally does not require any explicit congestion control action at all: the congestion control itself, too, happens implicitly, just by following the protocol's forwarding rules. As an immediate result, the algorithm is not only able to perform congestion control for connection-oriented, TCP-like traffic, but it works equally well for connectionless, UDP-like traffic and can easily be extended to other forms of communication such as multicasting, flooding and geocasting.

CXCC is not intended to replace the functionality of the layers it encompasses. Instead, the responsibilities of these layers and their interfaces are slightly changed. An overview of the altered architecture is shown in Fig. 3.

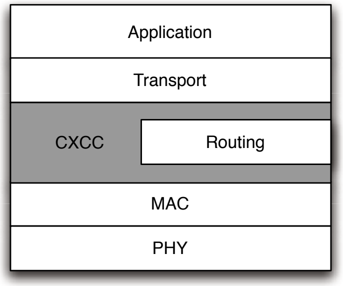


Fig. 3. Cross-layer properties of CXCC

CXCC alleviates network congestion by limiting the number of packets queued for each flow in each single node. This queue length constraint is not maintained by dropping packets exceeding the limit, but by pausing the transmission of a flow by an intermediate node once the limit has been reached by its successor on the path to the destination. In the basic variant of CXCC, only a single packet of each flow is allowed to be queued at each hop along the route. To decide whether a packet may be sent, a node has to know whether the downstream node still has a packet of the same flow in the queue. This information could be sent explicitly, but such a mechanism would induce additional traffic and increase the congestion of the network. In CXCC, the shared nature of the medium is used to gain the necessary information at no additional cost. Because every transmission is a defacto broadcast and the next hop generally is within communication range, the forwarding of a packet by the downstream node can be overheard. This indicates that the next packet may be transmitted.

At the same time, this implicit forwarding notification can serve as an acknowledgment, indicating successful packet delivery to the next hop. Common MAC layer acknowledgments are therefore no longer required. This yields an additional benefit: during the usual DATA–ACK sequence as used in 802.11, collisions may occur as well with the data packet transmission as with the ACK, therefore an area around both sender and receiver is affected by the transmission. Although the ACK packet is very small, it may well collide with and destroy any larger packet whose transmission is in progress. Eliminating the need for the transmission of an explicit ACK reduces the area of the network affected by the transmission.

Crosscutting Properties. CXCC acts as a kind of wrapper around the routing layer of the TCP/IP stack. Thus, it has to interfere with packet transport from the transport to the routing and from the routing to the MAC layer and handle the queue lengths and ACK generation accordingly.

The following files are affected by the implementation of CXCC:

– **src/sys/netinet/ip_input.c** – 3 locations
  The function `ip_input` handles IP input and defragmentation. Here, packets have to be sent to the CXCC implementation insted of directly to the transport layer.
– **src/sys/netinet/tcp_input.c** – 5 locations
  Here, the `tcp_input` function has to be modified in order to implement the modified ACK processing.
– **src/sys/netinet/tcp_output.c** – 6 locations
  In `tcp_output`, the function `tcp_output` generates ACKs. This generation behavior is changed.
– **src/sys/netinet/ip_state.c** – 4 locations
  Functions in `ip_state` handle TCP option processing and parts of the TCP timeout handling. Here, changes to functions `fr_tcpstate` and various window size handling functions have to be made.
– **src/sys/netinet/tcp_userreq.c** – 5 locations
  In `tcp_userreq`, user issued connection requests (e.g., termination) are handled. Here, ACK handling for forcibly closed connections has to be modified in several functions.

AOP Implementation. The “wrapped” implementation of CXCC is handled by one aspect consisting of several advice functions. These functions handle, in turn, redirection of traffic between the transport and routing layers as well as between the routing and MAC layers. ACK generation and surveillance of the broadcast medium for ACKs generated by remote receivers are realized.

Dynamic aspects are required to guarantee correct operation of the congestion control subsystem for hosts that maintain connections to wired and wireless hop-to-hop networks at the same time, like wireless access points. Depending on the destination of the network packet in question, CXCC congestion control must be either active or inactive. This can be achieved by activating and deactivating the aspect; otherwise, two mostly similar network stacks with different congestion control mechanisms would have to be employed.

```
/*
 * Check for ACK/NACK
 */
pointcut data_packet_received(int type, struct ackpkt *nmo):
    cflow(tcp_input)
    && calls(int tcp_recv(struct ackpkt *nmo, int type, ..));

before(data_packet_received):
{
   if (type == NPT_NACK) {       // This is a NACK
     set_header(nmo->getHead())->retry() = 0;
     // Reset retry count, so the real packet and not a RFA is sent
     nmo->stopTimer();        // Resend immediately
     sendPacket();
   } else {                              // This is an ACK
     nmo->shift();
     if (type == NPT_DATA) { // Impl. ACK via data pkt => upd. tmout
       influence_timeout(nh->queue_delay());
     } else if (type == NPT_RFA) { // Impl. ACK via RFA => upd. tmout
       influence_timeout(nh->queue_delay());
     }
   }
}
```

Listing 1.5. Advice code implementing ACK control

Listing 1.5 shows an excerpt from the handling of positive and negative acknowledgments. Here, the retry delay timer is stopped whenever a negative ACK for a packet arrived, causing the packet in question to be retransmitted as soon as possible. In case of a positive ACK, the timeout values for ACK handling are modified for two kinds of implicit ACKs occuring. This handling is an advice function activated whenever a joinpoint described by the pointcut `data_packet_received`, which matches all calls to the `tcp_recv` function to handle received packets inside the execution of `tcp_input`, is encountered.

4.3 Grid Application Confinement

In grid systems [13], unused computing resources available on the Internet worldwide are used to solve distributed computing problems. When deploying applications in an ad-hoc manner on systems participating in a grid [14], this implies running unknown and untrusted code on these machines with the potential danger of capturing spyware or other kinds of malicious software.

This software, however, can be sandboxed so that the process is running in a mode restricted by either a permit- or a deny-list of system calls and system call parameters. Using this method, an untrusted process can only read and write files and directories permitted by its sandboxing policies, regardless of Unix permissions or access control lists. In addition, access to other system resources, like process status, system load, disk space, etc., can be controlled.

Native Code in Ad Hoc Grid Applications. For Globus-based grid services [15] written in Java, the actions of grid applications can be sufficiently restricted by the JVM sandbox model [16]. Many applications, however, have to rely on legacy binary code for certain functionality or simply to improve performance. This legacy code is usually integrated into the application using shared libraries that communicate with the Java code using the Java native interface (JNI) [17]. This code executing natively on the CPU of the system running the grid application is not restricted by the Java sandbox and can, in turn, execute all operations that a regular native executable on the specific operating system can—including reading arbitrary files and producing buffer overflows in system libraries.

Several approaches to confining native code exist; these range from running the code in separate virtual machine instances and using small-scale virtualization solutions for kernel-level confinement to monitoring and restricting system calls. The approach of restricting system calls relative to a process context is implemented in the systrace package [18] which is available for various BSD-based operating systems and Linux. The structure of a systrace-based sandbox is shown in Fig. 4.

Normally, an application has access to any system call it requests. In Unix-based systems, nothing prevents a program from searching through "interesting" public readable files in users' home directories or opening a network connection that sends unsolicited emails to other hosts. Such behavior, of course, is unacceptable for a grid application; however, the simple solution of restricting the native parts of the application by denying file system and network access is too limiting, since native code may well read and write temporary files or connect over the internet in order to check for a valid run-time license.

To implement system call restrictions, policies have to be defined that tell the systrace facility which system call actions are permitted for a specific process. A systrace policy comprises a set of rules. Each rule controls a system call and its parameters, specifying whether or not this system call is allowed. A simple rule allowing the use of the "chdir()" and "stat()" system calls looks as follows:

```
chdir: permit
stat: permit
```

A rule containing the keyword "deny" would prevent these system calls to be executed. Rules can also control parameters of a system call:

```
fsread: filename eq "/tmp/gridjob" then permit
fsread: filename match "/etc/*" then deny[enoent]
```

Based on these rules, the program is allowed to read the file "/tmp/gridjob", but trying to read files that match the wildcard expression "/etc/*" (i.e., system configuration files) will result in an `ENOENT` error code returned instead.

One additional requirement for the use of systrace for ad hoc grid deployment is the ability to only activate its functionality for a specific shared object—the dynamically loaded native JNI-based code—in a process context. Systracing

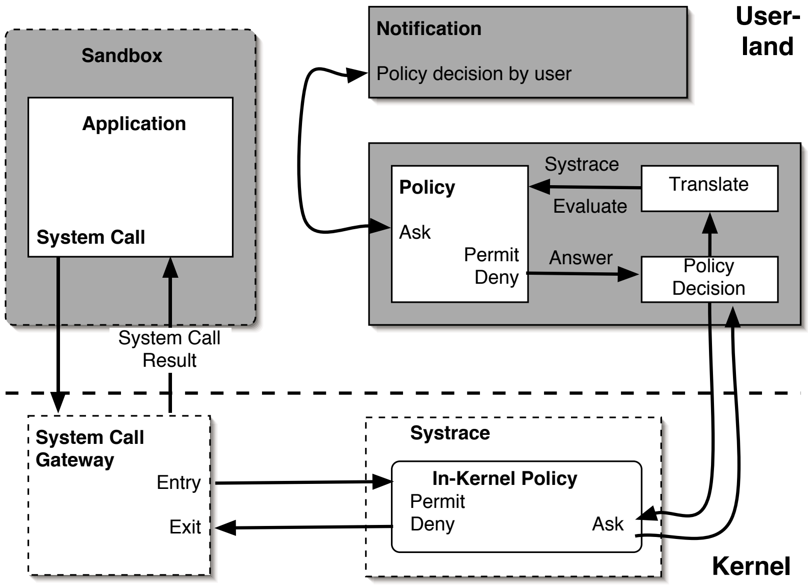


Fig. 4. Sandboxing in systrace

the complete Java VM environment including all loaded shared libraries would restrict the functionality of the already sandboxed Java code too far.

It must be mentioned that restricting shared libraries to execute only a well-defined number of system calls will not provide absolute security, since the policies provided may be incomplete so that “legal” system calls might be abused by native code to execute unwanted actions. However, this system is far easier to implement than virtualization-based solutions and, given a sufficiently restrictive set of rules, still provides a far greater level of security than running native library code unprotected.

Systrace Implementation. An AOP-based implementation of the systrace functionality can use one of several ways to insert control mechanisms. Looking at the path of a system call reveals several potential mechanisms. Applications initiate system calls by writing to specific registers and invoking software interrupts (in BSD on x86 systems, the “int” instruction). Applications usually do not execute system calls directly; they usually relay this task to the standard C library (libc), which handles setting up and initializing the calls. In fact, a large part of the C library simply consists of functions that provide a wrapper for system calls like *open(), read()* or *write()*.

The kernel layer is the suited place to intercept system calls, since it is the only place where it can be assured that every system call is caught, no matter which way it was invoked. Every system call enters the kernel via the `syscall` gateway, which acts as a kind of interrupt handler for the software interrupts the system calls use.

Upon invocation, the gateway reads a register (`eax` on x86 processors) to determine the requested system call number, which is an index into the system call table that contains pointers to individual kernel functions. The gateway parses that number and then initiates the correct function which performs the task specified by the system call. In order to reject a system call, systrace must intercept it before it is executed, so it hooks into the `syscall` gateway.

A part of systrace's functionality is implemented in a user space program which evaluates the policy files. It communicates with the kernel using the `/dev/systrace` device special file by reading kernel messages from the device and invoking `ioctl` calls in order to send messages to the kernel. The user space program is not relevant for this discussion, since the aspect-oriented implementation described below is concerned with the crosscutting problems in kernel mode.

Crosscutting Properties. The additional code required for implementing systrace consists of about 2,900 lines of context diffs that contain modifications to four source code files:

- **systemcalls.c** – Implementation of central system call code (3 locations)
  The function affected here is the system call dispatcher. The systrace helper function that checks for permissions and valid parameters is crosscutting the dispatcher functionality;
- **kern_exec.c** – Functions related to loading and starting programs from a file system (4 locations)
  Here, the function implementing the "exec" system call is affected. Various pieces of code implementing object loading for several binary formats (ELF, a.out) as well as code providing special treatment for interpreter-based executables like shell scripts, are crosscut. The systrace code fragments check for valid command line parameters, permission of program execution and access to files passed by file handles from the parent process.
- **kern_exit.c** – Functions related to terminated and finishing processes (3 locations)
  In this file, the function that implements the "exit" system call is crosscut. A process may exit due to normal or abnormal (forced) termination. Here, systrace functionality that calls a helper function to check return values and clear up outstanding signals as well as functionality that releases ruleset information related to this process is crosscutting the exit functionality.
- **kern_fork.c** – Functions related to process creation (2 locations)
  Finally, in `kern_fork.c`, the "fork" system call itself is affected. Fork is used to create an identical copy of the calling process. The systrace functionality crosscutting fork checks for valid permissions to execute fork at all, for restrictions on the open files passed via file descriptors from the parent and for possible restrictions on return values from fork.

AOP Implementation. The systrace functionality requires changes in `sys_exec`, `sys_exit` and `sys_fork` functions implementing the particular system call. In the system call dispatcher itself, the permission to call a specific system call at all is controlled. Since systrace functionality is different for every user-mode process, a dependency of the pointcuts on a dynamic process context is implicit—in this case, pointcut descriptions are actually controlled from the systrace user-mode components that parse and communicate the rules to the in-kernel components.

The interaction of systrace with the system call dispatcher is shown in listing 1.6. Execution of the system call itself is performed by the call to the corresponding entry in the system call table via a function pointer: `error = (*callp->sy_call)(p, args);`. If systrace is active, the support functions `systrace_enter` and `systrace_exit` are called before and after the system call execution itself:

```
        if ( (error == 0) && (ISSET(p->p_flag, P_SYSTRACE)) ) {
                error = systrace_enter(p, code, args, p->p_retval);

                if (error == 0)
                  error = (*callp->sy_call)(p, args);
        } else
          error = (*callp->sy_call)(p, args);

    /*
     * MP SAFE (we may or may not have the MP lock at this point)
     */
    have_mplock = userret(p, &frame, sticks, have_mplock);

        if (ISSET(p->p_flag, P_SYSTRACE)) {
                systrace_exit(p, code, args, p->p_retval, error);
        }
```

Listing 1.6. Systrace Functionality in `systemcalls.c`

In addition, three new files are introduced implementing the systrace core functionality, which provide the in-kernel routines for systrace as well as the interfacing to the user-mode components:

```
kern_systrace.c
systrace.h
tree.h
```

5 TOSKANA Implementation

In the following, selected implementation details of TOSKANA are presented. TOSKANA consists of a set of interacting user-mode programs and kernel modules. The central point of user interaction is the dynamic weaver, `/sbin/weave`. `Weave` is implemented as a daemon running permanently in the background. At startup, `weave` checks if the *aspectlib.ko* support library module is loaded and inserts it into the kernel if needed using `/sbin/kextload`. Then, the current kernel symbol table is read out of the ELF symbol table information in `/dev/ksyms`.

5.1 Modular Kernel Extensions

The NetBSD kernel can be extended at runtime through loadable modules. Modules are not restricted to implementing device drivers. Instead, a kernel module can provide any function that is possibly running in kernel context, since it has complete direct access to the system hardware as well as all kernel memory—i.e., all code and data structures.

Module support is implemented using a user-level dynamic kernel linker, `/sbin/ modload`, which verifies a module's object code and loads the code into kernel virtual memory space. After loading, the module is initialized by a call to a module-provided `init` function, which is responsible for setting up the module's local variables and checking conditions of kernel internals related to the module's functionality. TOSKANA uses the module functionality provided by NetBSD to load the code for advice functions into the running kernel.

5.2 Access to Kernel Internals

NetBSD provides access to information about the structure of the running kernel—kernel virtual memory and the symbol table—through two special devices, `/dev/kmem` and `/dev/ksyms`. A user-mode program can either access the symbol table using a set of `ioctls` or by directly interpreting the ELF symbol table that is available by `mmaping` `/dev/ksyms`, i.e., directly accessing the contents of the symbol table as a memory area in a user-mode application.[4]

Of interest for dynamic aspect weaving are the `STT_FUNC` and `STT_OBJECT` symbol entries provided by the ELF binary format which describe the virtual addresses of in-kernel functions and data objects (variables, arrays, etc.), respectively.

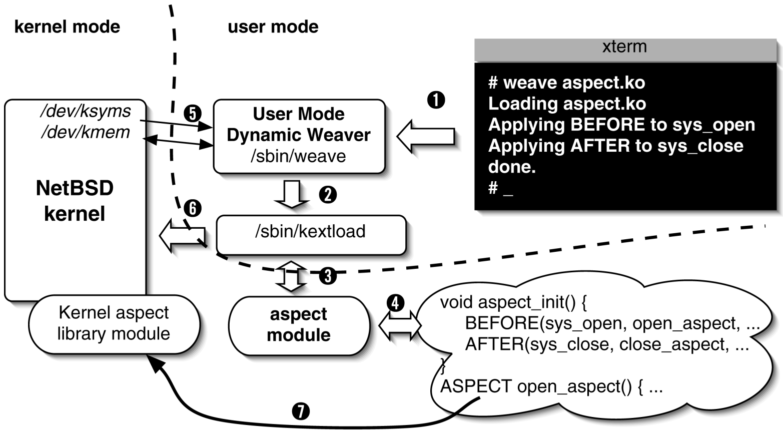
Experience with `/dev/ksyms` showed, however, that accessing the symbol information via the provided `ioctl` calls did not provide all available symbols in the binary. Access to all symbol information contained in the ELF binary was only available using the symbol table directly. The information required for dynamic aspect weaving is contained in `FUNC` entries that associate an address in kernel virtual memory space with the start of a function.

Aspect modules are implemented as kernel extension modules. They rely on functions provided by the support library module that instruct the weaver to modify kernel memory via `/dev/kmem` in order to deploy the aspect hooks.[5]

Figure 5 shows the components and tasks involved in weaving an aspect. Loading an aspect module is initiated from user mode (1) by invoking the `weave` command. The `weave` command then checks for availability and integrity of the aspect module (which is built like a standard kernel module, described earlier in this section) and extracts the symbolic pointcut and joinpoint information from the module binary. While manual loading and direct weaving of aspect modules from user mode is possible, normally aspect modules contain pointcut descriptions that specify additional dynamic contexts in their initialization functions so that the aspect is not woven in unconditionally.

[4] It is also possible—but more complicated—to directly obtain the symbol information from kernel virtual memory.

[5] This is a modification of kernel space memory. An alternative implementation would be to modify kernel virtual memory directly from the support library module. Using the `kmem` device, however, is less complex since the `kmem` device driver already takes care of setting page protection bits.



```
# weave aspect.ko
Loading aspect.ko
Applying BEFORE to sys_open
Applying AFTER to sys_close
done.
# _
```

```
void aspect_init() {
    BEFORE(sys_open, open_aspect, ...
    AFTER(sys_close, close_aspect, ...
}
ASPECT open_aspect() { ...
```

Fig. 5. Weaving a dynamic aspect into the kernel

The joinpoints are checked for validity against the kernel symbol information provided in `/dev/ksyms` (5). If the aspect module exists and contains valid weaving metainformation, the weaver instructs the NetBSD kernel module loader, `/sbin/kextload` to load the aspect module into kernel virtual memory (2). The module loader, then, treats the aspect module like normal (nondevice driver) kernel modules (4), opens it (3) and loads the module into kernel memory, executing the modules initialization function (6). The initialization function, in turn, calls a support library function that is contained in the kernel aspect library module for each advice (7)—implemented as a function in the same module—that is to be deployed.

Aspect weaving is performed by the NetBSD kernel support library. During kernel runtime, aspects can be activated and deactivated at run time from advice functions belonging to other aspects as well as from dedicated kernel components (e.g., the scheduler, which can activate and deactivate advice that depends on process context).

Name	Parameters	Function
activate	aspect, advice	activate advice in aspect
deactivate	aspect, advice	deactivate advice in aspect
weave	aspect	weave aspect
unweave	aspect	unweave aspect
proceed	various	proceed with parameters at joinpoint
check_symbol	symbol_name	check if symbol exists
get_symbol_value	symbol_name	get address of symbol
get_loaded_aspects	–	get list of loaded aspects

Fig. 6. Functions of the TOSKANA support library

The calls to the support library are implemented as macros that specify aspect weaving behavior. Currently, macros for `before`, `after` and `around` advice are provided. The aspect modules are compiled like standard NetBSD kernel modules. The only addition is the inclusion of the `#include <sys/aspects.h>` header file that provides the macro definitions to interface to the support library. The support library supplies some additional functions to aspect modules, listed in Fig. 6. The unweave operation currently works only when the aspect in question does not interfere with other aspects woven at the same joinpoint.

5.3 TOSKANA at the Code Level: Code Splicing

The basic method for inserting dynamic joinpoints shadows into native code is *code splicing*. Code splicing is a technology that replaces the bit patterns of instructions in native code with a branch to a location outside of the predefined code flow, where additional instructions followed by the originally replaced instruction and a jump back to the instruction after the splicing location are inserted.

TOSKANA offers support for fine-grained code splicing, which is able to insert instrumentation code with the granularity of a single machine instruction. While pointcut descriptions currently only support the specification of joinpoints on a function level, TOSKANA's underlying finer splicing granularity could be made available to the programmer in the future. As shown in Fig. 7, splicing replaces one or more machine instructions at a joinpoint with a jump instruction to the advice code with the effect that the advice code is executed before the replaced instruction.

Some precautions have to be taken, when modifying native code, to avoid corrupting activities currently running in kernel mode.

Execution of kernel functions can be interrupted at any time. It is therefore necessary to make the splicing operation atomic. On x86 processors, the *jump* instruction that redirects control flow to the dynamic aspect is at least five bytes long. Atomic writes on x86 can be handled in word or double word quantities (4 and 8 bytes, respectively) using the `FIST` or `CMPXCHG8` instructions. `FIST` is a Pentium instruction that was originally intended to convert floating point to integer operands. When applied to a memory location, it works atomically by stalling the floating point and integer pipe during its operation. `CMPXCHG8`, on the other hand, performs a compare and exchange operation on 64-bit quantities atomically.

These operations were originally introduced to support atomic operations or have atomicity as a side effect, but work as well for atomically exchanging instructions at joinpoints.

Special care, however, has to be taken in order not to corrupt the instructions following the jump instruction. Since x86 instructions can have arbitrary lengths ranging from 1 to 16 bytes, bytes that belong to instructions following the inserted jump to the advice code have to be replicated in the value that is being exchanged in order not to modify the binary representation of these instructions.

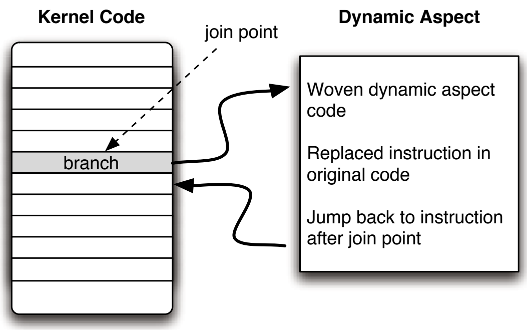

Kernel Code
join point
Dynamic Aspect
branch
Woven dynamic aspect code
Replaced instruction in original code
Jump back to instruction after join point

Fig. 7. Code splicing

Another problem of the splicing process is that potentially more than one instruction has to be replaced by the jump instruction and the second of these replaced instructions is a branch target. In this case, a branch to that address would execute the bit pattern that is part of the target address as operation code with unpredictable and most probably erroneous results.

This situation can only occur in functions written in assembly language, since only five bytes are being replaced. Fortunately, no such function could be found inside the x86-related and general parts of the NetBSD kernel so that the discussed issue does not affect TOSKANA's implementation.

Weaving. Code splicing is the basic functionality used by the weaver to insert joinpoint shadows. For each of the three possible advice variants, a different action must be undertaken in order to assure correct behavior of the dynamic aspect.

Weaving "before". Weaving *before* advice is the simple case. Code splicing is applied to the first instruction of the function involved, this instruction is replaced by a *jmp* instruction to the advice code.[6] At the end of the advice code, the instructions that were at the beginning of the original function the aspect was deployed in are inserted (see Fig. 8). The necessary space for these instructions is reserved by the `ASPECT` macro which inserts `nop` operations before the return of the advice code.

This method currently requires a separate copy of the advice code for each joinpoint shadow in question, since the address to return to after advice code execution differs for each joinpoint shadow at which the specific piece of advice can be executed. However, using this method is preferable to just calling the advice using subroutine jumps, since these use up potentially scarce in-kernel stack space and require additional execution time.

When multiple pointcuts resolve to identical joinpoints, the various pieces of advice code are executed in the order in which they were deployed. This is

[6] The splicing code takes care of padding the replaced code segment with `nop` if required.

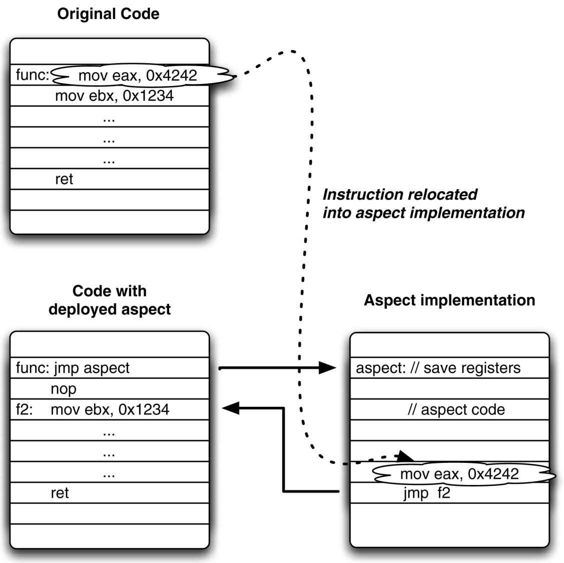


Fig. 8. Implementing “before”

handled by daisy-chaining the pieces together so that all exit points of the first advice function jump to the entry of the second piece of advice.

Weaving “after”. Weaving *after* advice is more difficult. While a *before* advice only has one possible entry point which is easily identifiable by processing the symbol table, the number of ways to return from a function and the location of the return instructions cannot be determined from the table. It is therefore necessary to search the memory area the function occupies for the pattern of the x86 *ret* instruction (`0xc3`), to find all possible return points of a function.

A further issue which has to be taken into account is that instructions on x86 machines can be of arbitrary length (from 1 to 16 bytes) and need not be aligned to word or longword boundaries. Thus, the weaver must take care not to accidentally identify a value of 0xc3 that is part of something else than a return instruction. To avoid these critical false positives, the weaver uses a table containing x86 instruction length information. Starting from the symbol table address entry of the function, the weaver only searches the first byte of possible instructions for the value 0xc3.

Weaving “around”. The weaving of *around* advice basically combines the methods used for weaving *before* and *after* advice. Beyond the issues described above, an additional problem must be dealt with by the implementation of the *proceed* macro, since each jump instruction to the original function has to be replaced by a jump instruction to the specific function matches by the current joinpoint. The address of this function cannot be determined at weaving time, since the advice code may contain more than one *proceed* macro.

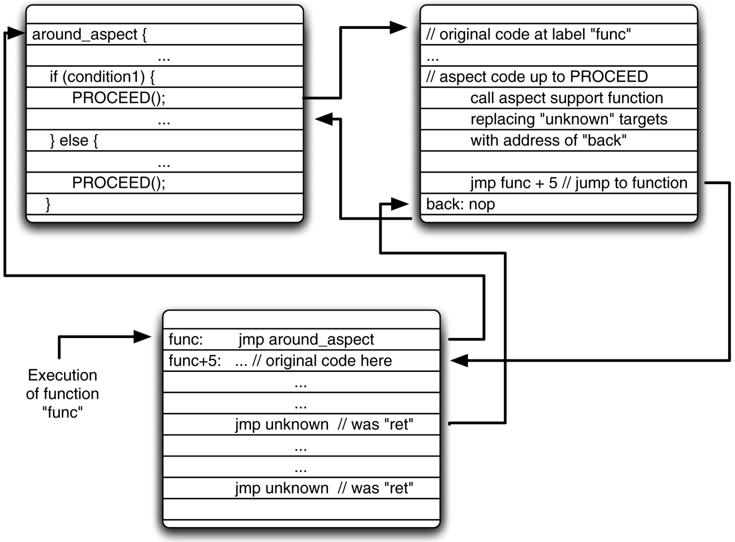

around_aspect {
...
if (condition1) {
PROCEED();
...
} else {
...
PROCEED();
}
// original code at label "func"
...
// aspect code up to PROCEED
call aspect support function
replacing "unknown" targets
with address of "back"
jmp func + 5 // jump to function
back: nop
Execution of function "func"
func: jmp around_aspect
func+5: ... // original code here
...
...
jmp unknown // was "ret"
...
...
jmp unknown // was "ret"

Fig. 9. Implementing "proceed"

As shown in Fig. 9, the code flow gets redirected dynamically. While the initial call of the function works just like a *before* advice, handling the return case changes depending on the dynamic execution of the *proceed* statements contained in the advice code. Initially, all returns from the function are simply replaced by a jump to a return instruction which is the default in case no *proceed* macro at all is used in the advice code. Whenever the advice code reaches a *proceed* macro, the macro calls a function in the aspect support library that replaces the target addresses in all positions where a return instruction was originally located with the address of the instruction directly after the `jmp` instruction to the function code itself. Since threads executing in the kernel in parallel are in danger of returning to an incorrect address providing a jump to a different function, the return addresses have to be modified according to the in-kernel context. This, like activating and deactivating aspects, occurs at scheduling time.

This approach to implement proceed currently has some limitations. The case in which advice code calls the advice function itself recursively is not supported, since this would require maintaining a stack of advice invocations. A similar problem shows up when multiple around advices are defined for a single joinpoint. Here, ensuring that a return from the advised function called by proceed continues execution at the correct location in code would require additional administrative overhead.

5.4 Dynamic Aspects in a Process Context

To provide aspects local to a process context, the activation and deactivation of the aspect has to occur in the scheduler. Since process context dependency is important for some applications, an implementation of context dependency is required. This allows pointcuts to be dynamically selected based on the user-mode process context that is currently active.

The deployment implementation is located in `kern/kern_sync.c` and extends the function `awaken` with the code in lines 16. . . 27 in listing 1.7. Here, the current thread control block is checked for deployed aspects. If one or more aspects are active for the current thread, these are in turn activated using the `toskana_activate` function from TOSKANA's in-kernel support library.

```
 1  __inline void awaken(struct lwp *l) {
 2      SCHED_ASSERT_LOCKED();
 3
 4      if (l->l_proc->p_sa)
 5              sa_awaken(l);
 6
 7      if (l->l_slptime > 1)
 8              updatepri(l);
 9      l->l_slptime = 0;
10      l->l_stat = LSRUN;
11      l->l_proc->p_nrlwps++;
12
13      /* Activate dynamic aspects for current process */
14      if (l->l_proc->numaspects > 0) {
15         toskana_aspect *current = *(l->l_proc->aspects);
16         int count;
17
18         for (count = 0; count < l->l_proc->numaspects; count++) {
19             toskana_activate(current);
20             current++;
21         }
22      }
23      /* Since curpriority is a user priority, p->p_priority is
24       * always better than curpriority on the last CPU on which
25       * it ran */
26      if (l->l_flag & L_INMEM) {
27              setrunqueue(l);
28              KASSERT(l->l_cpu != NULL);
29              need_resched(l->l_cpu);
30      } else
31              sched_wakeup(&proc0);
32  }
```

Listing 1.7. Activating dynamic aspects in the scheduler

This implementation is currently unoptimized with respect to efficient activation and deactivation of aspects. The overhead involved currently scales linearly with the number of aspects to be deactivated from the previous process context and to be activated for the new process context. Here, an optimization using patterns of instructions to modify—similar to a just-in-time compiled dynamic weaver—or a kernel modification that activates and deactivates in-kernel aspects using duplicate pages with and without the woven aspect would greatly enhance performance.

Activation in the context of the scheduler may seem unsafe at first, but the switching actually is safe, since the timer interrupt occurs at the highest priority compared to other interrupts. Thus, no interrupt bottom-half code is being executed while activating or deactivating aspects. The only problem is the increased time required if a large set of aspects has to be activated or deactivated during the context switch. However, this approach currently does not support multiprocessor configurations.

6 Performance Results

In this section, the performance overhead associated with TOSKANA is evaluated. The measurement method used to obtain performance results is discussed and a set of measurements of TOSKANA's timing behavior is presented.

The measurement results presented here concentrate on micromeasurements of various important parts of TOSKANA's functionality, thus giving an overview of the performance of the overhead introduced by code splicing, weaving as well as activating and deactivating aspects. Macromeasurements could also be performed using the measurement technology described below; however, macromeasurements are not rerally useful for a number of reasons. The applications presented in Sect. 4 that are implemented using TOSKANA draw their improvements, with respect to performance, not from using aspect-oriented technology, but rather from the application itself. This is the case for CXCC as well as for the ABC implementation. The third example presented, grid application confinement, is a security topic. Performance results on a macroscopic scale would rather measure the properties of the systrace implementation used.

6.1 Evaluation Method

Evaluating the performance characteristics of changes to kernel code is a complex task—on the one hand, measuring the impact of a single modification undisturbed by other tasks running in parallel is difficult; on the other hand, getting reliable measurements out of a system running on the bare hardware poses a completely new set of problems.

The measurement method selected for evaluating TOSKANA was chosen to be as close to measuring effects on real hardware as possible. Since no hardware-based CPU emulator was available for the measurements and introspection utilities like Sun's DTrace [19] or KProbes [20], which would be well suited for measuring kernel timing, are currently not available for NetBSD, an approach using the Xen virtual machine [21] was used. For evaluation purposes, NetBSD is running as an unprivileged guest domain on the Xen hypervisor. For this purpose, a modified version of the NetBSD kernel suitable for running on top of Xen was implemented and the entire system was installed on a 2 GHz Pentium M laptop. The measurements presented below use the remote gdb (GNU debugger) infrastructure provided by Xen. A remote gdb server running in the privileged domain 0 (in this case, a Linux system) is configured to control the NetBSD system running in an unprivileged domU domain. Since Xen has full control over interrupt assignments and memory mappings, the remote gdb server can insert breakpoints at arbitary locations in the execution of any code executing in the domU domain, including the kernel.

The system provided by Xen provides two methods to measure performance. On the one hand, a microsecond-resolution timer is provided in each execution domain that is incremented only when Xen schedules the respective domain to run. On the other hand, the virtual CPU model provided by Xen implements a cycle counter that also only counts the instructions executed in the context of this particular domain. Both values were used for evaluation and, in addition, compared to each other in order to ascertain the validity of the values measured. The implementation of counters through hardware features of the CPU, together with the domain multiplexing provided by Xen, ensures that the results gathered are as similar as possible to a system using NetBSD running on the bare hardware. Compared to a full-system emulation like qemu [22] which only provides a behavioral model of a system but not accurate timing emulation, running the code multiplexed on the real CPU ensures that all measurements are performed in a cycle-accurate way.

6.2 Performance

To evaluate the effects of using TOSKANA in a NetBSD system, a set of measurements was performed. Of special interest for the evaluation was the infrastructural overhead for deploying and using aspects, the cost of saving and restoring state modified by advice code and a simple example to ascertain the results of the previous measurements.

6.2.1 Basic Values

In order to obtain a reference, the basic duration of two specific system calls were measured. The first system call is `sys_getuid`, a call that simply returns the user ID related to the calling process; the second system call is `sys_chmod`, which is used to change the access mode of inodes (files and directories) in a file system. To avoid the influence of I/O operations on the results, all operations were performed on a RAM disk. All measurements were performed in single user mode 100 times to eliminate the influence of processes running in parallel as well as to filter out inaccuracies.

Figure 10 shows the distribution of 100 successive calls to the `sys_getuid` function.[7] In the same diagram, timing values for the `sys_mknod` function are given. From the diagrams we can see that while the average time for executing `sys_getuid` is 1.996 μs, the file system access overhead brought the average `sys_chmod` execution time up to 3.610 μs.

6.2.2 Overhead of Advice Invocation

The next interesting question is what the overhead of invoking the deployed advice code is. Here, calls to an empty advice function are inserted in the code flow of the two system calls. The number of inserted advice calls was increased from 1 to 100; each of these runs was again averaged over 100 iterations. The results, depicted in Fig. 11, show a linear increase in system call execution time for both system calls up to about 50 advice calls per system call invocation. While

[7] Like all other measurements, these results were taken out of the middle of a larger series in order to eliminate differences due to caching.

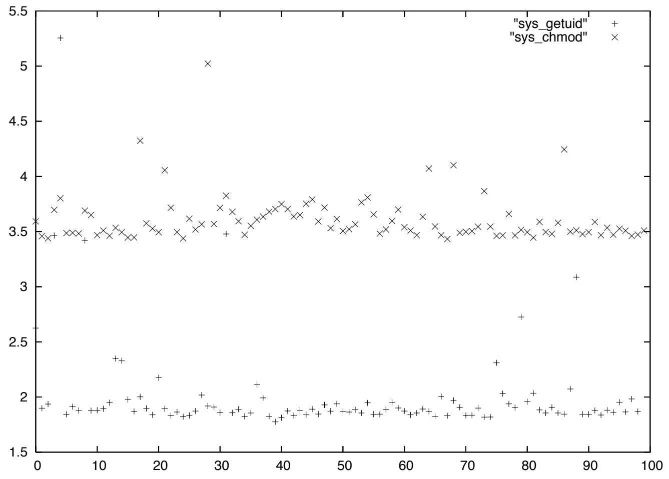


Fig. 10. Duration (microsecond) of the sys_getuid and sys_chmod system calls over 100 iterations

the `sys_getuid` call continues to scale in a linear way, the values for `sys_chmod` start growing faster than linear. We attribute these results due to caching effects in the processor; a more precise evaluation of this effect would require a precise CPU model to analyze cache usage and bus contention. The average overhead for calling a single advice function over all samples was 13.99 ns. The time required for a single invocation varied from 8 to 23 ns; again, this is most likely the effect of caching and different behavior in the CPU's branch prediction.

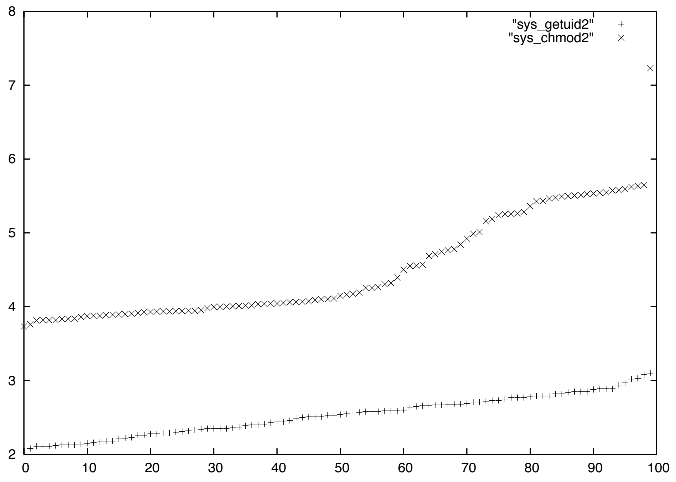


Fig. 11. Timing (microsecond) of the sys_getuid and sys_chmod system calls with 1–100 joinpoints

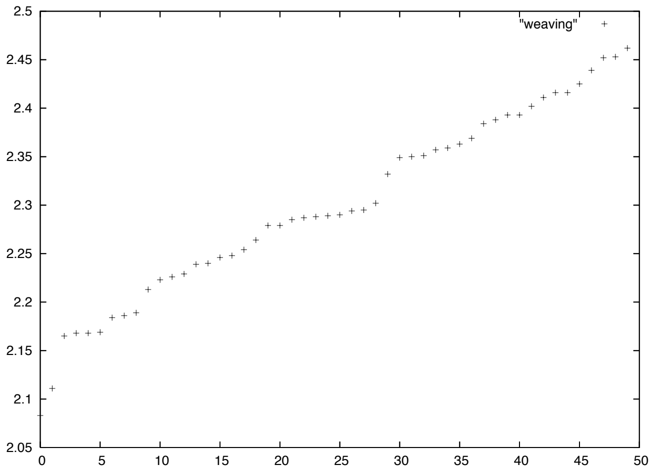


Fig. 12. Time (second) to deploy an aspect relative to the number of symbols involved

6.2.3 Overhead of Simple Operations

For this measurement, a simple operation (increasing a variable and referencing a local variable) was used, thereby evaluating the overhead of saving context for an advice invocation while providing only minimal execution overhead. The results are similar to the previous measurements;[8] the average time for an advice invocation grew from 13.99 to 17.23 ns with samples ranging from 12 to 29 ns. Thus, the insertion of a simple instruction does not have significant effects on timing; more extreme caching effects will probably only show up with very large advice code functions.

6.2.4 Deployment Overhead

This measurement evaluates the time required to dynamically insert a new aspect into the running system. Measurements were performed with respect to the number of symbols that had to be referenced in a pointcut, since symbol lookup has the most significant on TOSKANA's weaving performance. Figure 12 shows the time it takes to weave an aspect that references from 1 to 50 symbols in the kernel. Since weaving is initiated from user mode, reading the kernel's symbol table initially is the most time-consuming step. In current versions of TOSKANA, the symbol table is not cached, since no mechanism exists with which the kernel informs the weaver about recently loaded kernel modules.[9] This results in an overhead of about 2 s to extract the symbols using the `/dev/ksyms` special device. However, this lookup is only required the first time an aspect is used in the system. After extracting the symbol table, the time required to weave referenced symbols grows linear with the number of symbols.

[8] The diagram is not shown since no significant differences can be observed from the diagram.

[9] Here, this is only relevant modules other than the aspect modules.

7 Related Work

Related research efforts can roughly be classified into research in program instrumentation, research in using aspect-orientation in an operating system context, dynamic aspect research, and AOP compiler tools for languages used in operating system development like C and C++.

7.1 Instrumentation Tools

Kernel instrumentation is a technology that aims to perform run-time system analysis, including performance measurement, debugging, code coverage, run-time installation of patches and run-time optimizations. Static kernel instrumentation incurs a high overhead, since instrumentation points are statically compiled into the existing code, using up compute time for checks if a given instrumentation point is actually activated. Dynamic kernel instrumentation, however, can implement instrumentation points at run time into an existing kernel binary, thus reducing the overhead and providing a greater level of flexibility.

KernInst. KernInst [23] is a tool that enables a user to dynamically instrument an already-running unmodified Solaris operating system kernel in a fine-grained manner. It has been successfully applied to kernel performance measurement and run-time optimization.

The basic principles used by KernInst to dynamically insert instrumentation points into running native code are similar to the approaches TOSKANA uses. However, KernInst is not suited as a basis for dynamic aspect deployment. The space available for code patches and associated data is very limited in KernInst, so the deployment of a large number of aspects is not feasible.

Dyninst. Dyninst [24] is a runtime program manipulation tool which provides a C++ class library for program instrumentation. Using this library, it is possible to instrument and modify an application program during execution. The next time the modified program executes the block of code that has been modified, the new code is executed in addition to the original code to either augment the existing program with ancillary operations or to alter the semantics of the program by changing the subroutines executed or manipulating application data structures. A unique feature of Dyninst is that it permits machine-independent binary instrumentation programs to be written. Many features of Dyninst are similar to the approaches used by TOSKANA. However, Dyninst is currently only available for user-mode code.

Sun DTrace. A recent development extending the ideas of KernInst is DTrace from Sun Microsystems [19]. Like KernInst, DTrace is able to dynamically insert instrumentation code into a running Solaris OS kernel. Unlike other solutions for dynamic instrumentation that execute native instrumentation code, DTrace implements a simple virtual machine in kernel space that interprets byte code generated by a compiler for the "D" language, which is an extension of C specifically developed for writing instrumentation code.

D was especially designed to make instrumenting the Solaris kernel safe in all circumstances. To avoid the danger that a piece of instrumentation code creates loops that hang-up kernel operation, only forward branches are permitted by the virtual machine. As a result, the functionality of D programs is relatively restricted. While this in principle provides a lot of security when dynamically inserting code into random spots in the kernel, the execution model provided by DTrace is too restricted to implement general advice code. In addition, D does not provide an instrumentation similar to an "around" advice. Overall, DTrace contains too many tradeoffs to be used in a more general implementation of dynamic AOP like TOSKANA. An open-source implementation of DTrace in the forthcoming OpenSolaris 10 will nevertheless be interesting to analyze.

7.2 AOP Compiler Tools

AspectC. AspectC was proposed as an extension of the ANSI C language by Coady et al. [25, 26], but no implementation is available so far. In addition, the language has no formal definition so far, but it was rather described using example code, which, unfortunately, is inconsistent between publications on AspectC.

AspectC, however, provides the basic idea for using aspect-orientation in kernel mode. The lack of a readily available AspectC compiler resulted in the development our the AspectC prototype [3].

AspectC++. AspectC++ [27] is an extension of the C++ language that supports static joinpoints as well as dynamic joinpoints modeled after the AspectJ [28] approach. While static joinpoints are named entities in the static program structure, dynamic joinpoints are events that happen during the program execution.

AspectC++ is used in research and commercial embedded operating system projects like the PURE [29, 30] embedded real-time kernels. In the context of TOSKANA, however, AspectC++ did not prove to be usable due to the compiler generating C++ classes from the AspectC++ source code, which requires in-kernel runtime support.

xtc. xtc [31] is a program transformation toolkit similar to the CIL toolkit used by the AspectC compiler. Currently, xtc is used by the Princeton PlanetLab project to manage large-scale modifications to the Linux kernel source code [32]. Xtc could be used as an implementation basis for a TOSKANA-based dynamic AspectC compiler implementation.

XWeaver. XWeaver [33] is an aspect weaver for C/C++. The aspects to be processed by XWeaver are written in the AspectX language.

XWeaver is an extensible, customizable and minimally intrusive aspect weaver. It is extensible in the sense that the aspect weaving process is controlled by a set of rules which are independent of each other and are each implemented in a dedicated module. The weaver can be extended by adding new modules that cover new rules to handle additional types of aspects. Similarly, the weaver is customizable because existing rules can be tailored to the needs of a particular project by updating the corresponding rule module.

XWeaver, however, uses aspect specifications in a special XML format that break the continuity of the C code the aspects are to be woven into. In order to gain acceptance for kernel development, a solution extending C in a more "natural" way has been preferred for TOSKANA.

7.3 Extensible Operating Systems

An extensible operating system lets an application safely tailor the operating system's behavior to the needs of the application. How to best build a system that explicitly allows applications to insert code into the kernel has been an active research topic for many years, with several different methodologies explored.

SUPERZAP. SUPERZAP (or IMASPZAP) (http://www.cs.niu.edu/csci/567/ho/ho4.shtml) is the IBM supplied utility for applying patches to object modules. SUPERZAP allows to inspect and modify instructions in load modules (without leaving any audit trail). With some positive thinking, SUPERZAP may be viewed as a tool for extending operating systems by machine code patching techniques. However, even though some of the basic technological approaches SUPERZAP uses are similar to the approaches TOSKANA uses, its main application is supplying patches to a live operating system. While this is also possible using TOSKANA, it is rather a side effect of the code manipulation technology employed and not a central feature.

SPIN. SPIN [34] is an operating system that allows applications to dynamically insert low-level system services, the so-called spindles, into the kernel, such that they can receive notification of, or take action on, certain kernel events. Spindles, as well as SPIN itself, are written in Modula-3, and the compiler is the only user-mode process that is allowed to insert spindles at run time into the kernel (by performing compilation at run time). This has the drawback that the time to compile and optimize a spindle has to be spent for each spindle insertion; dynamically linking a spindle into the running kernel also takes time. Furthermore, there may be semantic difficulties when multiple handlers are installed on a single event. While SPIN is an interesting approach that could possibly be used as a basic technology for a TOSKANA-like aspect system, it never gained widespread acceptance. One advantage of SPIN is that—due to the use of Modula-3 as its implementation language—many of the problems TOSKANA has to handle in C code are nonexistent, so SPIN can actually provide a more secure approach to extending operating systems. However, to our knowledge, manipulation of already existing pieces of functionality in SPIN is quite hard, whereas it is one of the common usage scenarios for TOSKANA.

VINO. The Vino operating system [35] allows applications to dynamically insert code that replaces the implementation of a method on any object. Alternatively, code can be installed to run when a specified kernel event (such as a network packet arrival) occurs. The granularity at which the system can be extended is dependent upon the objects and events exported by the system. The complexity and performance overheads associated with Vino prevent it from being as efficient as SPIN, for example. Also, like SPIN, Vino does not include any features for dynamic system reconfiguration.

KEA. The Kea operating system [36] provides means through which applications can dynamically reconfigure, replace or extend existing services in an incremental and safe manner. Kea incorporates several features, the so-called portals and portal remappings, whereby a system can be easily reconfigured. Kea provides a mechanism (called IDC) which has all the semantics of a procedure call, but lets a calling thread continue execution in a different domain. Portals are the Kea abstraction used to represent an IDC entry point. They provide a handle into both a domain and an entry point within that domain, and once created they can be freely passed around between domains. The most important property of portals, and the one that makes reconfiguration possible, is that they can be dynamically remapped to a different domain/entry-point at runtime. Compared to TOSKANA, KEA only offers an incremental model to extend kernel functionality. This requires reimplementing complete kernel subsystems, whereas using TOSKANA, relatively small changes, e.g., to improve the performance of certain code paths, can be implemented directly into an existing subsystem structure.

SLIC. SLIC [37] allows to insert trusted extension code into existing operating systems with minor or no modifications to operating system source code. Conceptually, SLIC dynamically "hijacks" various kernel interfaces (such as the system call, signal, or virtual memory paging interfaces) and transparently reroutes events which cross that interface to extensions located either in the kernel (for performance) or at the user-level (for ease of development). Extensions both use and implement the intercepted kernel interface, enabling new functionality to be added to the system while the underlying kernel and existing application binaries remain oblivious to those extensions. SLIC dynamically interposes extensions on kernel interfaces by modifying jump tables or by binary patching kernel routines. While the basic approaches of SLIC are comparable to the splicing approach used by TOSKANA, the hijacking can only occur to kernel interfaces, not to low-level functions inside the kernel.

GILK. GILK [38] is a dynamic instrumentation toolkit for the Linux kernel. It allows a standard Linux kernel to be modified during execution using instruments implemented as kernel modules. The GILK authors face many problems similar to the ones that occurred during the design and development of TOSKANA, e.g., the variable-length instructions of the x86 architecture when replacing instructions in binary code. While the basic technological approaches of GILK are quite similar to the technologies TOSKANA uses, GILK is restricted to developing instrumentation code for Linux, whereas TOSKANA can be applied to cover a broader range of in-kernel applications.

7.4 AOP in Kernel Space

Since aspect orientation is well suited to work on large software systems, research on using AOP in kernel space has been the topic of some projects. The existing developments, however, concentrate on providing static AOP functionality mostly using AspectC or AspectC++ as a basis.

a-kernel. Research on aspect orientation in operating system kernels was initiated by Coady et al. [25], where problems that crosscut the traditional layered operating system structure were identified using FreeBSD as an example.

Based on AspectC, the a-kernel project tries to determine whether AOP is a suitable method to improve OS modularity and thereby reduce complexity and fragility that is associated with the implementation of an operating system.

In Coady et al.'s work [26, 39, 40, 41], various crosscutting concerns are implemented as static aspects using AspectC in the FreeBSD kernel. In addition, an analysis of code evolution implementing crosscutting concerns between different versions of FreeBSD is undertaken and the evolution is remodeled as static aspects.

Further development in this project resulted in the concept for RADAR [42], which is a proposal for a low-level infrastructure for system analysis using dynamic aspects in OS code. This is an interesting development similar to TOSKANA, although the application area is targeted to analysis. Currently, no experience with implementing the system seems to exist.

CiAO. Spinczyk et al. [29] and Gal et al. [30] describe the design and implementation of an aspect-oriented family of operating systems which provide, based on aspects and static configuration, a full encapsulation of architectural properties of an OS. CiAO is targeted at the area of embedded systems, ranging from deeply embedded devices up to Unix-like operating systems, requiring a so far unattained level of adaptability and configurability. The language used for developing operating systems of the CiAO family is AspectC++.

CiAO, however, is targeted to embedded systems. A general use in operating systems may be beyond the scope of that system. In addition, the use of AspectC++ is only possible in systems newly developed in an object-oriented style. The operating systems this paper is concerned with are, however, mostly written in procedural C, so the CiAO approaches are not applicable here.

7.5 Dynamic AOP

There are several approaches for implementing dynamic aspect orientation. Of interest for TOSKANA are solutions that work on C/C++ code as well as solutions that operate in the execution layer.

μDyner. μDyner [43] is a dynamic AOP system for user programs written in the C language. When a source file is compiled, it inserts hook code at every joinpoint marked as hookable. The hook code examines whether or not the joinpoint is selected by a pointcut during runtime and, if it is selected, the hook code executes the associated advice. μDiner provides a special *hookable* source-level annotation with which the developer of the base program annotates points in the program at which later on dynamic adaptation is permitted. Here, only functions and global variables can be declared to be hookable. If the number of hookable joinpoints is large, the overheads due to hook code handling are noticeable.

Arachne. Arachne [43] is a dynamic AOP system for writing and dynamically deploying aspects into running user-mode C applications without disturbing their service. It is built around two tools, an aspect compiler and a runtime weaver. The former translates the aspect source code into a compiled library that directs the latter to place the hooks at weaving time into the base program. The hooking mechanisms used in Arachne are based on improved techniques originally developed for μDyner (see above), which allow to rewrite the binary code of executable files without pausing the base program.

The aspect code is written using a special extension of C that provides the syntax for specification of joinpoints and advice types. Aspects are compiled into dynamically loadable libraries that can be woven and unwoven using the Arachne *weave* and *deweave* tools. Although it also uses the symbol information produced by a normal compiler, the pointcut designators of Arachne cover not only function calls but also accesses to global variables and memory blocks allocated by malloc. However, performance penalties are associated with pointcutting accesses to memory blocks, since Arachne uses a page fault for detecting accesses to a memory block.

Arachne is an interesting approach, but its use is restricted to weaving aspects in user-mode programs. Since these can be easily controlled by a running kernel without changing state, Arachne would require extensive modifications to be used in kernel code.

TinyC^2. TinyC^2 [44] is a dynamic AOP approach for userland C programs which is based on the idea of implementing an aspect-oriented language on top of an existing system instrumentation technique [45]. The TinyC^2 language is designed as an extension of the C language with new language constructs to enable the composition of aspect programs. The compiler of TinyC^2 is essentially a source-to-source translator that translates C statements to the API instructions of the target instrumentation tool, which currently is the Dyninst run-time instrumentation platform [24]. Due to the runtime instrumentation nature of Dyninst, TinyC^2 can be treated as the runtime weaving aspect language.

Since TinyC^2 uses the symbol information produced by a normal C compiler, it supports pointcuts for selecting function execution as joinpoints. Again, the use of TinyC^2 is restricted to user mode C programs, and thus it cannot easily be applied to kernel code.

DAC++. DAC++ [46] is a new C++ compiler that supports dynamic aspect weaving and unweaving. To weave an aspect, DAC++ modifies the compiled binary of a C++ program during run time. DAC++ supports per application, per thread, per class and per instance weaving. In addition, it controls the weaving process so that it does not impact the overall system availability. Since DAC++

uses the symbol information produced by a normal C++ compiler, it supports pointcuts for selecting function execution as joinpoints. Redefining the pointcut designator for an aspect at runtime is not allowed since it needs extra space at run time and it slows down the weaving process.

Steamloom. The Steamloom virtual machine [47] is an extension to an existing Java virtual machine. It follows the approach to tightly integrate support for AOP with the virtual machine implementation itself, instead of delivering a set of Java classes providing such support. Steamloom comprises changes to the underlying VM's object model and just-in-time compiler architecture. Its approach to dynamic weaving is to dynamically modify method bytecodes and to schedule those methods for JIT recompilation afterwards.

Steamloom was built to evaluate the benefits that can be gained by implementing AOP support in the execution layer instead of at application level. Various measurements [48] have shown that both flexibility and performance significantly benefit from such an integrative approach. Using Steamloom for dynamic aspects in kernel code, however, would require a kernel written in Java. Some approaches to create a Java-based operating system exist, but none of these has shown significant success.

8 Conclusions

This paper presented an approach to deploy dynamic aspects into the native code of a running NetBSD operating system kernel, based on an analysis of crosscutting concerns related to the structure and C-based implementation of kernel code. The proposed TOSKANA toolkit provides a novel set of tools, macros and libraries for developing and deploying dynamic aspects in the kernel space of NetBSD. It provides *before*, *after* and *around* advice for in-kernel functions and supports the specification of pointcuts as well as the implementation of aspects themselves as dynamically exchangeable kernel modules.

The use of TOSKANA was demonstrated by several examples indicating the crosscutting nature of providing cross-layer and context-dependent functionality in an operating system kernel. Performance results were presented to characterize the aspect deployment overhead incurred by using TOSKANA.

While earlier efforts provided ideas and implementations for partial problems like instrumentation, dynamic aspects or using AOP in kernel space, TOSKANA is, to our knowledge, the first solution that implements dynamic aspects in kernel space based on extending dynamic instrumentation technologies adapted to the requirements of AOP.

There are several areas for future research. While TOSKANA itself is a working prototype, stability and usability improvements are still necessary. In order to ease the development of aspect code, especially an integration with an AspectC compiler system is required.

The principles exemplified in the NetBSD-based approach are generally usable in other operating systems and in systems not based on the x86 processor architecture. Here, future research aims at integrating the dynamic aspect approach implemented in TOSKANA with a system structure that no longer uses a monolithic kernel-like NetBSD, but rather a microkernel-based structure with operating system personalities implemented as user-mode processes on top of the microkernel [49]. Aspect interaction could be handled by a separate, microkernel-based process. Since the microkernel has complete control over memory and CPU resources consumed by a kernel personality running on top of it, the complexity of implementing TOSKANA could even be reduced.

The development of kernel code can be further simplified by identifying crosscutting concerns that can be refactored into aspects. This so-called "aspect mining" is one of the requirements for the widespread adoption of AOP, especially when maintaining existing large software systems. Current aspect mining approaches require either the help of a human expert [50, 51] or are restricted to a very small part of crosscutting problems that are easy to identify [52, 53]. For a large-scale refactoring of kernel code, both approaches are too limited; instead, using a semantic model of kernel tasks and related code patterns could improve the quality of machine-based aspect mining since it makes the identification of tangled and scattered code easier and provides a relation to the code's functional context.

References

[1] M. Engel, B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In: *Proceedings of the Fourth International Conference on Aspect Oriented Software Development*, ACM, New York, pp. 51–62, 2005

[2] S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quaterman. *The Design and Implementation of the 4.4 BSD UNIX Operating System.* Addison-Wesley, Reading, 1994

[3] M. Engel. Advancing operating systems via aspect-oriented programming. Dissertation, Department of Mathematics and Computer Science, University of Marburg, 2005

[4] ISO. The Open Systems Interconnection (OSI) Reference Model Standard, 1979

[5] S. McCanne, V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In: *Proceedings of USENIX'93*, Usenix, pp. 259–269, 1993

[6] R. Braden, T. Faber, M. Handley. From protocol stack to protocol heap: role-based architecture. *SIGCOMM Computer Communication Review*, 33:17–22, 2003

[7] J. Liedtke. μ-Kernels must and can be small. In: *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Seattle, WA, IEEE Press, pp. 152–155, 1996

[8] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, W. Schröder-Preikschat. A quantitative analysis of aspects in the eCOS kernel. In: *Proceedings of EuroSys 2006*, ACM SIGOPS, 2006 (to appear)

[9] J. Mogul. TCP offload is a dumb idea whose time has come. In: *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Usenix, pp. 25–30, 2003

[10] G. Kiczales, M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In: *Proceedings of the 2005 European Conference on Object-Oriented Programming (ECOOP)*, pp. 195–213, 2005

[11] M. Allman. RFC3465: TCP Congestion Control with Appropriate Byte Counting (ABC). IETF, 2003

[12] B. Scheuermann, C. Lochert, M. Mauve. Implicit hop-by-hop congestion control in wireless multihop networks. In: *Preprint*, University of Duesseldorf, 2005

[13] I. Foster, C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufman, New York, 1997

[14] M. Smith, T. Friese, B. Freisleben. Towards a service-oriented ad hoc grid. In: *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing*, Cork, Ireland, IEEE Press, New York, pp. 201–208, 2004

[15] I. Foster, C. Kesselman. Globus: a metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11:115–128, 1997

[16] T. Jensen, D. Le Metayer, T. Thorn. Security and dynamic class loading in Java: a formalisation. In: *IEEE International Conference on Computer Languages*, Chicago, Illinois, pp. 4–15, 1998

[17] Sun Microsystems: Java Native Interface Specification. Sun, 1997

[18] N. Provos. Improving host security with system call policies. In: *Proceedings of the 12th Usenix Security Symposium*, Usenix, pp. 257–272, 2003

[19] B.M. Cantrill, M.W. Shapiro, A.H. Leventhal. Dynamic instrumentation of production systems. In: *Proceedings of the USENIX Annual Technical Conference*, USENIX, pp. 15–28, 2004

[20] V. Prasad, W. Cohen, F.C. Eigler, M. Hunt, J. Keniston, B. Chen. Locating system problems using dynamic instrumentation. In: *Proceedings of the Ottawa Linux Symposium*, Linux International, pp. 49–64, 2005

[21] P.T. Barham, B. Dragovic, K. Fraser, S. Hand, T.L. Harris, A. Ho, R. Neugebauer. Xen and the art of virtualization. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 164–177, 2003

[22] F. Bellard. QEMU, a fast and portable dynamic translator. In: *Usenix 2005 Annual Technical Conference*, Usenix, pp. 41–46, 2005

[23] A. Tamches, B.P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In: *Proceedings of Third Symposium on Operating Systems Design and Implementation (OSDI)*, ACM, New York, pp. 117–130, 1999

[24] R. Bryan, J.K.H. Buck. An API for runtime code patching. In: *Journal of High Performance Computing Applications*, 14(4), 2000

[25] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J.S. Ong, S. Gudmundson. Position summary: aspect-oriented system structure. In: *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, IEEE, Piscataway, p. 166, 2001

[26] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J.S. Ong. Structuring system aspects: using AOP to improve OS structure modularity. In: *Communications of the ACM*, vol. 44, Issue 10, ACM, New York, pp. 79–82, 2001

[27] O. Spinczyk, A. Gal, W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In: *Proceedings of the Fortieth International Conference on Tools*, Australian Computer Society, Inc., pp. 53–60, 2002

[28] B. Griswold, E. Hilsdale, J. Hugunin, V. Ivanovic, M. Kersten, G. Kiczales, J. Palm. Tutorial—Aspect-Oriented Programming with AspectJ, 2001

[29] O. Spinczyk, D. Lohmann. Using AOP to develop architectural-neutral operating system components. In: *Proceedings of the 11th ACM SIGOPS European Workshop (SIGOPS04)*, ACM, New York, 2004

[30] A. Gal, W. Schröder-Preikschat, O. Spinczyk. On minimal overhead operating systems and aspect-oriented programming. In: *Proceedings of the 4th Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSW)*, Universidad de Oviedo, 2001

[31] R. Grimm. xtc: making C safely extensible. In: *Proceedings of the Workshop on Domain-Specific Languages for Numerical Optimization*, Argonne, 2004

[32] S. Bray, M. Yuen, Y. Coady, M.E. Fiuczynski. Managing variability in systems: oh what a tangled OS we weave. In: *Proceedings of the Workshop on Managing Variabilities Consistently in Design and Code (MVCDC OOPSLA04)*, pp. 6–7, 2004

[33] O. Rohlik, A. Pasetti, P. Chevalley, I. Birrer. An aspect weaver for qualifiable applications. In: *Proceedings of the 2004 Conference on Data Systems in Aerospace (DASIA)*, pp. 10–16, 2004

[34] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, S. Eggers. Extensibility, safety and performance in the SPIN operating system. In: *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ACM, New York, pp. 267–283, 1995

[35] M.I. Seltzer, Y. Endo, C. Small, K.A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. SIGOPS *Operational System Review*, 30:213–227, 1996

[36] A.C. Veitch, N.C. Hutchinson. Kea—a dynamically extensible and configurable operating system kernel. In: *Proceedings of the Third International Conference on Configurable Distributed Systems*, pp. 123–129, 1996

[37] D.P. Ghormley, D. Petrou, S.H. Rodrigues, T.E. Anderson. SLIC: an extensibility system for commodity operating systems. In: *Proceedings of the USENIX 1998 Annual Technical Conference*, USENIX Association, pp. 39–52, 1998

[38] D.J. Pearce, P.H.J. Kelly, T. Field, U. Harder. GILK: a dynamic instrumentation tool for the linux kernel. In: *Computer Performance Evaluation / TOOLS*, pp. 220–226, 2002

[39] Y. Coady, G. Kiczales, M. Feeley, G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In: *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, ACM, New York, pp. 88–98, 2001

[40] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J.S. Ong, S. Gudmundson. Exploring an aspect-oriented approach to OS code. In: *Proceedings of the 4th Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSW)*, Universidad de Oviedo, 2001

[41] Y. Coady, G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In: *Proceedings of the Second International Conference on Aspect-Oriented Software Development*, ACM, New York, pp. 50–59, 2003

[42] O. Stampflee, C. Gibbs, Y. Coady. RADAR: really low-level aspects for dynamic analysis and reasoning. In: *Proceedings of the ECOOP Workshop on Programming Languages and Operating Systems*, ECOOP, 2004

[43] M. Segura-Devillechaise, J.M. Menaud, G. Muller, J. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, ACM, New York, pp. 110–119, 2003

[44] C. Zhang, H. Jacobsen. TinyC2: towards building a dynamic weaving aspect language for C. In: *FOAL 2003 Proceedings Foundation of Aspect-Oriented Languages Workshop at AOSD 2003*, pp. 25–34, 2003

[45] M. Deters, R.K. Cytron. Introduction of program instrumentation using aspects. In: *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, pp. 131–147, 2001

[46] S. Almajali, T. Elrad. Coupling availability and efficiency for aspect oriented runtime weaving systems. In: *Proceedings of the Second Dynamic Aspects Workshop (DAW05)*, pp. 47–55, 2005

[47] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann. Virtual machine support for dynamic join points. In: *Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)*, ACM, New York, pp. 83–92, 2004

[48] M. Haupt, M. Mezini. Micro-measurements for dynamic aspect-oriented systems. In: M. Weske, P. Liggesmeyer (eds.) *Proceedings of Net.ObjectDays. LNCS vol. 3263*, Springer Press, pp. 81–96, 2004

[49] M. Engel, B. Freisleben. Using a low-level virtual machine to improve dynamic aspect support in operating system kernels. In: *Proceedings of the ACP4IS Workshop, Conference on Aspect-Oriented Software Development*, pp. 1–6, 2005

[50] M. Bruntink, A. van Deursen, T. Tourwé. An initial experiment in reverse engineering aspects from existing applications. In: *Proceedings of the Working Conference on Reverse Engineering*, IEEE Computer Society, pp. 306–307, 2004

[51] M. Bruntink, A. van Deursen, R. van Engelen, T. Tourwé. An evaluation of clone detection techniques for identifying cross-cutting concerns. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, pp. 200–209, 2004

[52] S. Breu, J. Krinke. Aspect mining using dynamic analysis. In: *GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik*, vol. 23., Bad Honnef, Germany, pp. 21–22, 2003

[53] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, T. Tourwe. A qualitative comparison of three aspect mining techniques. In: *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, Washington, DC, USA, IEEE Computer Society, pp. 13–22, 2005

Lean and Efficient System Software Product Lines: Where Aspects Beat Objects[⋆]

Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat

Friedrich-Alexander-University Erlangen-Nuremberg, Germany
Computer Science 4 – Distributed Systems and Operating Systems
{dl, os, wosch}@cs.fau.de

Abstract. Software development in the domain of embedded and deeply embedded systems is dominated by cost pressure and extremely limited hardware resources. As a result, modern concepts for separation of concerns and software reuse are widely ignored, as developers worry about the thereby induced memory and performance overhead. Especially object-oriented programming (OOP) is still little in demand. For the development of highly configurable fine-grained system software product lines, however, separation of concerns (SoC) is a crucial property. As the overhead of object-orientation is not acceptable in this domain, we propose aspect-oriented programming (AOP) as an alternative. Compared to OOP, AOP makes it possible to reach similar or even better separation of concerns with significantly smaller memory footprints. In a case study for an embedded system product line the memory costs for SoC could be reduced from 148–236% to 2–10% by using AOP *instead* of OOP.

1 Introduction

The domain of embedded and deeply embedded devices is dominated by 8 bit μ-controllers with 0.25–4 KB of RAM and a few KB of flash memory. In 2000, more than 98% of the total worldwide CPU production (8 billion entities) were dedicated to the domain of embedded systems; 87% of the entities used in this domain were 8 bit or smaller [47]. From the viewpoint of procurement, this “old-fashioned technology” is still the best compromise with respect to functionality and costs. In areas of mass production a *few cents* decide over market success or failure – a situation that cannot be expected to change soon, given that the envisioned scenarios of *smart dust* [30], *ubiquitous computing* [48], and *proactive computing* [47] crucially depend on the bulk availability of very cheap, self-organizing “intelligent” devices.

Counting cents in hardware procurement basically leads to counting bytes in software development. To cope with hardware cost pressure and extremely limited resources, software developers for deeply embedded systems intentionally avoid modern language concepts for a better separation of concerns (SoC), as they worry about the thereby induced memory and performance overhead. Especially object-oriented programming (OOP) is still little in demand, as some of its fundamental concepts (e.g., *late*

[⋆] This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4 and SP 968/2-1.

binding by virtual functions) are known to have non-negligible costs [20]. Hence, most software development for embedded systems is still performed “ad-hoc” in C (often even assembler) with a strong focus on minimizing hardware requirements. Organized reuse and separation of concerns are often considered as less important.

1.1 System Software Product Lines

The common “ad-hoc” application-specific development approach is doomed to fail for reusable system software, such as operating systems or light-weight middleware for embedded devices. System software for this domain has to not only cope with the aforementioned resource constraints, but also with a very broad variety of functional and non-functional requirements [10]. It has to be tailorable to provide exactly the functionality required by the intended application, but nothing more. This leads to a *family-based* or *product line* approach, where the variability and commonality among family members is expressed by *feature models* [16]. The tailorability of software product lines depends mostly on the offered level of functional exchangeability (here denoted as *variability*) and functional selectability (here denoted as *granularity*). Both, variability and granularity require a (right-unique) mapping from implementation components to the features they implement; thus, a good separation of concerns. As the principal overhead of OOP is not acceptable in this domain, we advocate to use *aspect-oriented programming (AOP)* instead.

The AOP, as well as the OOP, provides means for a better separation of concerns. Today, AOP is mostly perceived as an *extension* to OOP, which leads to the impression that it has to induce similar or even higher overheads. This is understandable, as most aspect languages are actually extensions to object-oriented languages like Java. For AspectJ [31], studies show furthermore that AOP concepts may indeed lead to some extra overhead [21]. Nevertheless, the AOP itself is not limited to OOP. Language extensions have as well been proposed for non-OOP languages (such as AspectC [12]) and multi-paradigm languages (such as AspectC++ [46]). In these languages it is possible to use AOP not only as an extension, but also as an *alternative* to OOP.

1.2 Objectives

This article shows that for the specific requirements of resource-constrained software development, AOP can be superior to OOP. Our focus is *not* on separation of crosscutting concerns and other structural benefits of AOP that have earlier been discussed in many papers. Instead, we want to draw attention to a mostly unexplored benefit of AOP: given a well-optimizing static weaver, AOP can provide SoC more *resource-efficiently* than OOP. Our goals are in particular to:

- broaden the perception of AOP. Currently AOP is mostly perceived as an extension to OOP for a better SoC. Understanding it as an *alternative* to OOP with respect to *resource-efficiency* and *hardware costs* is a new point of view.
- attract new communities to AOP. Especially the systems and embedded communities have a profound skepticism regarding the suitability of high-level programming paradigms (“everything beyond C”), as they (have to) favor efficiency over SoC.

Demonstrating that with aspects it can be possible to reach SoC without having to give up efficency would help to broaden the acceptance of AOP.

For these purposes, we conducted a study in which we compare the resource requirements of three implementations of the same embedded software product line. A highly efficient C-based implementation that does not provide SoC, an object-oriented, and an aspect-oriented implementation. We evaluate the OO and AO implementations regarding the techniques used to reach SoC. The focus is on the thereby induced *costs*. While both implementations reach the SoC goals, the OO-based solution leads to significantly higher resource requirements than the AO-based solution, which can compete with plain C.

1.3 Structure of the Paper

The paper is organized as follows: in Sect. 2, we present our case study about an aspect-oriented and an object-oriented implementation of an embedded software product line. Both implementations are analyzed with respect to used idioms and potential cost drivers. In Sect. 4, these implementations are further investigated regarding their resource requirements. The results are discussed in Sect. 5. Section 6 provides an overview of related work. Finally, the paper is briefly summarized in Sect. 7.

2 Scenario

In the following, we present and analyze a case study of an embedded software product line with three different implementations: C-based, OOP-based, and AOP-based. For the sake of comprehensibility, we have conducted this case study in a more application-oriented field. The scenario is an *embedded weather station* program family that should be configurable in various different variants. It is a somewhat typical example of an embedded application software product line. The identified patterns and results are, however, equally relevant for the development of system software, which is our main field of research.

2.1 Overview

A weather station variant basically consists of one or more *sensors* to gain environmental information and one or more *actors* to process the gathered weather data. Figure 1 shows the possible variants of the weather station as a feature model. Note that the list of available sensors as well as actors is expected to grow in future versions, e.g., by additional sensors for wind direction and humidity. Not visible in the feature model, but nevertheless part of the product line definition, is that we have to distinguish between two kinds of actors:

generic actors are able to process/aggregate information of any set of sensors. *Display* and *XMLProto* are examples of generic actors. The XML representation of weather information, for instance, can easily and automatically be extended for additional sensors.

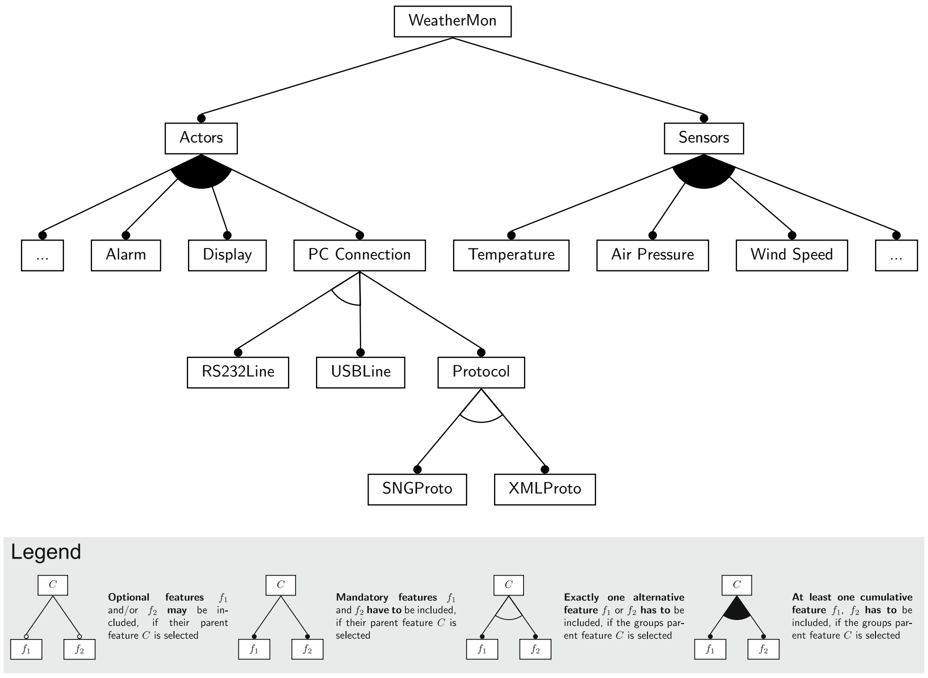

WeatherMon
Actors
Sensors
...
Alarm
Display
PC Connection
Temperature
Air Pressure
Wind Speed
...
RS232Line
USBLine
Protocol
SNGProto
XMLProto
Legend
Optional features f_1 and/or f_2 may be included, if their parent feature C is selected
Mandatory features f_1 and f_2 have to be included, if their parent feature C is selected
Exactly one alternative feature f_1 or f_2 has to be included, if the groups parent feature C is selected
At least one cumulative feature f_1, f_2 has to be included, if the groups parent feature C is selected

Fig. 1. Feature diagram of the embedded weather station product line. A *WeatherMon* variant consists of at least one sensor and one actor. *Sensors* gain information about the environment, such as *Temperature*, *Air Pressure*, or *Wind Speed*. *Actors* process sensor data: weather information can be printed on a (LCD-) *Display*, monitored to raise an *Alarm* if values exceed some threshold, and passed to a PC through a *PCConnection*, which can be either an *RS232Line* or an *USBLine*, using either an XML-based data representation (*XMLProto*) or a proprietary format (*SNGProto*).

non-generic actors process/aggregate sensor data of some specific sensors only. *SNGProto* is an example for a specific actor. It implements a legacy binary data representation for compatibility with existing PC applications. This format encodes information of certain sensors only.

2.2 Hardware Platform

On the hardware side, sensors and actors of the weather station are connected to a small μ-controller (Fig. 2). The AVR series by Atmel is a typical μ-controller product line. It is based on an 8 bit RISC core and offered with a broad variety of on-board equipment: 0–4 KB RAM, 1–128 KB program memory (flash), various busses and connectors (I^{2}C, serial lines, A/D converters). Wholesale prices of the chip scale between 0.3 and 7 EUR. They depend mostly on the on-board equipment, especially the quantity of RAM and flash memory.

The goal is to reach a similar level of scalability in the software product line, so that hardware costs do scale with the quantity of selected features.

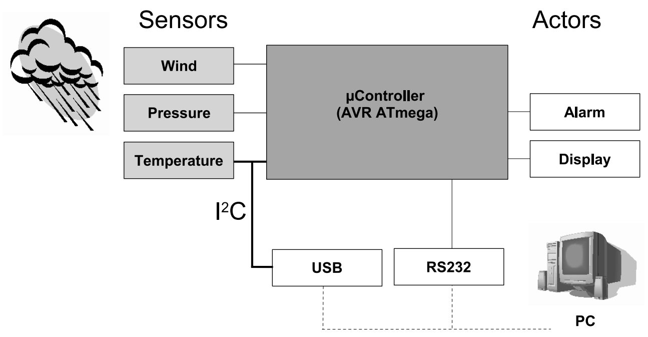


Fig. 2. Weather station hardware platform

2.3 Configuration Process

A weather station variant is configured by selecting features from the feature model. Conceptually, the feature model can be considered as the description of the *configuration space*, while a (valid) feature selection is a description of a concrete variant. In the *implementation space*, a *family model* provides a mapping from the abstract features to concrete implementation artifacts (such as .h and .cpp files) which implement the selected features. The feature selection is evaluated against the configurations space by a variant management tool [9], which copies the thereby defined set of implementation artifacts from the component repository into an output directory. The build process is then started in this directory to compile and link the configured system into the concrete variant (Fig. 3).

The approach to map conceptual features by a family or platform model to concrete implementation artifacts is quite common the domain of embedded system software such as operating system product lines.[1] The main advantages are flexibility and portability. By providing hardware-specific versions of the implementation space (family model and implementation components), the same conceptual OS model and configuration tool can be used to “generate” highly optimized variants for very different μ-controller platforms and derivates.

The overall configurability (variability and granularity) of such product lines depend, however, on the (right-unique) mapping from implementation components to the features they implement; thus, a good separation of concerns. As a matter of fact, many features cannot be mapped to single (C-language) artifacts; their implementation is intermingled within the implementation of other features. The result is scattered and tangled code, typically processed by an intra-artifact “second-order configuration” by means of the C preprocessor and conditional compilation. This hampers maintainability and evolvability of the implementation artifacts and thereby limits on the long-term extensibility of the whole product line. Hence, we strive for a better separation of concerns—as long as it does not lead to significant extra costs on the hardware side.

[1] Examples are eCos[1], PURE[10], or many implementations of the OSEK system software standard used in the automotive industry [2].

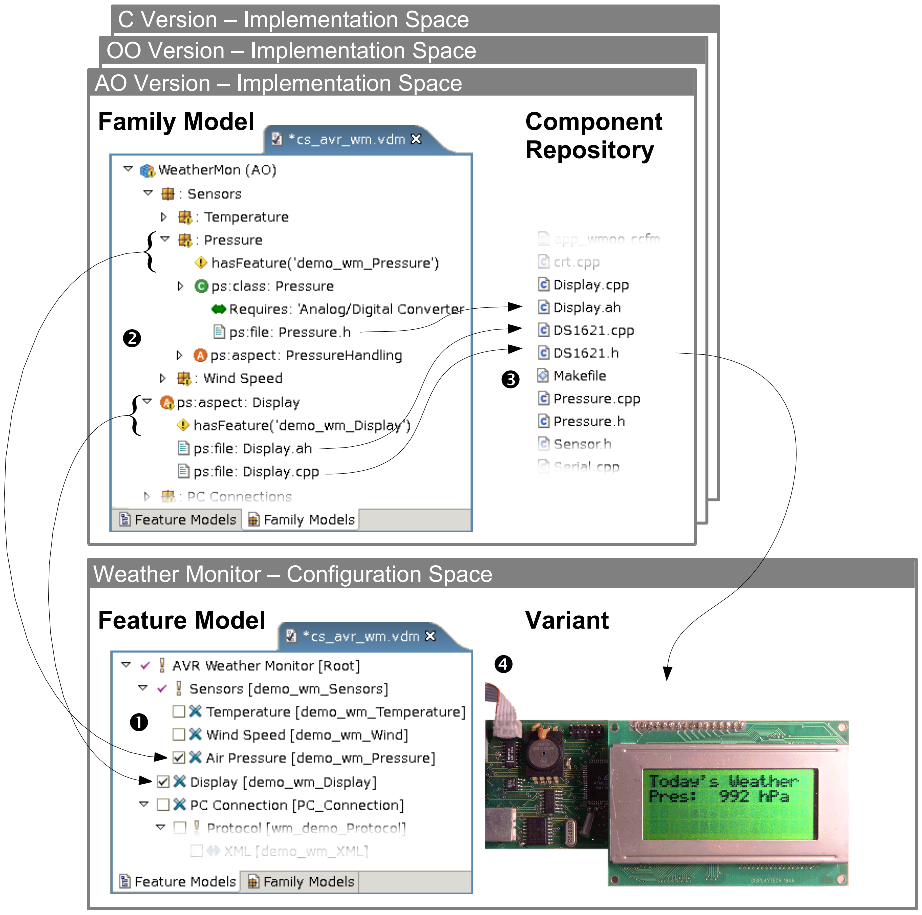

C Version – Implementation Space
OO Version – Implementation Space
AO Version – Implementation Space
Family Model
*cs_avr_wm.vdm
WeatherMon (AO)
: Sensors
: Temperature
: Pressure
hasFeature('demo_wm_Pressure')
ps:class: Pressure
Requires: 'Analog/Digital Converter
ps:file: Pressure.h
ps:aspect: PressureHandling
: Wind Speed
ps:aspect: Display
hasFeature('demo_wm_Display')
ps:file: Display.ah
ps:file: Display.cpp
: PC Connections
Feature Models
Family Models
Component Repository
app_wmoo.ccfm
crt.cpp
Display.cpp
Display.ah
DS1621.cpp
DS1621.h
Makefile
Pressure.cpp
Pressure.h
Sensor.h
Serial.cpp
❷
❸
Weather Monitor – Configuration Space
Feature Model
*cs_avr_wm.vdm
AVR Weather Monitor [Root]
Sensors [demo_wm_Sensors]
Temperature [demo_wm_Temperature]
Wind Speed [demo_wm_Wind]
Air Pressure [demo_wm_Pressure]
Display [demo_wm_Display]
PC Connection [PC_Connection]
Protocol [wm_demo_Protocol]
XML [demo_wm_XML]
Feature Models
Family Models
❶
Variant
❹
Today's Weather
Pres: 992 hPa

Fig. 3. Weather station configuration process. In the *configuration space* the user selects all wished features from the *feature model* (*1*). The *implementation space* consists of a *family model* and a *component repository*. The family model maps features to *logical implementation components* (*2*), which in turn are mapped to *physical implementation files* (*3*). The thereby determined set of implementation files for a concrete configuration is copied into the target directory, and finally compiled and linked into the actual *weather station variant* (*4*).

3 Implementation

In the following, we present and analyze two different implementations of the weather station product line that fulfill the goal of SoC. One implementation is OOP-based, while the other is AOP-based. The implementations were performed by two different, but equally experienced developers. Both have been (under the premise that SoC has to be achieved) carefully optimized with respect to resource requirements.

A third "traditional" C-based implementation is solely optimized for minimal resource consumption. Its only purpose is to provide the lower bounds of the resource consumption that can be reached in the different product line configurations.

The section is organized as follows: We start with a list of requirements in Sect. 3.1, which is followed by a description of the three implementations (C in Sect. 3.2, OO in Sect. 3.3, and AO in Sect. 3.4). These descriptions are intended as a brief overview only. In particular, they do not motivate the chosen design of the OO and AO versions. Instead, all design decisions and idioms used to reach SoC in the OO and AO versions are discussed collectively (to set them in contrast with each other) in Sect. 3.5. The results are summarized in Sect. 3.6.

3.1 Implementation Requirements

Besides the functional features that have to be implemented, the additional requirements on the implementations can be summarized as *resource-thriftiness* (all versions) and *separation of concerns* (AO and OO versions). This means in particular:

granularity. Components should be fine-grained. Each implementation element should be either mandatory (such as the application main loop) or dedicated to a single feature only.

economy. The use of expensive language features should be avoided as far as possible. For instance, a method should only be declared as `virtual` if polymorphic behavior of this particular method is required.

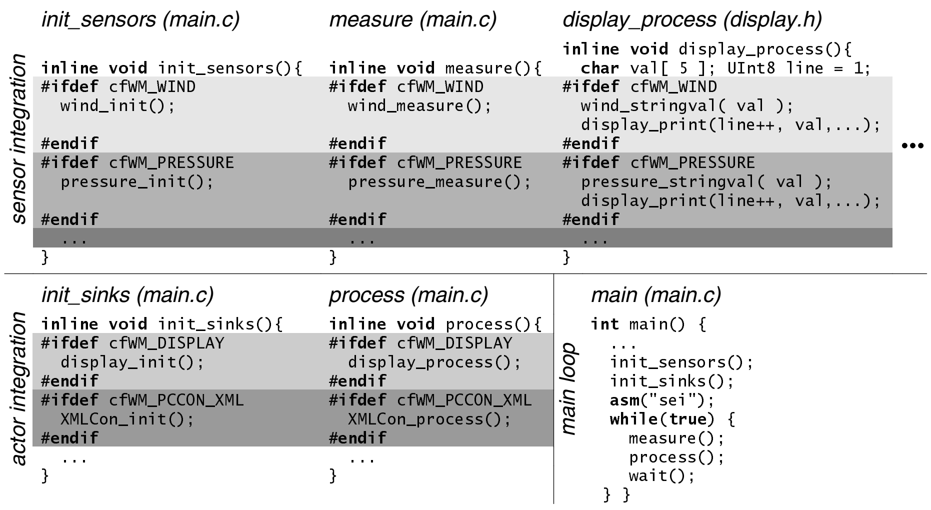
pluggability. Changing the set of selected sensors and/or actors should not require modifications of any other part of the implementation. This basically means that sensor/actor implementations should be able to integrate themselves into the system.

extensibility. The same should hold for new sensor or actor types, which may be available in a future version of the product line.

3.2 Implementation of the C Version

Figure 4 shows the basic structure of the C version. Weather information is updated by the `measure()` function, which invokes—configured by conditional compilation—all existing sensors. For efficiency reasons, the actual weather data are passed as a global variable (not shown). Weather information is processed by the `process()` function, which invokes each actor (such as `display_process()`)—again configured by means of conditional compilation. The `..._process()` function of each actor retrieves the weather information for all configured sensors it is interested in—once more using conditional compilation.

Due to the design goal to minimize the resource consumption, the C version does *not* fulfill our requirements on SoC (*granularity*, *pluggability*, *extensability*). By using global variables, conditional compilation, and inlining of all functions that are referred only once, it offers, however, efficiency.



init_sensors (main.c)

```
inline void init_sensors(){
#ifdef cfWM_WIND
  wind_init();

#endif
#ifdef cfWM_PRESSURE
  pressure_init();

#endif
  ...
}
```

measure (main.c)

```
inline void measure(){
#ifdef cfWM_WIND
  wind_measure();

#endif
#ifdef cfWM_PRESSURE
  pressure_measure();

#endif
  ...
}
```

display_process (display.h)

```
inline void display_process(){
  char val[ 5 ]; UInt8 line = 1;
#ifdef cfWM_WIND
  wind_stringval( val );
  display_print(line++, val,...);
#endif
#ifdef cfWM_PRESSURE
  pressure_stringval( val );
  display_print(line++, val,...);
#endif
  ...
}
```

init_sinks (main.c)

```
inline void init_sinks(){
#ifdef cfWM_DISPLAY
  display_init();
#endif
#ifdef cfWM_PCCON_XML
  XMLCon_init();
#endif
  ...
}
```

process (main.c)

```
inline void process(){
#ifdef cfWM_DISPLAY
  display_process();
#endif
#ifdef cfWM_PCCON_XML
  XMLCon_process();
#endif
  ...
}
```

main (main.c)

```
int main() {
  ...
  init_sensors();
  init_sinks();
  asm("sei");
  while(true) {
    measure();
    process();
    wait();
 } }
```

Fig. 4. Static structure of / scattered code in the C version (excerpt)

3.3 Implementation of the OO Version

Figure 5 shows the class model of the OO version. Central elements are the `Weather` and `Sink` classes. `Weather` aggregates all sensors, which are registered at run time by calling `Weather::registerSensor()`. The `Sink` class aggregates all actors, respectively. Internally both, sensors and actors, are managed by chaining them into light-weight single-linked lists (`ChainBase`).

Principle of Operation

1. Weather information is acquired by calling `Weather::measure()`, which in turn invokes the `Sensor::measure()` method on every registered sensor to update the sensor data.
2. Weather information is processed by the `Sink::process()` method. `Sink::process()` first calls `Actor::before_process()` for each registered actor to initialize the processing.
3. Sensor information is passed to the actors by calling `Actor::process()` for each registered sensor. Actors retrieve the actual sensor name, unit, and measured data (as character strings) by calling the respective Sensor methods.
4. Finally, data processing is finalized by `Sink::process()` invoking `Actor::after_process()` on each actor.
5. The whole process is repeated by the application main loop every second.

3.4 Implementation of the AO Version

The class/aspect model of the AO version is shown in Fig. 6. Central elements are, again, the classes `Weather` and `Sink`. Each sensor class is accompanied by a *han-*

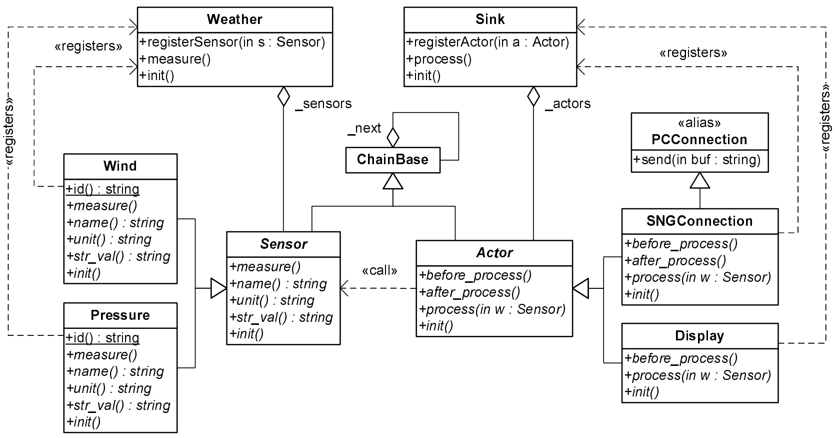


Fig. 5. Static structure of the OO version (excerpt with two sensors/actors, virtual functions are depicted in *italics*, static functions are underlined)

dling aspect, which performs the actual integration into `Weather` and `Sink`. A handling aspect performs two introductions: it aggregates the sensor as an instance variable into class `Weather` and a sensor-specific (empty) `process_data()` method into class `Sink`. Additionally, it defines function execution advice for `Weather::measure()` and `Sink::process`. Actors are implemented as aspects, which define execution advice for `Sink::process()` (for initialization and finalization) and the sensor-introduced `Sink::process_data()` methods (for the actual data processing).

Principle of Operation

1. Weather information is acquired by calling `Weather::measure()`, which is advised by the handling aspects of every sensor to call the sensor's `measure()` method.
2. Weather information is processed by the `Sink::process()` method. `Sink::process()` is before-advised by any actor that needs to initialize before processing.
3. `Sink::process()` is advised for each sensor to call the introduced sensor-specific `process_data()` method.
4. The sensor-specific `process_data()` method is advised by every actor that processes data of this sensor.
5. Finally, data processing is finalized by `Sink::process()` being after-advised for any actor that needs to finalize its processing.
6. The whole process is repeated by the application main loop every second.

3.5 Used AOP and OOP Idioms

To achieve the required level of SoC as well as pluggability and extensibility, both versions use approach-specific idioms and patterns in the design and implementation. In

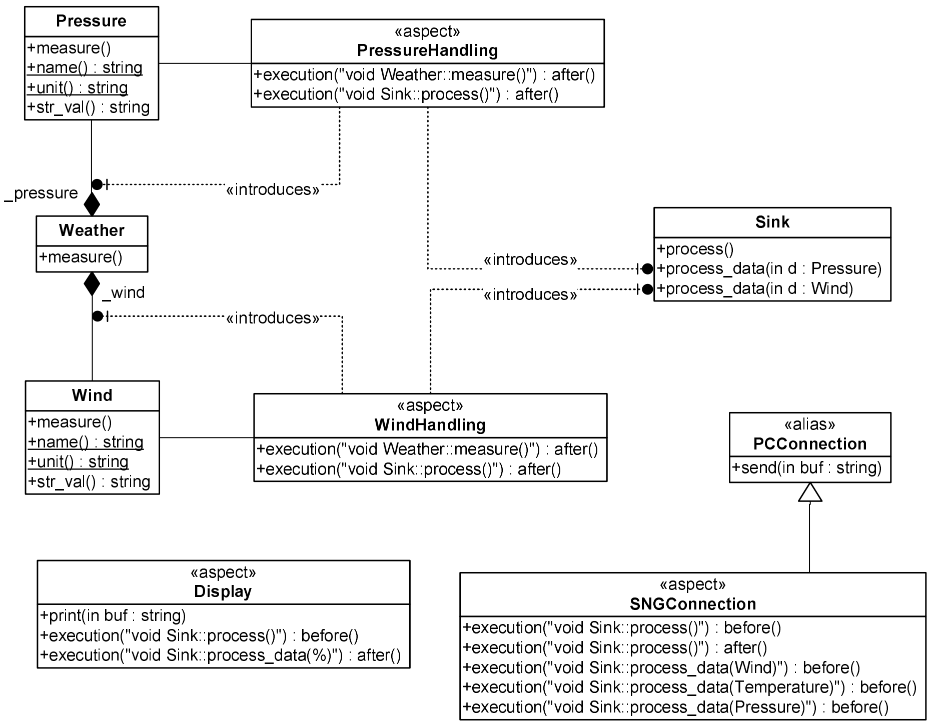

Pressure
+measure()
+name() : string
+unit() : string
+str_val() : string
«aspect»
PressureHandling
+execution("void Weather::measure()") : after()
+execution("void Sink::process()") : after()
_pressure
«introduces»
Weather
+measure()
_wind
«introduces»
Sink
+process()
+process_data(in d : Pressure)
+process_data(in d : Wind)
«introduces»
«introduces»
Wind
+measure()
+name() : string
+unit() : string
+str_val() : string
«aspect»
WindHandling
+execution("void Weather::measure()") : after()
+execution("void Sink::process()") : after()
«alias»
PCConnection
+send(in buf : string)
«aspect»
Display
+print(in buf : string)
+execution("void Sink::process()") : before()
+execution("void Sink::process_data(%)") : after()
«aspect»
SNGConnection
+execution("void Sink::process()") : before()
+execution("void Sink::process()") : after()
+execution("void Sink::process_data(Wind)") : before()
+execution("void Sink::process_data(Temperature)") : before()
+execution("void Sink::process_data(Pressure)") : before()

Fig. 6. Static structure of the AOP version (excerpt with two sensors/actors)

the following, we will analyze some interesting parts of the AOP and OOP implementations. The goal is to identify, compare, and discuss the idioms that have to be used to reach SoC. In particular, the following problems had to be solved.

3.5.1 Working with Configuration-Dependent Sensors/Actors Sets

Both implementations have in common that the `Weather`/`Sink` classes are used to abstract from the configured sets of sensors and actors. These two abstractions are unavoidable, because the `main()` function, which performs the endless measurement and processing loop, should be configuration-independent and, thus, robust with respect to extensions. However, the implementation of the 1:n relationship of `Weather`/`Sink` and the concrete sensors and actors is completely different.

Interface dependency between Weather/Sink and sensors/actors

The `Weather`/`Sink` classes need to be able to invoke the measuring/processing of data independent of the actual sensor/actor *types*. Otherwise, pluggability and extensibility would be broken:

OO version: In the OO version, this is solved by *common interfaces* and *late binding*. Sensors have to inherit from the abstract class `Sensor`, actors from the abstract class `Actor`, respectively. `Weather`/`Sink` invoke sensors/actors via these interfaces, and thus depend on them. `Sensor::measure()`, `Actor::before_process()`, `Actor::after_process()`, and `Actor::process()` have to be declared as virtual functions:

```
struct Sensor : public ChainBase {
  virtual void measure() = 0;
  ...
};
...
class Weather {
public:
  ...
  void measure () {
    for( Sensor* s = ... )
      s->measure();        // virtual function call!
  }
};
```

AO version: In the AO version, the interface relationship is reverted. `Weather` and `Sink` do not depend on sensors/actors providing any specific interface. Sensors/actors are integrated by defining function execution advice for `Weather::measure()` and `Actor::process()`:

```
class Weather {
public:
  void measure () {} // empty implementation
};
...
aspect PressureHandling {
public:
  advice "Weather" : Pressure _pressure;
  advice execution ("void Weather::measure()") : before () {
    // non-virtual inlineable function call
    theWeather._pressure.measure ();
  }
  ...
};
```

Potential cost drivers: In the OO version, four methods in two classes have to be declared and called as virtual functions. The AO version induces no costs, as all sensor/actor code can be inlined.

Working with sets of sensors/actors

`Weather`/`Sink` need to be independent of the actual *number* of sensors/actors configured into the system. There is a 1:n relationship between `Weather`/`Sink` and sensors/actors:

OO version: In the OO version, this is solved by a simple *publisher/subscriber* mechanism. Sensors are chained into a linked list of “publishers”, which is frequently iterated by `Weather::measure()` to update the data of each sensor:

```
class Weather {
  static  Sensor* _sensors;
public:
  Sensor* firstSensor() const {
    return _sensors;
  }
  void measure () {
    for( Sensor* s = firstSensor(); s != 0;
         s = static_cast< Sensor* >( s->getNext() ) )
      s->measure();
    }
    ...
};
```

Actors are similarly chained into a list of "subscribers" which are invoked if new sensor data are available. The `Sink` class acts as "mediator" between sensors and actors. For the sake of memory efficiency, actors do not subscribe for single sensors, but are implicitly subscribed for the complete list.

AO version: In the AO version, actors and sensors are implicitly chained at compile time by multiple aspects defining advice for the same `Weather::measure()` and `Sink::process()` execution joinpoints.

Potential cost drivers: In the OO version, `Weather`/`Sink` as well as sensors/actors need to carry an extra pointer for the chaining. Some additional code is required to iterate over the chain of sensors/actors.[2]

Registration of sensors/actors
This problem is closely related to the previous one. Sensors and actors need to be able to register themselves for the publisher/subscriber chain:

OO version: In the OO version, registration is done at run time by calling `Weather::registerSensor()`/`Sink::registerActor()`. Self-registration requires some C++ trickery: Each sensor/actor is instantiated as global object. The registration is performed by the constructor, which is "automatically" triggered during system startup.

AO version: In the AO version, no extra efforts are required for self-registration. The chaining of actors and sensors is implicitly performed at compile time by advice-code weaving. Basically, it is the presence of some aspect in the source tree that triggers the registration.

Potential cost drivers: In the OO version, the use of global object instances causes some overhead, as the compiler has to generate extra initialization code and puts a reference to this code in a special linker section. A system startup function has to be provided that iterates over this section and invokes the constructors of all global instances.

3.5.2 Implementation of Generic Actors

Generic actors process sensor data from any sensor type. For example, the Display actor should print the measured values of any configured sensor, regardless of the current configuration or future extensions with new sensor types. However, this leads to an interface dependency between actors and sensors, because (at least) the value has to be obtained in a generic way:

[2] The `firstSensor()` and `getNext()` operations are, however, inlined as they perform just a pointer lookup.

OO version: In the OO version, this is again solved by *interfaces* and *late binding*. To enable actors to retrieve sensor information from any sensor, the `Sensor` interface has to be extended by three additional virtual functions: `Sensor::name()`, `Sensor::unit()`, and `Sensor::str_val()`:

```
class Display : public Actor {
public:
    UInt8 _line;

  // print a line on the display, increment _line
  void print(const char *name, const char *val_str, const char *unit );
  ...
  virtual void before_process() {
    _line = 1;
  }
  // called by Sink::process for every sensor
  virtual void process( Sensor* s ) {
    char val[ 5 ];
    s->str_val( val );
    print( s->name(), val, s->unit() );
  }
};
```

AO version: In the AO version, this is solved by using the AspectC++ concept of *generic advice* [32, 35]. For every sensor `MySensor`, the corresponding `MySensorHandling` aspect introduces an empty sensor-specific `process_data(const MySensor&)` method into class `Sink` and gives advice to `Sink::process()` to invoke the introduced method:

```
aspect PressureHandling {
public:
  advice "Weather" : Pressure _pressure;
  ...
  // introduce an empty process_data function for the pressure
  advice "Sink" : void process_data( const Pressure & ) {}
  // call Sink::process_data for the pressure
  advice execution("void Sink::process()") : after () {
    theSink.process_data( theWeather._pressure );
  }
};
```

An actor gives generic advice that matches every (overloaded) `Sink::process_data()` function. To be independent of the actual sensor type, the advice body uses the AspectC++ joinpoint API to retrieve a typed reference to the sensor instance:

```
aspect Display {
  ...
  // display each element of the weather data
  advice execution("void Sink::process_data(%)") : before () {
    typedef JoinPoint::template Arg<0>::ReferredType Data;
    char val[5];
    tjp->arg<0>()->str_val( val );
    print( Data::name(), val, Data::unit() );
  }
};
```

AspectC++ instantiates advice bodies per joinpoint. Therefore, the calls to the actual sensor's `str_val()`, `name()`, and `unit()` methods can be bound at compile time. As an additional optimization, `name()` and `unit()` are implemented as static (class) functions.

Potential cost drivers: In the OO version, three additional virtual functions are required in the sensor classes, as well as an additional virtual function calls in the actor classes.

In the AO version, the joinpoint API has to be used for a uniform and type-safe access to the sensor instance, which induces some overhead. Furthermore, the advice body of a generic actor is instantiated once per sensor, which may lead to code bloating effects.

3.5.3 Implementation of Non-generic Actors

Non-generic actors process data of some sensors only. The legacy SNG protocol, for instance, encodes weather data in a record of wind speed, temperature, and air pressure. It exposes the actual data using sensor-specific interfaces. The record may be sparse, meaning that a specific sensor, such as Temperature, may or may not be present in the actual system configuration.

OO version: In the OO version, a non-generic actor filters the sensors it is interested in by applying *run time type checks* in the `process()` method. Each passed sensor is tested against the handled sensor types. If the run-time type matches with a handled sensor, a downcast is performed to get access to the sensor-specific interface:

```
class Pressure : public Sensor {
  static const char *_id;
public:
  ...
  static const char * id () { return _id; }
  const char *name () const { return _id; }
};
...
class SNGConnection : public Actor, protected PCConnection {
    UInt8 _p, _w, _t1, _t2;     // weather record
public:
  virtual void before_process() { ... /* init record */ };
  virtual void after_process() { ... /* transmit record */ };
  ...
 // collect wind, pressure, temperature data
  virtual void process( Sensor* s ) {
  const char *name = s->name();
    if (name == Wind::id()) // pointer comparison
      _w = ((Wind*)s)->_w;
    else if (name == Pressure::id())
      _p = ((Pressure*)s)->_p - 850;
    else if (name == Temperature::id()) {
      _t1 = (UInt8)((Temperature*)s)->_t1;
      _t2 = ((Temperature*)s)->_t2;
  } }
};
```

The idiom commonly used for run-time type checks in C++ is the `dynamic_cast` operator, which is part of the C++ *run-time type interface* (*RTTI*). RTTI is, however, quite expensive, as it requires additional run-time support and leads to some extra overhead in *every* class that contains virtual functions. To avoid this overhead, our implementation uses a "home-grown" dynamic type-check mechanism: The test is performed by comparing the string address returned by the (late-bound) `Sensor::name()` method with the address of the name string stored in the concrete class,[3] which is also returned by the static `Sensor::id()` method. The expensive C++ RTTI mechanism has been disabled.

Normally, dynamic type checks are considered harmful, because of a lack of extensibility and the accumulated costs of type checks, which sometimes outweigh the costs

[3] This basically reduces the overhead of a run-time type test to a virtual function call and a pointer comparison. As the storage for the name string has to be provided anyway, this mechanism also induces no extra overhead in the sensor classes.

of a single virtual function call. However, in our case a *non-generic* actor shall be implemented. Therefore, extensibility is not an issue and the overhead of our type check implementation is acceptable. At the same time, alternative designs such as a *visitor* [24] fail, because a Visitor interface would have to list a `visitSensor()` method for every sensor type. However, the set of sensors is configurable and should nowhere be hard-wired.

AO version: In the AO version, binding an actor to selected sensors only is realized by giving advice for specific `Sink::process_data()` methods only instead of using generic advice:

```
aspect SNGConnection : protected PCConnection {
  UInt8 _p, _w, _t1, _t2;      // weather record
  ...
  // let this aspect take a higher precedence than <Sensor>Handling
  advice process () : order ("SNGConnection", "%Handling");

  advice execution("void Sink::process(const Weather&)")
      : before () { ... /* init record */ }
  advice execution("void Sink::process(const Weather&)")
      : after () { ... /* transmit record */ }

  // collect wind, pressure, temperature data by giving specific advice
  advice execution("void Sink::process_data(...)") && args (wind)
      : before (const Wind &wind) {
    _w = wind._w;
  }
  advice execution("void Sink::process_data(...)") && args (pressure)
      : before (const Pressure &pressure) {
    _p = pressure._p - 850;
  }
  advice execution("void Sink::process_data(...)") && args (temp)
      : before (const Temperature &temp) {
    _t1 = (UInt8)temp._t1;
    _t2 = temp._t2;
  }
};
```

Potential cost drivers: Run-time type checks in the OO version induce nevertheless some overhead. In the AO version, some overhead is induced by the `args()` pointcut function, which is used here to get the actual sensor instance.

3.6 Implementation Summary

Both, the OOP as well as the AOP implementation of the embedded weather station product line provide good SoC. In particular, the implementation requirements described in Sect. 3.1 are met by both versions:

granularity is achieved by the OO as well as the AO version. Each implementation component is either mandatory (such as the `Weather` and `Sink` classes), or dedicated to a single feature only.

economy is achieved as far as possible. In the OO version, only methods that have to be available via a generic interface are declared as virtual. RTTI is not used, as the required run-time type checks can be implemented with less overhead. In the AO version, joinpoint-specific context information is used only sparingly.

pluggability is achieved as well. In both versions, no component has to be adapted if the set of selected sensors/actors is changed. Sensors and actors basically integrate themselves, if their implementation component is present in the configured source tree. The OO version uses global instance construction for this purpose. In the AO version, the integration is performed by advice.

extensibility is also achieved. In the OO version, new sensor/actor types just need to implement the common `Sensor`/`Actor` interface. In the AO version, new sensor/actor types just need to provide some aspect that performs the integration into `Weather`/`Sink`.

Overall, the AO and OO versions are equipollent from the SoC viewpoint. We, however, identified noticeably more potential cost drivers in the OO version than in the AO version. Especially virtual functions were unavoidable in many places to realize loose coupling and genericity of components. In the next section, we analyze how this affects scalability and memory demands of the product line.

4 Cost Analysis

In this section, we analyze scalability and memory requirements of the embedded weather station product line. For this purpose, several configurations of the weather station were generated as AO, OO, and C variants.[4] For each variant, we measured:

- static memory demands, which are determined by the amount of generated machine code (*text*), static initialized data (*data*), and static non-initialized data (*bss*).
- dynamic memory demands, which are determined by the maximum stack space used by the running application (*stack*).[5]
- the run time of a complete measure/process-cycle.

On the actual hardware, *text* occupies flash memory space. *Data* occupies flash memory *and* RAM space, as it is writable at run time and therefore has to be copied from flash into RAM during system startup. *Bss* and *stack* occupy RAM space only.

4.1 Measurement Methods

Static memory demands (*text*, *data*, *bss*) could easily be retrieved directly from the linker map file. Dynamic memory demands (*stack*) and run time had to be measured in the running targets:

[4] All variants were compiled with `avr-g++` (GCC) 3.4.1 using `-Wall -fno-rtti -Os -fno-exceptions -fomit-frame-pointer -ffunction-sections` optimization flags. AO variants were woven with `ac++` 0.9.3.

[5] The weather station software uses no heap, which otherwise would also contribute to dynamic memory demands.

4.1.1 Measuring Stack Utilization

For simple programs it is possible to determine the stack utilization offline and byte-exact by static analysis of the machine code. Unfortunately in our case the program execution graph is not predictable due to the use of interrupts and late bound functions. Therefore we used run-time monitoring of the stack as a pragmatic alternative. A common technique for run-time stack monitoring is to initialize the entire stack space with some specific *magic pattern* during system startup. The maximum quantity of stack used can then be measured at any time by searching (from bottom of stack) for the first byte where the pattern has been overwritten.

In the weather station variants, this technique was used to implement stack measurement as an additional *sensor type*. Understanding stack measurement as just another sensor had some nice advantages. Because of the achieved pluggability (in the AO and OO versions) it was very easy to apply stack measurement to any weather station configuration. Of course, some extra care had to be taken to ensure that the maximum stack utilization is not caused by the stack measurement sensor itself. For this reason, the stack measurement implementation uses only global variables which do not occupy stack space. By analyzing the generated machine code we ensured that the stack utilization of the stack sensor methods is minimal among all sensors. As all sensor methods are invoked from the same call depth level, and at least one "real" sensor besides stack measurement is used in a weather station configuration, it can thereby be guaranteed that stack measurement itself does not tamper the maximum stack utilization. In the actual targets, the thereby acquired maximum stack utilization remained stable after a short startup time and could be read from one of the attached generic actors.

4.1.2 Measuring Run Time

Run-time measuring was not implemented as another sensor type, as it had been too difficult to distinguish the run time taken by the sensor processing itself from the run time of the target to measure. Instead, we used a less invasive approach that could be implemented with just two additional assembler statements: in the application main loop, a digital I/O port of the AVR μ-controller is set to *high* before the call to `Weather::measure()` and reset to *low* after the return of `Sink::process()`. The result is a rectangular signal on this port, which was recorded and analyzed with a storage oscilloscope.[6] A *high* period in the signal represents the run time taken by a complete `Weather::measure()` / `Sink::process()` cycle. After a short startup time, period and phase of the signal remained stable, and the length of the *high* phase could be measured.

4.2 Overall Scalability of the Product Line

As the graphs in Fig. 7 show, the resulting RAM/flash demands of the weather station software do scale quite well with the amount of selected features. The "Barometer" configuration (P+Display), consisting of just an air pressure sensor and an LCD

[6] Tektronix TDS 2012, 100 MHz resolution.

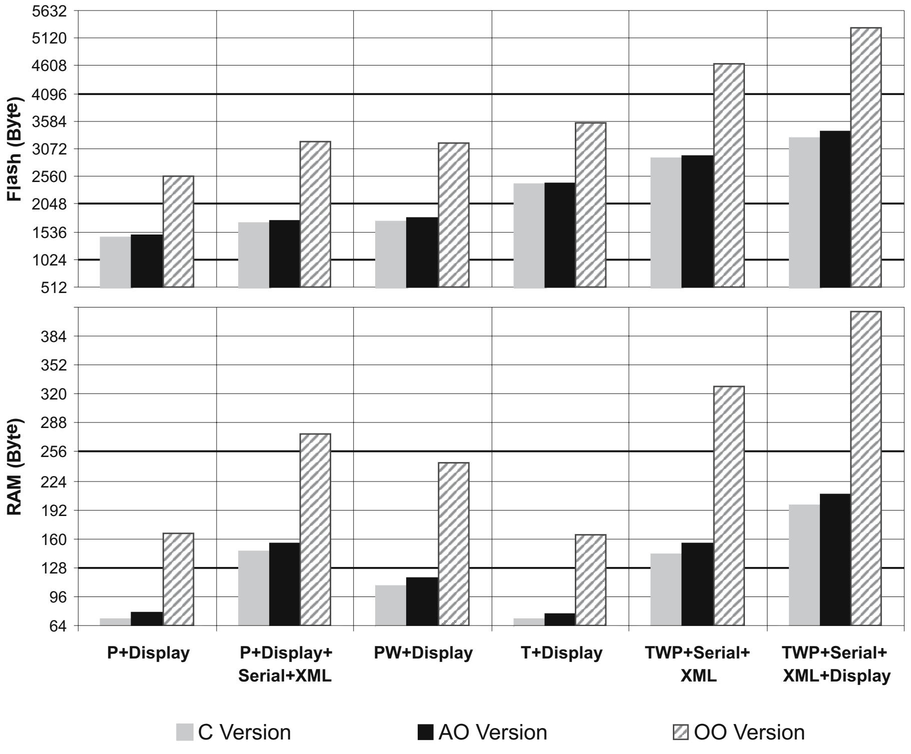


Fig. 7. Footprint comparison (RAM, flash) for different configurations

display, induces significantly smaller memory demands than the "Deluxe-PC" configuration (TWP+Serial+XML+Display) which bundles three sensors, a LCD display, and an XML-based PC connection over a serial interface. The memory requirements of the other examined configurations are in-between. The noticeably high amount of flash memory required by the "Thermometer" configuration (T+Display) can be explained by the fact that this sensor is connected via the I^2C bus to the μ-controller (see Fig. 2). To drive this bus, additional driver code is required that has not been included for other sensors.

Overall, all three versions meet the goal of scalability, which is an indicator for achieved granularity. In every case, however, the OOP version requires significantly more memory space than its AOP counterpart which comes very close to the C version. Depending on the configuration, the required quantity of RAM is up to 138% higher in the OO version, while the AO version takes only an extra of 10% at maximum (up to 13 byte)—both compared to the C-based version that does not provide SoC. The amount of flash memory is up to 91% higher in the OO version, but only 4% at maximum in the AO version. The net difference between using AOP and OOP has to be considered as even higher, as both versions of each configuration are linked with the same

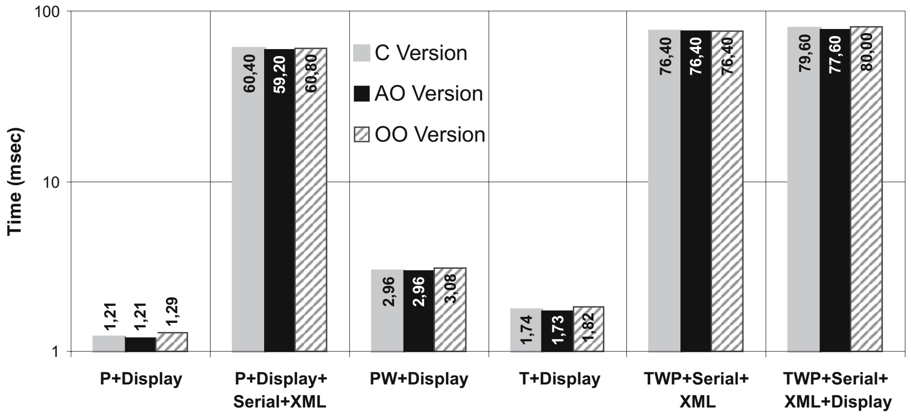


Fig. 8. Footprint comparison (time taken by a `Weather::measure()` / `Sink::process()` cycle) for different configurations

(configuration-dependent) set of device drivers whose memory requirements are included in these numbers.[7]

The run time of all three versions is almost the same in all configurations (see Fig. 8). It is mostly dominated by hardware and driver-related costs.

4.3 Memory Requirements in Detail

Table 1 breaks down the required overall amount of RAM and flash memory into their origins. It is evident, that the OO variants induce especially higher static memory demands. The *text* sections of the OO variants are up to 78%, the *data* sections up to 284% bigger than in the AO variants. The following cost drivers can be mainly accounted for this:

virtual functions are the main source of additional code and data, as they induce overhead on both, the caller and the callee side. On the caller side, each call to a virtual function requires, compared to advice code inlining, at least 16 additional byte in the machine code. On the callee side, virtual function tables have to be provided (4 byte + 2 byte per entry, *data*), object instances need to carry a pointer to the virtual function table (2 byte per instance), and constructors to initialize the vtable pointer of an instance have to be generated (at least 24 byte per class, *text*). In the “Barometer” configuration (P+Display), for instance, 52 byte of the *data* section are occupied solely by virtual function tables.[8]

Regarding code size, the situation may become even worse in larger projects: due to late binding, the bodies of virtual functions are never removed from the final image

[7] As parts of the driver code are implemented as inline functions, it is not possible to differentiate between driver-induced and application-induced code here.

[8] The AVR RISC core uses a Harvard architecture; thus vtables cannot be placed in the text section.

Table 1. Memory usage and run time of the AO and OO versions for different configurations

Configuration		text	data	bss	stack	flash	ram	time
P+Display	C	1392	30	7	34	1422	71	1.21
	AO	1430	30	10	38	1460	78	1.21
	OO	2460	100	22	44	2560	166	1.29
P+Display+ Serial+XML	C	1578	104	7	34	1682	145	60.40
	AO	1622	104	12	38	1726	154	59.20
	OO	3008	206	26	44	3214	276	60.80
PW+Display	C	1686	38	14	55	1724	107	2.96
	AO	1748	38	18	61	1786	117	2.96
	OO	3020	146	33	65	3166	244	3.08
T+Display	C	2378	28	8	34	2406	70	1.74
	AO	2416	28	11	38	2444	77	1.73
	OO	3464	98	23	44	3562	165	1.82
TWP+Serial+ XML	C	2804	90	17	35	2894	142	76.40
	AO	2858	90	23	41	2948	154	76.40
	OO	4388	248	39	41	4636	328	76.40
TWP+Serial+ XML+Display	C	3148	122	17	57	3270	196	79.60
	AO	3262	122	24	63	3384	209	77.60
	OO	5008	300	44	67	5308	411	80.00

flash := text + data ram := bss + stack + data (Bytes) (ms)

by means of function-level linking. Thus, "dead" function code that is never called at run time becomes nevertheless part of the *text* section.

dynamic data structures are another source of additional overhead. The chaining of actors and sensors induces 8 additional data byte in the "Barometer" configuration, plus some extra code to access and iterate over the lists.

global instance construction causes some more "hidden" code to be generated. For each translation unit that defines one or more global objects, the compiler has to generate a specific initialization-and-destruction function (88 bytes, *text*). Pointers to these functions are additionally stored in the *data* section.

Regarding dynamic memory usage (*stack*), the differences between the AO and OO version are less significant. The OO variants need a few byte (up to 16%) more stack space than the related AO variants. This seems surprising at first, given that virtual function calls cannot be inlined, which therefore lead to a higher call depth and, thus, higher stack utilization. Part of this effect can be explained by the fact that the AOP version requires some additional stack space as well, namely by context-binding pointcut functions and the joinpoint API (2–4 byte). The main reason is, however, that the maximum virtual function call depth is with two levels quite low. As the AVR architecture provides 32 general-purpose registers, which are also used for passing function parameters, it can furthermore be considered as quite "stack-friendly". On other CPU architectures (such as Intel), the differences between AO- and OO-based solutions would be more significant.

4.4 Run-Time Requirements in Detail

Table 1 also lists the measured run times in detail. In all configurations that support a serial connection the measurement/processing cycle time is mainly dominated by the underlying serial device driver. Here, the costs are almost the same. In all other configurations the performance of the AO version and our highly efficient C implementation are the same. The run-time overhead of the OO version is between 4 and 6.6%. It can be explained with the numerous virtual function calls on the application level.

4.5 Cost Analysis Summary

Both, the AO and OO version of our product line, scale quite well. The run-time differences are small and only of minor importance in this domain. The OO version, however, induces dramatically higher memory requirements. Given a hardware product line like the AVR ATmega series, these differences can directly be mapped to the required μ-controller features. As Fig. 9 shows, the hardware costs of the C-based and AO-based product line would be exactly the same in all configurations. Using AO versus OO leads even for our small example product line to significant differences regarding hardware costs. This effect would probably be even higher for larger product lines.

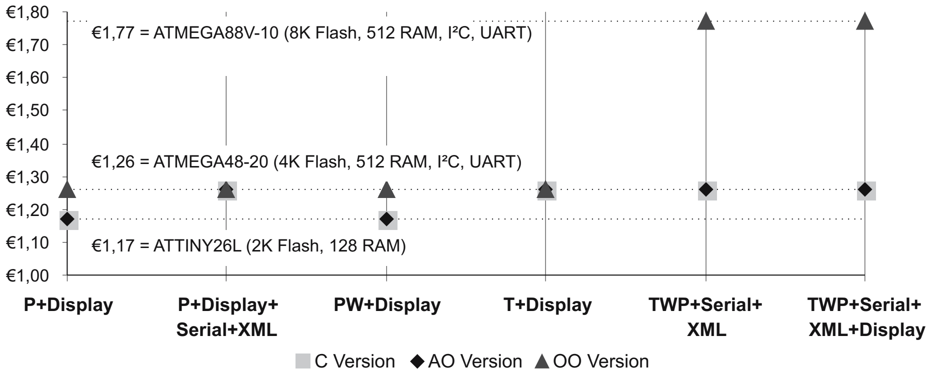


Fig. 9. Hardware scalability (required μ-controller variant) of the AO and OO versions for different configurations (wholesale prices: Digi-Key Corporation, Product Catalogue, Summer 2005)

5 Discussion

As pointed out in the previous sections, the AOP-based version of the weather station product line combines good SoC with high memory efficiency. In the following we discuss the principal reasons for this effect, the potential disadvantages, and to what degree our results are applicable to (system software) product lines in general and achievable with other AOP approaches.

5.1 General Advantages of the Approach

To ensure variability and granularity in product lines, components need to be implemented in a fine-grained and loosely coupled way. In our case study, we identified three fundamental concepts that are required to reach this: *Abstraction from concrete components* (interfaces), *abstraction from component sets* (1:n relationships), and *abstraction from component integration* (self-pluggability). The major disadvantage of OOP is that the features it provides to reach these goals are basically run-time concepts, while with AOP (in the case of static weaving) it is possible to use compile-time concepts:

abstraction from concrete components requires to use interfaces and run-time binding in OOP (thus, virtual functions in C++), while in AOP advice code is woven at compile time to the joinpoints (and in AspectC++ is even inlined).

abstraction from component sets requires to use some kind of run-time data structures in OOP, such as linked lists, while in AOP multiple advice can be woven at the same joinpoint. Moreover, by means of aspect ordering, it is even possible to define a context-dependent invocation order.

abstraction from component integration is generally difficult to achieve in most OOP languages. In C++ it requires some constructor trickery to perform the integration at run time. In AOP, Advice is *per se* an integration mechanism and resolved at compile time. Moreover, aspects provide means for context-sensitive integration: Giving advice for a specific joinpoint can be understood as a *weak reference* to another component, as the advice is silently ignored if the joinpoint does not exist.

5.2 Compiler Issues

As pointed out in the analysis, much of the extra overhead of the OO version is directly or indirectly induced by the costs of late binding, even though the extra flexibility at run time is not required in every case. Much work has been conducted to reduce the costs of C++ virtual functions [3, 8] and late binding in general, e.g., by static whole program optimization [18]. This gives rise to the question, if and to what extent the measured OO overhead could be prevented by better compilers.

Especially in the context of the highly complex and non-academic programming languages C and C++, which are the state of the art in system software and embedded software development, the implementation of optimizations based on static whole program analysis is extremely difficult. Although many of the aforementioned algorithms and techniques were proposed at least 10 years ago, today's commercial compilers have not incorporated these innovations yet. The main reasons are their complexity and existing de-facto standards like object file formats and ABIs.[9] Therefore, we pragmatically advocate for using OO and AO mechanisms in combination in order to get the best out of both worlds. While OO is well suited for applications that actually need dynamism, static advice weaving is an interesting alternative that avoids the OO overhead from the beginning if dynamism is not required.

5.3 Applicability to System Software

Compared to application software product lines, system software product lines have to fulfill even higher demands on configurability and resource-efficiency. With respect to efficiency, the aforementioned abstractions are particularly useful in system software:

[9] ABI = *Application Binary Interface.*

efficient callbacks by advice code inlining. System software usually has to provide means to bind user-defined code to (asynchronous) system events, such as hardware interrupts in an operating system. To prevent uses-dependencies [41] of the system code to driver/application code, a *callback* or *upcall* mechanism via function-pointers has to be provided. This leads to extra overhead and higher latencies at run time. As aspects reverse such dependencies [14], they provide means to break user-dependencies without this extra overhead.

handler chains by multiple advice. The implementation of handler chains can be implemented by the same mechanism as single callbacks. Due to advice code inlining, this furthermore induces no additional overhead.

configurable initialization order by aspect precedences. In system software it is often necessary to initialize components and subsystems in some well-defined (partial) order during system startup. The order is determined by component-interdependencies; however, not necessarily fixed in the case of configurable product lines. It may well depend on the actual configuration. By aspect precedences (given as *order advice* in AspectC++), aspects provide perfect means to solve this problem at compile time.

5.4 Potential Disadvantages of the Approach

The advantages regarding resource efficiency are mainly caused by binding and inlining component code at compile time that is bound at run time in OOP. This may, however, also lead to limitations of the approach:

compile time fixation limits the approach to static configuration of product lines. In domains, where the set of selected features can arbitrarily change at run time, additional support has to be provided, e.g., by a dynamic aspect weaver. As most dynamic weaving approaches induce inevitable overhead [27], it is questionable if in such cases AOP still outweighs the overhead of OOP. However, in most cases only some features can be expected to change at run time. A tailorable low-cost weaving infrastructure, which provides means to configure *per-feature* whether static or dynamic weaving should be used [45], may be a promising solution for such cases, as this enables developers to optimize the trade-off between dynamism and resource overhead at configuration time.

code bloating due to advice code inlining. This occurs if functionality given by advice is invoked from (and therefore woven to) many different joinpoints. While this situation has to be considered for highly crosscutting concerns such as tracing, component composition usually takes place at few well-known joinpoints only. In any case, however, the programmer is able to prevent this effect by simply moving the advice body into a non-inlined function that is invoked from the advice code.

5.5 Applicability to Other AOP Approaches

Our results show, that in the case of C++ and AspectC++, using AOP instead of OOP may lead to significant saving effects in terms of memory and hardware costs. This gives rise to the question, whether similar results can be reached with other languages and/or AOP approaches as well. This is clearly a huge field for further research and cannot be answered in this paper. We think, however, that the following points are of particular importance:

weaver run time support may lead to some unavoidable base overhead that limits the reachable quantity of memory savings. This is clearly an issue for the domain of embedded systems. Experiments conducted with AspectJ on Java2 Micro Edition, for instance, have shown that the AspectJ run time induces an extra 35 KB overhead [49]. AOP approaches that do not require run-time library support are probably better suited for the goal of resource-thriftiness. Besides AspectC++, XWeaver [44] and AspectC [12], for instance, may be promising candidates.

language capabilities of the host language have a high influence on the applicability of the approach. As the resource savings are mainly caused by using compile time instead of run-time concepts, the host language has to support such concepts. On the one hand, it is probably difficult to get "rid of the OOP overhead" in languages that have been particularly designed as object-oriented languages, such as Java or C#. On the other hand, languages such as C may not be sufficiently expressive with respect to (compile time) genericity to take full advantage of the approach.

5.6 Design Issues

The actual design and implementation of the AO and OO versions have a significant influence on the resulting memory and performance numbers. Both versions have achieved the *granularity*, *pluggability*, and *extensibility* goals. One might, however, question if especially the design of the OO version is optimal with respect to the *economy* goal as well. Theoretically, it might be possible to find a better OO implementation that leads to lower memory demands.

As a matter of fact, it is impossible to proof that the design and implementation of some software is optimal with respect to run-time resource requirements. The authors have, however, profound experience in developing resource-minimal object-oriented product lines [10, 11, 34]. There is, furthermore, quite some evidence that any OO version would lead to higher memory requirements than the AO version: as pointed out in the previous sections, run-time mechanisms have to be used with OO to reach the required flexibility which, as described in Sects. 3.5 and 4, induce overhead. Given that the OO version takes up to 138% more RAM and up to 91% more flash memory than the C and AO versions; it is hard to imagine that these differences could be levelled by a better OO implementation.

For the AO version the resource requirements are only minimally higher than for the C version. Hence, it can already be considered as "nearly optimal". In a recent study on the eCos operating system product line [1], we could show that the C-mechanism to reach configurability (conditional compilation), can be replaced by AOP concepts without any extra run time and memory costs. In this study, we refactored several hundred `#ifdef`-blocks caused by configurable features into aspects [33].

6 Related Work

Several papers have been published that analyze and compare different implementation techniques of product line variabilities [23, 36]. All authors come to the conclusion that AOP is a very promising approach, although still not widely used in the industrial practice [38]. In the middleware area several case studies showed that AOP allows software developers to scale features and footprint by supporting the static configuration of fine-grained configurable features [13, 50, 51]. All of these studies were conducted with AspectJ. In one case the authors clearly expressed that AspectC++ would be more appropriate for the embedded systems domain [28].

Other proposed implementation techniques were Generative Programming [16], the application of frame processors such as XVCL [29] or Angie [19], and Feature-Oriented Programming (FOP) [6]. All three techniques have in common with AOP that in a layered system design the higher-level layers can use special means to refine the behavior and/or structure of the lower-level layers. This *dependency alignment* [14] is a key to successful product line design. In comparison to frame processing and C++ template-based generative programming [17], the main advantage of the AOP approach is its *obliviousness*, i.e., refined layers do not have to be prepared for intervention by aspects. In comparison with refinements in FOP, the advantage of AOP is *quantification*, i.e., a single aspect can affect various different joinpoints. Of course, FOP is much more than just the refinement mechanism. Therefore, the novel FeatureC++ language [5] is very promising, as it combines AOP and FOP.

When using OOP, the discussed refinement can only be achieved with virtual functions at the cost of dynamic binding. However, in many cases dynamic binding offers more flexibility than it is actually needed. Much work has been conducted to reduce the costs of virtual functions [3, 8, 20], but these optimizations are still not state of the art in current C++ compilers.

Another alternative approach is to combine OOP with partial program evaluation tools like Tempo [15] and C-Mix [4], especially as partial evaluation has succesfully been applied in the context of system software [37]. However, the authors are not aware of any tools or concept papers that cover the combination of these techniques with OOP. Furthermore, the case studies on partial evaluation of system software focus on performance improvements. Some of these techniques increase the code size by keeping specialized and original code in memory at the same time or by applying dynamic compilation [42].

In the systems software area configurable software is often component based. Popular examples are the OSKit [22, 39] and TinyOS [7]. In both cases the requirements on performance and footprint can only be achieved with additional tool support. For example, Knit can be regarded as a weaver for OSKit components [43]. TinyOS is written in the NesC language [25]. Here, the compiler performs a whole program transformation in order to optimize the system. In both cases the overhead, which is the result of the component model, is later removed by an additional tool. With the AOP approach an overhead is avoided from the beginning.

An interesting alternative to layered designs is Subject-Oriented Programming (SOP) [26, 40]. This technique is one of the roots of AOP. It supports system composition from *slices* instead of layers. In our embedded weather station case study, the sensors are in fact slices. By using the introduction mechanism of AspectC++, SOP was simulated.

7 Summary and Conclusions

The development of fine-grained and resource-efficient system software product lines requires means for separation of concerns that do not lead to an extra overhead in terms of memory and performance. Especially in the important domain of small embedded systems, where even a few additional bytes of code or data may lead to higher hardware costs, the inherent overhead of OOP is not acceptable for most developers. For this domain we propose AOP as an alternative.

In this paper, we compared an object-oriented with an aspect-oriented implementation of an configurable embedded software product line. We could show that the aspect-oriented implementation induces significantly lower memory requirements than the object-oriented implementation, while providing similar or even better qualities with respect to separation of concerns. Although our results were produced with C++ and AspectC++, similar effects should be achievable with any AOP approach that does not require additional run-time support and performs inlining of advice code. Advice code inlining also leads to an excellent performance. Our measurements have shown that the performance of the AO version was identical with the performance of our C-based reference implementation.

Overall, our results show that aspects provide very good separation of concerns in conjuction with high resource efficiency. This makes AOP well suited for the development of embedded system software product lines—where aspects beat objects.

Regarding future work, we will continue to evaluate costs and benefits of applying AOP to the operating systems domain. We are furthermore working on a benchmark suite that provides detailed results regarding the run time and memory costs of AspectC++ features.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. We furthermore thank Danilo Beuche from pure::systems GmbH for leaving us the weather station hardware. Finally, a big thankyou goes to the editors, especially Hans-Arno Jacobsen, whose suggestions helped us a lot to improve the quality and comprehensibility of this paper. This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4 and SP 968/2-1.

References

[1] eCos homepage. http://ecos.sourceware.org/
[2] OSEK/VDX standard. http://www.osek-vdx.org/
[3] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. Technical Report TRCS95-22, Computer Science Department, University of California, Santa Barbara, December 1995

[4] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD Thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19)

[5] S. Apel, T. Leich, M. Rosenmller, and G. Saake. FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming. In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE '05)*, Tallinn, Estonia, 2005

[6] D. Batory. Feature-oriented programming and the AHEAD tool suite. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, IEEE Computer Society, pp. 702–703, 2004

[7] U.C. Berkeley. TinyOS homepage. http://www.tinyos.net/

[8] D. Bernstein, Y. Fedorov, S. Porat, J. Rodrigue, and E. Yahav. Compiler optimization of C++ virtual function calls. In: *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS '96)*, Toronto, Canada, 1996

[9] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/

[10] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In: *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99)*, St Malo, France, pp. 45–53, 1999

[11] D. Beuche, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Streamlining object-oriented software for deeply embedded applications. In: *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS '00)*, Mont Saint-Michel, France, pp. 33–44, 2000

[12] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In: *Proceedings of the 3rd Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE '01)*, 2001

[13] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin. Using AspectJ for component integration in middleware. In: *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, ACM, New York, pp. 339–344, 2003

[14] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical Report COMP-001-2004, Lancaster University, 2004

[15] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es), 1998

[16] K. Czarnecki and U.W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, Reading, 2000

[17] K. Czarnecki and U.W. Eisenecker. Synthesizing Objects. In: R. Guerraoui (ed.), *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, *LNCS vol. 1628*, Lisbon, Portugal, Springer, Berlin Heidelberg New York, pp. 18–42, 1999

[18] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In: *Proceedings of PLDI '95*, La Jolla, CA, 1995

[19] Delta Software Technology GmbH. Angie – an introduction, 2005

[20] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In: *Proceedings of the 11th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*, 1996

[21] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of AspectJ programs. In: *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, ACM, New York, pp. 150–169, 2004.

[22] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: a substrate for Kernel and language research. In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '97), ACM Operating Systems Review*, ACM, New York, pp. 38–51, 1997

[23] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In: *Proceedings of 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, ACM, New York, pp. 109–117, 2001

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, 1995

[25] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: a holistic approach to networked embedded systems. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, ACM, San Diego, New York, pp. 1–11, 2003

[26] W. Harrison and H. Ossher. Subject-oriented programming—a critique of pure objects. In: *Proceedings of the 8th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '93)*, pp. 411–428, 1993

[27] M. Haupt and M. Mezini. Micro-measurements for dynamic aspect-oriented systems. In: *NetObjectDays (NODe '04)*, Erfurt, Germany, *LNCS vol. 3263*, Springer, Berlin Heidelberg New York, pp. 81–96, 2004

[28] F. Hunleth and R. Cytron. Footprint and feature management using aspect-oriented programming techniques. In: *Proceedings of the 2002 Joint Conference on Languages, Compilers and Tools for Embedded Systems & Soft. and Compilers for Embedded Systems (LCTES/SCOPES '02)*, Berlin, Germany, ACM, New York, pp. 38–45, 2002

[29] S. Jarzabek and H. Zhang. XML-based method and tool for handling variant requirements in domain model. In: *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE '01)*, Toronto, Canada, IEEE Computer Society Press, pp. 116–123, 2001

[30] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Next century challenges: mobile networking for "smart dust". In: *International Conference on Mobile Computing and Networking (MOBICOM '99)*, pp. 271–278, 1999

[31] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. In: J. Lindskov Knudsen (ed.), *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01), LNCS vol. 2072*, Springer, Berlin Heidelberg New York, pp. 327–353, 2001

[32] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: on the combination of AOP with generative programming in AspectC++. In: G. Karsai and E. Visser (ed.), *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, *LNCS vol. 3286*, Springer, Berlin Heidelberg New York, pp. 55–74, 2004

[33] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In: *Proceedings of the EuroSys 2006 Conference (EuroSys '06)*, ACM, New York, pp. 191–204, 2006

[34] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. On the design and development of a customizable embedded operating system. In: *Proceedings of the SRDS Workshop on Dependable Embedded Systems (SRDS-DES '04)*, IEEE Computer Society, pp. 1–6, 2004

[35] D. Lohmann and O. Spinczyk. On typesafe aspect implementations in C++. In: F. Geschwind, U. Assmann, and O. Nierstrasz (eds.), *Proceedings of Software Composition 2005 (SC '05)*, Edinburgh, UK, *LNCS vol. 3628*, Springer, Berlin Heidelberg New York, pp. 135–149, 2005

[36] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In: *Proceedings of ACM SIGSOFT '04 / FSE-12*, 2004

[37] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the sun commercial rpc protocol. In: *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, ACM, New York, pp. 116–125, 1997

[38] D. Muthig and T. Patzke. Generic implementation of product line components. In: *NetObjectDays (NODe '02)*, Erfurt, Germany, *LNCS vol. 2591*, Springer, Berlin Heidelberg New York, pp. 313–329, 2003

[39] University of Utah. OSKit homepage. http://www.cs.utah.edu/flux/oskit

[40] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, pp. 43–50, 2001

[41] D.L. Parnas. Some hypothesis about the uses hierarchy for operating systems. Technical report, TH Darmstadt, Fachbereich Informatik, 1976

[42] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1): 11–32, 1988

[43] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: component composition for systems software. In: *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, USA, pp. 347–360, 2000 Usenix Association

[44] O. Rohlik, A. Pasetti, V. Cechticky, and I. Birrer. Implementing adaptability in embedded software through aspect oriented programming. In: *Proceedings of Mechatronics & Robotics (MechRob '04)*, Aachen, Germany, IEEE Computer Society Press, 2004

[45] W. Schröder-Preikschat, D. Lohmann, W. Gilani, F. Scheler, and O. Spinczyk. Static and dynamic weaving in system software with AspectC++. In: Y. Coady, J. Gray, and R. Klefstad (eds.), *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS '06) - Mini-Track on Adaptive and Evolvable Software Systems*. IEEE Computer Society, 2006

[46] O. Spinczyk, D. Lohmann, and M. Urban. Advances in AOP with AspectC++. In: H. Fujita and M. Mejri (eds.), *New Trends in Software Methodologies, Tools and Techniques (SoMeT '05), LNCS vol. 129*, Frontiers in Artificial Intelligence and Applications, Tokyo, Japan, IOS, pp. 33–53, 2005

[47] D. Tennenhouse. Proactive computing. *Communications of the ACM*, pp. 43–45, 2000

[48] M. Weiser. The computer for the 21st centrury. *Scientific American*, 265(3):94–104, 1991

[49] T. Young and G. Murphy. Using AspectJ to build a product line for mobile devices. *AOSD '05 Demo*, Chicago, Illinois, March 2005

[50] C. Zhang, D. Gao, and H.-A. Jacobsen. Generic middleware substrate through modelware. In: *Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference (Middleware '05)*, Grenoble, France, ACM, New York, 2005

[51] C. Zhang and H.-A. Jacobsen. Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11), 2003

Providing Configurable QoS Management in Real-Time Systems with QoS Aspect Packages

Aleksandra Tešanović[1], Mehdi Amirijoo[1], and Jörgen Hansson[1,2]

[1] Linköping University, Department of Computer Science, Linköping, Sweden
{alete, meham, jorha}@ida.liu.se
[2] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA
hansson@sei.cmu.edu

Abstract. Current quality of service (QoS) management approaches in real-time systems lack support for configurability and reusability as they cannot be configured for a target application or reused across many applications. In this paper we present the concept of a QoS aspect package that enables developing configurable QoS management for real-time systems. A QoS aspect package represents both the specification and the implementation of a set of aspects and components that provide a number of QoS policies. A QoS aspect package enables upgrades of already existing systems to support QoS performance assurance by adding aspects and components from the package. Furthermore, a family of real-time systems can easily be developed by adding aspects from the QoS aspect package into an existing system configuration. We illustrate the way a family of real-time database systems is developed using the QoS aspect package with a case study of an embedded real-time database system, called COMET. Our experiments with the COMET database have shown that it is indeed possible to design a real-time system without QoS management and then with a reasonable effort add the QoS dimension to the system using a QoS aspect package.

1 Introduction

Real-time systems are characterized by rigid requirements on quality of service (QoS). Namely, the failure to deliver a correct response in a timely manner results in deterioration of system performance and, in the worst case, a catastrophe. For example, in multimedia applications decoding of frames (images or sound) has to be done in a timely manner, otherwise the results of the computations are of no value [1]. Other examples can be found in telecommunications where a user request (e.g., placing a call) has to be addressed as soon as possible and packets have to arrive in a timely manner to avoid poor voice quality [2]. Furthermore, the engine of a modern car is controlled by an electronic control unit (ECU) that continuously monitors and controls the engine using actuators [3]. The control computations, e.g., fuel injection, must be finished within a certain time frame, otherwise the performance of the engine decays and can lead to an engine breakdown, potentially resulting in a catastrophe for the driver

[3, 4]. For these reasons, the majority of real-time research over the years has focused primarily on delivering appropriate methods for ensuring real-time QoS, e.g., [5, 6, 7, 8, 9, 10, 11, 12]. In recent years, however, with a dramatic increase in the use of real-time systems, the requirements on low development costs, short time to market, and high degree of configurability have become increasingly important in real-time system development [13].

Since QoS management is a core issue of real-time computing, configurability and reusability of software cannot be achieved without first ensuring configurability and reusability of QoS management. However, in our study of existing work we have observed that approaches for QoS management in real-time systems [5, 6, 7, 8, 9, 10, 11, 12] do not comply with essential software engineering principles that enable configuration and reuse.

In this paper we address the software engineering challenges in developing configurable QoS management for real-time systems by proposing the concept of a *QoS aspect package*, which represents a way of packaging the specification and implementation of real-time QoS management for reuse and configuration. At an abstract level, a QoS aspect package represents a way of specifying configurable QoS management, where a real-time QoS management policy is specified independently of an application by means of aspects and components. At a concrete level, a QoS aspect package consists of aspects and components implementing a variety of QoS management policies.

When studying existing QoS management approaches, we observed that a majority of approaches assume that a real-time system has a QoS management infrastructure upon which algorithms implementing specific QoS policies are implemented. The infrastructure is implemented as an integral part of the system and consists of services, functions, or methods for adjusting the system load. Algorithms implementing QoS policies crosscut the overall system, and use the services provided by the infrastructure to ensure performance guarantees.

The concept of a QoS aspect package provides a way of decomposing and implementing QoS management for a family of real-time applications. The decomposition is done such that the main functional infrastructure of QoS management is encapsulated into a set of components and various QoS policies are encapsulated into aspects. For a family of real-time applications a QoS aspect package has a unique set of components and possibly a great number of aspects implementing various QoS policies.

A QoS aspect package enables the following:

- specification and implementation of QoS policies independent of a real-time application,
- upgrades of existing systems to support QoS performance assurance by simply adding aspects that implement different QoS policies from the QoS aspect package,
- development of families of real-time systems with variations in QoS management, where variations (aspects from the package) are injected into the QoS management infrastructure at explicitly declared joinpoints of the components,

- offline and online configuration of QoS management as aspects can either be woven into the resulting system offline and then deployed into the run-time environment (preferred option in most real-time systems), or can be deployed into the running system online [14] (preferred when a high degree of availability is required),
- reusability of QoS management as aspects implemented within a QoS aspect package can be reused in multiple applications,
- configurability and reusability of QoS management for a large class of real-time systems across many application areas as the QoS aspect package concept can be used in any real-time system conforming to a set of requirements elaborated in this paper, e.g., available joinpoints in the code and conformance to an aspect language.

Since a QoS aspect package could consist of many aspects and components, the designer might need assistance in choosing the relevant subset of aspects and components for configuring QoS management of a system. Therefore, we provide appropriate tools for configuring QoS management. The tools also assist in determining if a suggested QoS configuration, i.e., adding of a QoS aspect package, is feasible for the target application.

We present a proof of concept implementation of a QoS aspect package for an embedded real-time database, called COMET [15, 16]. We report our experiences in using a QoS aspect package for real-time system development. We believe that they are valuable for current and future implementors of real-time systems that would like to use aspects for system development.

The paper is organized as follows. In Sect. 2 we present a background to real-time systems and discuss how QoS is maintained. The problem formulation is then presented in Sect. 3, where we identify problems in current QoS management approaches for real-time systems. We propose, in Sect. 4, a QoS aspect package as a possible solution to the identified problems. In Sect. 5 we present the COMET database, to which QoS management has been added using a QoS aspect package. Experiences from using aspect-orientation in real-time system design and implementation are discussed in Sect. 6. The paper finishes with the main conclusions in Sect. 7.

2 QoS Management

We now review the main characteristics of real-time systems and then discuss the key properties of QoS management in these systems.

2.1 Real-Time System Model

Real-time systems are traditionally constructed of concurrent programs, called tasks. A task is characterized by a number of temporal constraints amalgamating a task model. One of the most important temporal constraints a task needs to satisfy is a deadline, which represents the time point by which the task needs to be completed.

Depending on the consequence of a missed deadline, real-time systems can be classified as hard or soft. In hard real-time systems, e.g., aircraft and train control, the consequences of missing a deadline can be catastrophic. In soft real-time systems, e.g., video streaming and mobile services, missing a deadline does not cause catastrophic damage to the system but affects performance negatively.

Tasks in a system are scheduled by a real-time scheduler that determines an order of task execution that allows tasks to meet their respective deadlines [7]. Depending on the type of a real-time system and the scheduling policy used, different task models are applicable. For example, in hard real-time systems scheduling policies typically assume periodic tasks or sporadic tasks (aperiodic tasks with a minimum interarrival time), with known worst-case execution times [7]. Knowing the interarrival times and the worst-case execution times of the tasks enables a designer to check whether deadlines are met. Since no deadline misses can be tolerated, analysis of the system is normally done before system deployment, implying that the utilization of the system is fixed beforehand.

The assumptions taken for hard real-time systems are relaxed for soft real-time systems, i.e., for soft real-time systems it is assumed that the tasks can arrive aperiodically and that worst-case execution times are not available.[1] Instead, tasks in such systems are associated with an estimate of the average execution time. The execution time is typically estimated by profiling the code or by running experiments and monitoring the execution time of a task.

Since tasks in a soft real-time system arrive with unknown interarrival times and inaccurate worst-case execution times, the workload submitted to the system is unpredictable and can in the worst case cause the system to be overloaded. This means that it is difficult to adjust the utilization to a certain level beforehand (since admitted workload and utilization are related) and missing deadlines is inevitable. The latter follows from the fact that there is a correlation between the utilization of a system and deadline misses [7]. Rather than striving to achieve a certain utilization or meet deadlines, the focus in the resource allocation and task scheduling in soft real-time systems lies in mechanisms for ensuring QoS predictability, i.e., guaranteeing that the utilization does not exceed a certain threshold and no more than a certain number of tasks miss their deadlines during a period of time.

In recent years, a new set of requirements focusing on reuse and reconfiguration of real-time systems have emerged, resulting in the introduction of a number of approaches to development of reconfigurable real-time software using component-based and/or aspect-oriented software development, e.g., [16, 18, 19, 20]. In these approaches, components (possibly woven with aspects) are executed by tasks in a run-time environment. A task typically executes one or multiple operations a

[1] Note that in practice deriving accurate worst-case execution times is difficult as execution times depend on branches in the code, cache, processor pipelining, shared resources (forcing tasks to wait for allocated resources), and scheduling. This also implies that hard real-time system premises, e.g., the existence of accurate worst-case execution time estimates, are not realistic for a majority of recently developed intricate and large-scale real-time systems [17].

component. Which component is going to be executed by which task is determined before the system is deployed, in a process called component-to-task mapping, depending on the available resources in the underlying real-time environment. In this work we allow tasks and components to be mapped arbitrarily, i.e., without loss of generality a task can execute several components, and/or a component can consist of multiple tasks. We omit details about the mapping in this paper due to its length, and instead refer the interested reader to [20, 21] for further details.

2.2 QoS Management in Real-Time Systems

In this paper we focus primarily on methods for QoS management in soft real-time systems. This is because soft real-time systems are founded on more realistic assumptions than hard, i.e., worst-case execution times do not need to be known and aperiodic tasks are supported. Also, soft real-time systems constitute a wide spectrum of existing and emerging real-time applications, such as telecommunication, web servers, video streaming, and automatic and vehicle control.

Most of the research dealing with QoS management in real-time systems today uses some form of feedback for controlling the QoS, as it has been shown that feedback control is highly effective to support the specified performance of soft real-time systems that exhibit unpredictable workloads. For example, it has been shown that exact estimates of task execution times and arrival patterns are not required when feedback control is used [8, 22]. This implies that we can apply feedback control-based QoS management techniques on a wide spectrum of large-scale and complex systems where, e.g., execution time estimates are not available. We therefore primarily focus on QoS management using feedback control and refer to it as feedback-based QoS management.

In the remainder of this section, we introduce the general feedback control structure that has been used in all feedback-based QoS management approaches, and elaborate on representative QoS management approaches in real-time systems.

Feedback Control Structure. A typical structure of a feedback control system is given in Fig. 1 along with the control-related variables. A sampled variable $a(k)$ refers to the value of the variable a at time kT, where T is the sampling period and k is the sampling instant. In the remainder of the paper we omit k where the notion of time is not of primary interest.

Input to the controller is the difference between the reference $y_r(k)$, representing the desired state of the controlled system, and the actual system state given by the controlled variable $y(k)$, which is measured using the sensor. Based on the performance error, $y_r(k) - y(k)$, the controller changes the behavior of the controlled system via the manipulated variable $\delta_u(k)$ and the actuator. The objective of the control is to compute $\delta_u(k)$ such that the difference between the desired state and the actual state is minimized, i.e., we want to minimize $(y_r(k) - y(k))^2$. This minimization results in a reliable performance and system adaptability as the actual system performance is closer to the desired performance.

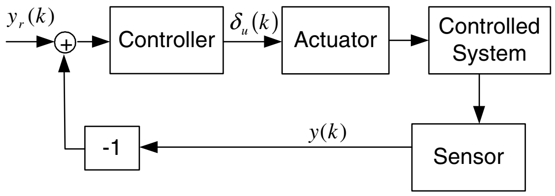


Fig. 1. An architecture of the real-time system using feedback control structure

Feedback-Based QoS Management of Real-Time Systems. Here we review the main feedback-based QoS management policies used in various real-time applications. The goal is to illustrate the variety of sensors, actuators, and controllers used in QoS management of real-time systems and to further motivate our work.

In controlling the utilization in soft real-time systems [6, 8, 11, 23, 24], the sensor periodically measures the utilization of the system. A controller compares the utilization reference with the actual measured utilization and computes a change to the manipulated variable. The actuator then carries out the change in utilization, in one of the following ways:

– The precision or the quality of the result of the tasks can be modified by applying imprecise computation techniques [25], which have been introduced to allow flexibility in operation and to provide means for achieving graceful degradation during transient overloads, e.g., a task is decomposed into a mandatory part producing a nonoptimal but satisfactory result, and optional parts that refine the initial result. This class of techniques enables trading CPU resource needs for the precision of requested service results. Using these techniques, the utilization is decreased by reducing the execution time of the tasks. This in turn implies that the precision of the task results is lowered. Conversely, the result precision increases as the utilization increases.
– The utilization is directly related to the the interarrival times of the tasks, i.e., the utilization increases with decreasing interarrival times. Hence, the utilization can be easily adjusted by changing the interarrival times of the tasks [8, 11].
– Since the utilization of the system increases with number of admitted tasks, it can be changed by enforcing admission control, where a subset of the tasks are allowed to execute [22].

In a number of real-time applications, e.g., video streaming and signal processing, tasks can deliver results of less precision in exchange for timely delivery of results. For example, during overloads alternative filters with less output quality and execution time may be used, ensuring that task deadlines are met. In general, the precision of the results increases with the execution time given to a task, calling for the use of a feedback structure to control the precision [5, 12, 26]. In such a structure, the sensor is used to estimate the output quality of the tasks, while a controller forces the tasks to maintain an output precision equal to the reference. The execution time given to individual tasks is controlled by the actuator, thereby, ensuring that the output precision is maintained at the desired level.

In telecommunication and web servers, arriving packets and requests are inserted into queues, where the requests wait to be processed. The time it takes to process requests once they arrive at a server is proportional to the length of the queue, i.e., the processing time increases with the length of the queue. Controlling the queue length is the key to guarantee timely processing of requests. If the queue length is too long, then the time it takes to process a request may be unacceptable as there are time constraints on the requests. Typically, a feedback controller is used to adjust the queue length such that the length equals its reference [10, 27, 28, 29, 30]. Ways of manipulating the queue length include changing the admission rate of the requests. Namely, by admitting more arriving requests the queue length increases, thus, increasing the latency time of the requests.

Controlling the execution times of tasks is important in real-time systems with constraints on energy consumption. Efforts have been carried out trying to reduce energy consumption in real-time systems, while preserving timely completion of tasks [31]. In this case execution times are monitored and the voltage and, thus, frequency of the CPU is varied such that the power consumption is reduced and tasks are executed in a timely manner. Hence, the sensor is used to measure the execution time of the tasks and the actuator is used to carry out the change in the voltage or the frequency of the CPU.

By studying the examples above, we note that there are many ways to implement sensors and actuators. This shows that the choice of a sensor and an actuator depends highly on the type of an application and its constraints. Control-related variables also vary depending on the application being controlled. Furthermore, controllers are implemented using a particular metric and a controlling algorithm, explicitly tuned for a specific application.

In the feedback-based policies mentioned in this section, traditional control algorithms such as PID, state-feedback, and Lead-Lag are used (details on these algorithms can be found in [32]). These algorithms assume that the controlled system does not change during run time, i.e., the behavior of the controlled system is time-invariant. Hence, the control algorithm parameters are tuned offline and may not be altered during run time. However, the behavior of some real-time systems is time-varying due to significant changes in load and/or execution times. This is addressed by employing adaptive control algorithms [33] where the behavior of the controlled system is monitored at run time and the control algorithm parameters are adapted accordingly. Hence, different types of controllers may be employed and, as such, there is a need for configurability of the controllers.

As can be observed, there are many different types of control algorithms that are applied to specific QoS management policies. The choice of a control algorithm depends on the control performance, run-time complexity, memory footprint, and adaptiveness.

3 Problem Formulation

As discussed in Sect. 2, a number of QoS management approaches have been explicitly developed to suit the needs of a particular application. Specifically, if feedback-based QoS management is considered, a QoS controller (see Fig. 1) is typically developed to control a certain type of metric, and it cannot be reused for another metric unless extensive modifications are made to the system and the controller. Furthermore, QoS policies used in existing approaches to real-time QoS management [5, 6, 8, 9, 10, 11, 12, 23, 24, 26, 27, 28, 29, 30, 31, 34, 35, 36] cannot be exchanged or modified separately from the system. Hence, QoS management is specific to a real-time application and a QoS management approach developed for one application cannot easily be reused in another application.

Additional limitation of these QoS approaches is that the architecture of the system is developed such that it is tightly integrated with QoS management, yielding in a system that has a fixed, monolithic, and nonevolutionary architecture. Modifications of QoS management require complex modifications of the code and the structure of the overall system. Moreover, current approaches do not enable taking existing systems without QoS management and adapting them to be used in an application with specific QoS needs. The trend in the vehicular industry, for example, is to incorporate QoS guarantees in vehicle control systems [15, 37]. In this case a cost-effective way of building QoS-aware vehicle control systems is to efficiently incorporate QoS mechanisms into already existing systems.

Software engineering research has shown that the implementation of QoS management should be reusable across applications and highly configurable [38, 39]. A number of software engineering approaches that provide configurable QoS management do exist. In these approaches QoS policies are encapsulated into aspects, developed independently of the application, and injected into the distributed middleware system when needed. In such approaches *qoskets* are frequently used as the means of encapsulating the code of a QoS policy [38, 39, 40]. Qoskets represent a set of QoS class specifications and their implementations. They are used for monitoring and adapting the system to, e.g., varying network loads. Separating QoS policies from the application and encapsulating them into reusable and exchangeable units has proven to be conducive to cost-effective development of software systems that can be more easily maintained and evolved [38, 39, 40]. Moreover, it has been shown that CoSMIC [41] is useful for facilitating automated system configuration and deployment [39, 41]. CoSMIC is a domain-specific tool based on the concept of model-driven architectures, developed to aid in configuration of QoS in the domain of distributed real-time CORBA-based middleware systems.

While software engineering approaches and tools for reusable and configurable QoS management [38, 39, 40] enable extending existing systems by adding qoskets, they assume a CORBA-based (including real-time versions) architecture. The domain of the tools developed for assisting in configuring QoS is also limited to CORBA-based systems [41]. This type of evolution and tool support is not sufficient for a large class of real-time systems as most of the existing systems do not have CORBA architecture, e.g., vehicle systems, control systems, and mobile devices.

4 Enabling Configurable Real-Time QoS

In this section we first elaborate on the requirements that a system needs to fulfill in order to use a QoS aspect package for configuring QoS. Then, we present the main idea behind the concept of a QoS aspect package using an example of feedback-based QoS management, and outline the procedure for configuring QoS management. This is followed by a discussion on how tools can be used to support configuration of QoS.

4.1 System Requirements

The concept of a QoS aspect package as the means of configuring and extending an existing system applies both to the class of traditional (monolithic) real-time systems and to the class of component-based real-time systems, provided that they conform to the requirements we identified for each class.

Traditional real-time systems should: (1) be written in a language that has a corresponding aspect language; (2) have the source code of the system available; and (3) have the code structured in fine-grained pieces that perform well-defined functions, i.e., good coding practice is employed.

Component-based real-time systems should be built using "glass box" or "gray box" component models. These models imply that components have well-defined interfaces, but also internals are accessible for manipulation by the software developer. Examples include Koala [18], RTCOM [16], PBO [19], AutoComp [20], and Rubus-based component models [42].

In addition, both monolithic and component-based real-time systems should have functions for controlling the system load. Recall from Sect. 2.2 that there are multiple ways of controlling the load in the system, e.g., by changing the output quality of the tasks [25], modifying the period of the tasks [8, 11], admission control [22], and changing the frequency of the CPU [31]. This implies that the tasks must be scheduled by an online scheduler [7], e.g., earliest deadline first or rate monotonic [43] (in contrast to systems with, for example, a cyclic executive).

Given that a system conforms to the named requirements, the concepts and tools for configuration that we present next can be used for adding and configuring QoS management.

4.2 The QoS Aspect Package

At the design and specification level, the QoS aspect package prescribes a hierarchical QoS management architecture that, at the highest level, consists of two classes of entities: the QoS component and the QoS aspect types (see Fig. 2). The top level of the QoS management architecture can be used for any QoS management. In the remainder of the paper we discuss feedback-based QoS management, where components include a QoS actuator component, a feedback controller component, and a sensor component. The aspect types include the following types of aspects (see Fig. 2):

- QoS management policy aspects,
- QoS task model aspects, and
- QoS composition aspects.

The QoS component is defined as a gray box component implementing functions that provide an infrastructure for the QoS management polices. As such, a component has an interface that contains the following information: (1) functionality, in terms of functions, procedures, or methods that a component requires (uses) from the system, (2) functionality that a component provides to the system, and (3) the list of explicitly declared joinpoints where changes can be made in the component functionality. Explicitly declared joinpoints are especially useful in the context of producing families of systems with variations in their QoS management. One other motivation for explicitly declaring these joinpoints is to ensure that the evolution of the QoS aspect package and the system can be done as efficiently as possible. Namely, having access to the points in the component structure, the aspect developer when developing new QoS aspects does not necessarily have to have full knowledge of the component code to develop aspects quickly and successfully (as we elaborate further in our experience report in Sect. 6).

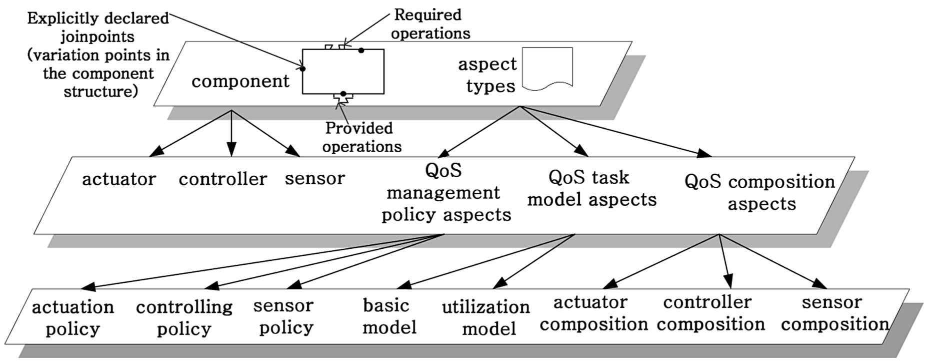


Fig. 2. A QoS aspect package at the specification level

The actuator is a QoS component and, in its simplest form, acts as a simple admission controller. It publishes a list of joinpoints in its interfaces where different actuator policies can be woven. Similarly, the feedback controller is by default designed with a simple control algorithm and identified joinpoints in the component structure, such that the controller can be extended to support more sophisticated control algorithms, e.g., adaptive control [33]. The sensor component collects necessary data and possibly aggregates it to form the metric representing the controlled variable. In its simplest form the sensor component measures utilization, which is commonly used as a controlled variable [8]. The sensor component publishes a set of joinpoints where it is possible to change the measured metric.

The QoS management policy aspects, namely actuator policy, controller policy, and the sensor policy, adapt the system to provide a variety of QoS management policies. Depending on the target application, the aspects modify the controller to support an appropriate QoS policy, and also change the actuator and the sensor component according to the choice of manipulated variable and controlled variable, respectively. For example, if the deadline miss ratio is to be controlled by changing the computation result quality of the tasks, then a QoS policy aspect measuring the deadline miss ratio is chosen. The actuator is modified by the aspect, exchanging the admission policy for an actuation mechanism where the quality of the tasks' results are modified.

The QoS task model aspects adapt the task model of a real-time system to the model used by QoS policies, e.g., a utilization task model needs to be used for a feedback-based QoS policy where utilization of the system is controlled. There can be a number of aspects defined to ensure enrichments or modifications of the task model, e.g., by adding various attributes to tasks, so that the resulting task model is suitable for distinct QoS or applications' needs. Concrete examples of task models are given in Sect. 5.3.

The QoS composition aspects facilitate composition of a real-time system with the QoS-related components: controller, actuator, and sensor. As we illustrate in Sect. 5.5, these aspects are rather simple and straightforward to implement.

Once a QoS aspect package is specified as described above, i.e., components and aspect types are identified, it needs to be populated with the actual implementations of aspects and components, which enable a system designer to develop a family of applications. Therefore, each family of applications would have its own QoS aspect package.

Now, an existing system that complies with requirements from Sect. 4.1 is configured for the specific real-time QoS management as follows.

– The architecture of the existing system is inspected and joinpoints are identified.
– The new context in which the system is going to be used, i.e., the application domain, is determined.
– Given the new usage context of the system, a suitable QoS policy consisting of the sensor policy, controlling policy, and the actuation policy, is identified.
– If the aspects implementing the QoS policy do not exist in the given QoS aspect package, then the corresponding aspects are defined, developed, and added to the package. Aspects use joinpoints in the existing system to inject the QoS policy. Similarly, the QoS aspect package has to be populated with a sensor component, controller component, and an actuator component, if these do not already exist in the package.

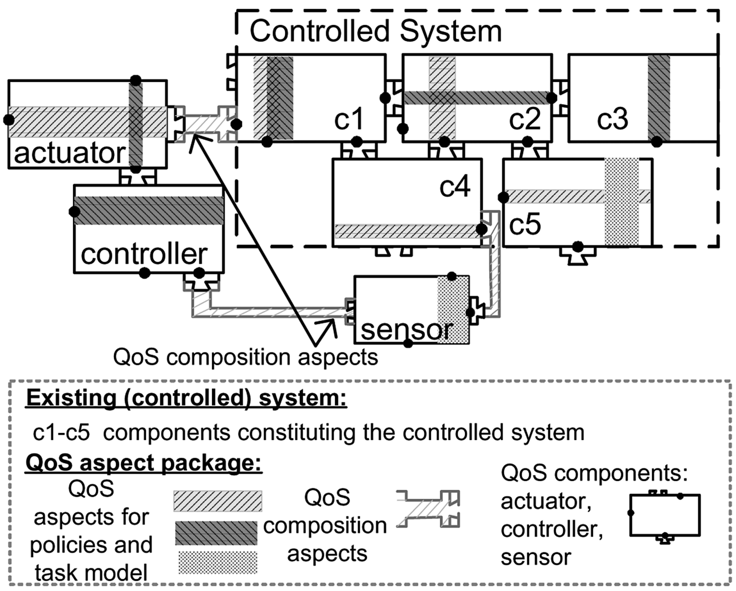

Controlled System
c1
c2
c3
c4
c5
actuator
controller
sensor
QoS composition aspects
Existing (controlled) system:
c1-c5 components constituting the controlled system
QoS aspect package:
QoS aspects for policies and task model
QoS composition aspects
QoS components: actuator, controller, sensor

Fig. 3. A real-time system where a QoS aspect package is used for configuring QoS management

– Aspects and components for a specific QoS configuration are chosen from the QoS aspect package and woven into the existing system.

A real-time system where a QoS aspect package is applied is shown in Fig. 3. In this figure, the controlled (existing) system is a component-based system that consists of gray box or glass box components $c1, \ldots, c5$, which have well-defined interfaces and internals accessible for modification (corresponding to the second type of real-time systems discussed in Sect. 4.1).[2]

QoS composition aspects from a QoS aspect package are used to add the sensor, controller, and actuator to the parts/components of the system where needed; these aspects are represented with gray dashed lines between component connections in Fig. 3. Additionally, QoS composition aspects offer significant flexibility in the system design as the feedback loop can easily be placed "outside" the system and between any components in the system by simply adding QoS composition aspects. QoS management policies are thus added to the system using appropriate aspects from the package. Moreover, QoS policies can easily be exchanged by adding or changing aspects within the QoS management policy type. Hence, a QoS aspect package ensures that QoS management policies are modifiable and configurable, depending on the application requirements.

[2] The explanation and the main points are the same as if the depicted controlled system would be a traditional monolithic real-time system conforming to the requirements listed in Sect. 4.1.

4.3 Tool Support

In this section we describe tool support for choosing relevant aspects and components from a QoS aspect package for a specific QoS configuration.

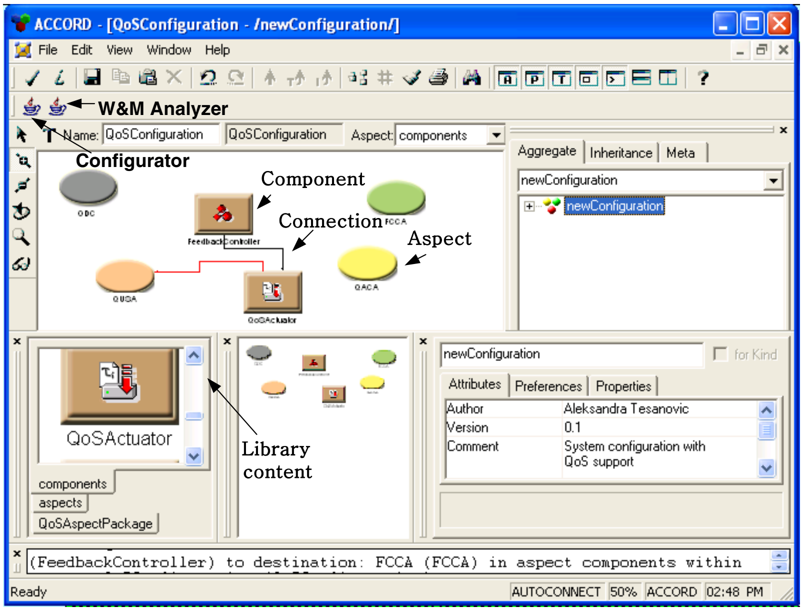
The tool supporting QoS configuration is a part of a tool called ACCORD modeling environment (ACCORD-ME). ACCORD-ME is, in turn, an integral part of the ACCORD tool development environment, which provides support for the development of reconfigurable real-time systems [44].[3] ACCORD-ME is a model-based tool, implemented using the generic modeling environment (GME), a toolkit for creating domain-specific modeling environments [45]. In general, the creation of a GME-based tool is accomplished by defining metamodels that specify the modeling paradigm (modeling language) of the application domain. In our case, based on the content of the QoS aspect package, modeling paradigms for QoS management are defined for ACCORD-ME. QoS modeling paradigms are given as UML diagrams and they define the relationship between components and aspects in the package, their possible relationship to other components of the system, and possibilities of combining components and aspects into different configurations. Note that a modeling paradigm needs to be specified for each new domain, i.e., each family of real-time systems.

The GME environment also enables specifying multiple tool plug-ins, which can be developed independently of the environment and then integrated with the GME to be used for different modeling and/or analysis purposes. Exploiting this, ACCORD-ME is developed with a number of subtools, namely the configurator, memory and worst-case execution time (M&W) analyzer, and formal verifier (see Fig. 4).

Next we explain the configurator part of ACCORD-ME and its role in configuration of QoS management. We omit detailed description of ACCORD-ME analysis tools (M&W analyzer and formal verifier) and refer the interested reader to [44].

When configuring QoS management of a system, the inputs to ACCORD-ME are the QoS requirements of the application that are placed on the system. QoS requirements can include specification of the controlled and manipulated variables, and the parameters characterizing the task model. The requirements can also be QoS policies that need to be enforced by the system, e.g., the utilization policy, the deadline miss ratio policy, and dynamic adaptation of the control algorithms. Based on the requirements and the predefined QoS modeling paradigms, the system developer can activate the configurator subtool to get adequate support for choosing relevant aspects and components from the QoS aspect package. To suggest a set of appropriate aspects and components, the configurator compares the requirements with the available configurations. Based on this analysis, configurator suggests the components and aspects that are found in suitable configurations.

[3] The name ACCORD-ME originates from the name of the approach for aspectual component-based real-time system development (ACCORD) [16] for which this tool was initially developed.


ACCORD - [QoSConfiguration - /newConfiguration/]
File Edit View Window Help
W&M Analyzer
Configurator
Component
Connection
Aspect
Library content
QDC
FeedbackController
FCCA
QoSA
QoSActuator
QACA
Aggregate Inheritance Meta
components
aspects
QoSAspectPackage
Attributes Preferences Properties
for Kind
Ready
AUTOCONNECT 50% ACCORD 02:48 PM

Name: QoSConfiguration QoSConfiguration Aspect: components

newConfiguration

Author	Aleksandra Tesanovic
Version	0.1
Comment	System configuration with QoS support

```
(FeedbackController) to destination: FCCA (FCCA) in aspect components within
```

Fig. 4. The editing window of ACCORD-ME, which is a part of the ACCORD Development Environment

The configurator provides three levels of support, expert, configuration, and requirement-based, based on the expertise and preferences of the developer. The expert option is used by developers familiar with the content of the QoS aspect package and all of its provided functionality and policies. The configuration-based option gives a list of possible QoS configurations of the system that can be created from the QoS aspect package. The requirement-based option provides the system developer with a list of possible choices of parameters in the feedback control structure, e.g., controlled and manipulated variables, from which the developer can choose a relevant subset suitable for a particular application. Thus, developers do not need to know the contents of the QoS aspect package to be able to configure QoS management.

After one of the above options is chosen, the configurator loads the relevant components and aspects into the ACCORD-ME editing area, where the designer can assemble the system by connecting the components and aspects, as shown in Fig. 4. Each aspect and component contains information stating with which it can be connected to form a functionally correct QoS management configuration.

The obtained QoS configuration can be analyzed by other subtools available in ACCORD-ME. Specifically, formal analysis can be performed to ensure that adding constituents of the QoS aspect package preserves already proven properties of the original system [46], thereby ensuring that functionality of the original system is preserved or modified in a well-understood way. For the purposes of formal analysis, a QoS configuration is transformed from a UML-based model to a timed automata-based model.

The output of ACCORD-ME is a configuration of a system that satisfies a specific QoS need of the application and that is functionally correctly assembled. The configuration, i.e., components, aspects, and their relationships, is stored in an XML file. This XML file is then fed to the tool of the ACCORD development environment, called the configuration compiler, which takes as input: (1) the information obtained from ACCORD-ME about the created QoS configuration, and (2) the source code of the needed aspects and components from the QoS aspect package (the source code of the system that is being controlled is also required). Based on this input, the configuration compiler generates a compilation file, which is used to compile source code of aspects and components into the final system. The configuration compiler also provides documentation about the generated QoS configuration that can later be used for maintaining the system.

5 Case Study: COMET Database

In this section we present a case study on a component-based embedded real-time database, called COMET. Initially, we developed COMET to be suitable for hard real-time applications in vehicular systems [15, 16]. Thus, the initial COMET implementation does not provide QoS guarantees. To adopt COMET to real-time systems with performance guarantees we developed a COMET QoS aspect package. In order to understand the choices made when developing different aspects for COMET QoS, we give an overview of COMET, and present the QoS management policies used for implementation of the QoS aspect package. We then discuss the data and transaction models used in various COMET configurations. We also describe a number of COMET QoS configurations and present the implementation of representative components and aspects. Finally, we demonstrate experimentally that COMET with the QoS extensions indeed provides the required QoS guarantees.

5.1 COMET Overview

The architecture of the COMET database consists of a number of components (see Fig. 5a): the user interface component, the transaction manager component, the index manager component, and the memory manager component. The user interface component provides a database interface to the application, which enables a user (i.e., an application using the database) to query and manipulate data elements. User requests are parsed by the user interface and are then converted into an execution plan. The transaction manager component is responsible for executing incoming execution plans, thereby performing the actual manipulation of data. The index manager component is responsible for maintaining an index of all tuples in the database. The COMET configuration containing the named components provides only basic functionality of the database, and this basic COMET configuration is especially suitable for small embedded vehicular systems [15].

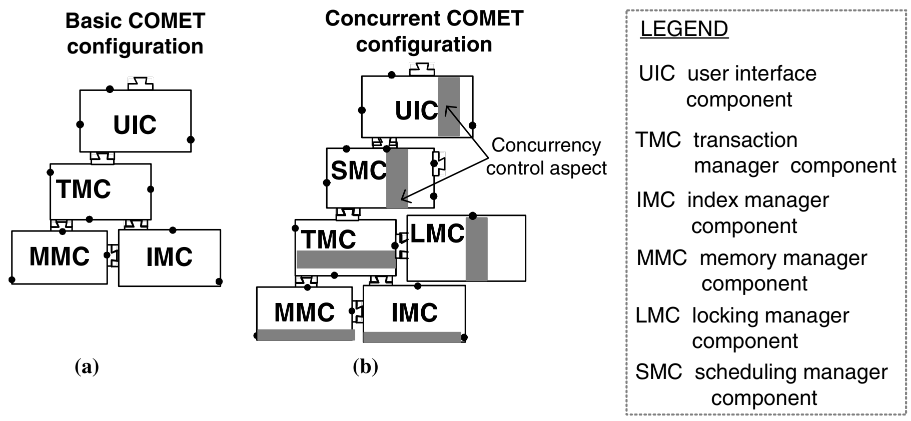

Basic COMET configuration
UIC
TMC
MMC
IMC
(a)
Concurrent COMET configuration
UIC
SMC
TMC
LMC
MMC
IMC
Concurrency control aspect
(b)
LEGEND
UIC user interface component
TMC transaction manager component
IMC index manager component
MMC memory manager component
LMC locking manager component
SMC scheduling manager component

Fig. 5. Basic and concurrent COMET configurations

Depending on the application with which the database is to be integrated, additional aspects and components can be added to the basic COMET configuration. For example, to enable concurrent access to the database two additional components, the locking manager component and the scheduling manager component, are needed (see Fig. 5b). The scheduling manager component is responsible for registering new transactions, i.e., tasks in a database system, to the system and scheduling them according to the chosen scheduling policy, e.g., earliest deadline first [47]. The locking manager component is responsible for obtaining and releasing locks on data items accessed by transactions. Concurrency control aspects, providing algorithms for detecting and resolving conflicts among transactions, can also be woven to the system. The concurrent COMET configuration is out of scope of this paper and we refer interested readers to [16].

Each component has been developed according to RTCOM [16] component model we developed previously. RTCOM is compliant with the QoS component discussed in Sect. 4.2 as it has interfaces where it publishes (provided and required) operations and declares joinpoints where variations can be introduced. For example, the transaction manager component executes transactions by executing an operation `getResult()` declared in its provided interface. The scheduling manager declares an operation `createNew()` in its provided interface, which registers transactions. Moreover, transactions in the scheduling manager component are scheduled using method `scheduleRecord()`, which is explicitly declared in the component interface as the joinpoint where variations in the structure of the scheduling manager component can be done.

5.2 QoS Policies

Given that we want to use the COMET database with applications that require performance guarantees, we need to choose some existing QoS policies and develop the QoS aspect package to integrate them into the database. Hence, in this section we give an overview of three feedback-based QoS management policies that we use in our case study; we found that these are especially suitable for ensuring performance guarantees in real-time systems. First we describe the feedback miss ratio control (FC-M) [8], where deadline miss ratio is controlled by modifying the number of admitted transactions. This is followed by a description of QMF [48], which is a QoS-sensitive approach for miss ratio and freshness guarantees used for managing QoS in real-time databases. Finally, we give a brief description of two adaptive QoS algorithms, the self-tuning regulator and the least squares regression algorithm [33].

FC-M uses a control loop to control the deadline miss ratio by adjusting the utilization in the system. We say that a transaction is terminated when it has completed or missed its deadline. Let $missedTransactions(k)$ be the number of transactions that have missed their deadline and $admittedTransactions(k)$ be the number of terminated admitted transactions in the time interval $[(k-1)T, kT]$. The deadline miss ratio,

$$m(k) = \frac{missedTransactions(k)}{admittedTransactions(k)} \tag{1}$$

denotes the ratio of transactions that have missed their deadlines. The performance error, $e_m(k) = m_r(k) - m(k)$, is computed to quantize the difference between the desired deadline miss ratio $m_r(k)$ and the measured deadline miss ratio $m(k)$. Note that $m(k)$ is a controlled variable, corresponding to $y(k)$ in Fig. 1, while $m_r(k)$ is a reference, corresponding to $y_r(k)$. The change to the utilization $\delta u(k)$, which we denote as the manipulated variable, is derived using a P-controller [32], hence, $\delta u(k) = K_\mathrm{P} e_m(k)$, where K_P is a tunable variable. The utilization target $u(k)$ is the integration of $\delta u(k)$. Admission control is then used to carry out the change in utilization.

Another way to change the requested utilization is to apply the policy used in QMF [48], where a feedback controller, similar to that of FC-M, is used to control the deadline miss ratio. The actuator in QMF manipulates the quality of data in real-time databases in combination with admission control to carry out changes in the controlled systems. If the database contains rarely requested data items, then updating them continuously is unnecessary, i.e., they can be updated on-demand. On the other hand, frequently requested data items should be updated continuously, because updating them on-demand would cause serious delays and possibly deadline overruns. When a lower utilization is requested via the deadline miss ratio controller, some of the least accessed data objects are classified as on-demand, thus, reducing the utilization. In contrast, if a greater utilization is requested then the data items that were previously updated on-demand, and have a relatively higher number of accesses, are moved from on-demand to immediate update, meaning that they are updated continuously. This way the utilization is changed according to the system performance.

The QoS management approaches presented so far in this section are not adaptive as they use linear feedback control and assume that the real-time system is time-invariant, which implies that the controller is tuned and fixed for that particular environment setting. For time-varying real-time systems it is beneficial to use adaptive QoS management that enables the controller in the feedback loop to dynamically adjust its control algorithm parameters such that the overall performance of the system is improved. Two representative adaptive QoS approaches are the self-tuning regulator and the least squares regression model [33].

5.3 Data and Transaction Model

We consider a main memory database model, where there is one CPU as the main processing element. We consider the following data and transaction models.

In the basic configuration of COMET we have a *basic data model* and a *basic transaction model*. The basic data model contains metadata used for concurrency control algorithms in databases. The basic transaction model characterizes each transaction τ_i only with a period p_i and a relative deadline d_i. However, QoS algorithms like FC-M, QMF, and adaptive algorithms require more complex data and transaction models that capture, e.g., metadata that express temporal constraints, such as mean interarrival and execution times.

In the *differentiated data model*, data objects are classified into two classes, temporal and nontemporal [49]. Nontemporal data constitutes data objects with no temporal requirements, e.g., arrival times and deadlines. For temporal data we only consider base data, i.e., data objects that hold the view of the real-world and are updated by sensors. A base data object b_i is considered temporally inconsistent or stale if the current time is later than the timestamp of b_i followed by the absolute validity interval avi_i of b_i, i.e., $currenttime > timestamp_i + avi_i$.

Both FC-M and QMF require a transaction model where transaction τ_i is classified as either an update or a user transaction. Update transactions arrive periodically and may only write to base (temporal) data objects. User transactions arrive aperiodically and may read temporal and read/write nontemporal data. In this model, denoted the *utilization transaction model*, each transaction has the following characteristics:

- the period p_i (update transactions),
- the estimated mean interarrival time $r_{\mathrm{E},i}$ (user transactions),
- the actual mean interarrival time $r_{\mathrm{A},i}$ (user transactions),
- the estimated execution time $x_{\mathrm{E},i}$,
- the actual execution time $x_{\mathrm{A},i}$,
- the relative deadline d_i,
- the estimated utilization[4] $u_{\mathrm{E},i}$, and
- the actual utilization $u_{\mathrm{A},i}$.

Table 1 presents the complete utilization transaction model. Upon arrival, a transaction presents the estimated average utilization $u_{\mathrm{E},i}$ and the relative deadline d_i to the system. The actual utilization of the transaction $u_{\mathrm{A},i}$ is not known in advance due to variations in the execution time.

[4] Utilization is also referred to as load.

Table 1. The utilization transaction model

Attribute	Periodic transactions	Aperiodic transactions
d_i	$d_i = p_i$	$d_i = r_{\mathrm{A},i}$
$u_{\mathrm{E},i}$	$u_{\mathrm{E},i} = x_{\mathrm{E},i}/p_i$	$u_{\mathrm{E},i} = x_{\mathrm{E},i}/r_{\mathrm{E},i}$
$u_{\mathrm{A},i}$	$u_{\mathrm{A},i} = x_{\mathrm{A},i}/p_i$	$u_{\mathrm{A},i} = x_{\mathrm{A},i}/r_{\mathrm{A},i}$

5.4 A Family of COMET QoS Configurations

We developed a QoS aspect package for COMET to enable the database to be used in applications that have uncertain workloads and where requirements for data freshness are essential, e.g., a new generation of vehicle control systems [3]. The aspects within the package are implemented using AspectC++ [50]. The COMET QoS aspect package consists of the actuator and controller components and the following aspects:

– QoS management policy aspects: actuator utilization policy, missed deadline monitor, missed deadline controller, scheduling strategy, data access monitor, QoS through update scheduling aspect, self-tuning regulator aspect, and adaptive regression model aspect;
– QoS transaction and data model aspects: utilization transaction model aspect and data differentiation aspect; and
– QoS composition aspects: actuator composition and controller composition aspects.

Figure 6 shows the family of COMET feedback-based QoS configurations. Each configuration is obtained by adding aspects and possibly components from the package to the previous configuration. The initial configuration is the concurrent COMET configuration. Table 2 indicates which elements of the QoS aspect package are used in different configurations.

The tool described in Sect. 4.3 can be used for specifying the QoS requirements of an application with which COMET is to be integrated. Once the requirements are specified, the tool suggests the subset of aspects and components from the aspect package that can be used for making a COMET QoS configuration. The tool decides on the relevant subset of components and aspects by cross-examining the requirements with the previously developed QoS modeling paradigms for the domain of embedded real-time databases.

To simplify the presentation, in the following we discuss only five distinct configurations that provide admission control, FC-M, QMF, self-tuning, and least squares regression QoS. Note however that depending on the chosen aspects and components from the package, the number of variants in COMET QoS family is higher (see Fig. 6).

The admission control configuration includes one component from the QoS aspect package, the actuator. The configuration also requires the actuator composition aspect to ensure adding the actuator to the controlled system, and the utilization transaction model aspect to extend the transaction model (see Table 2). This configuration is simple as it only provides facilities for admission control.

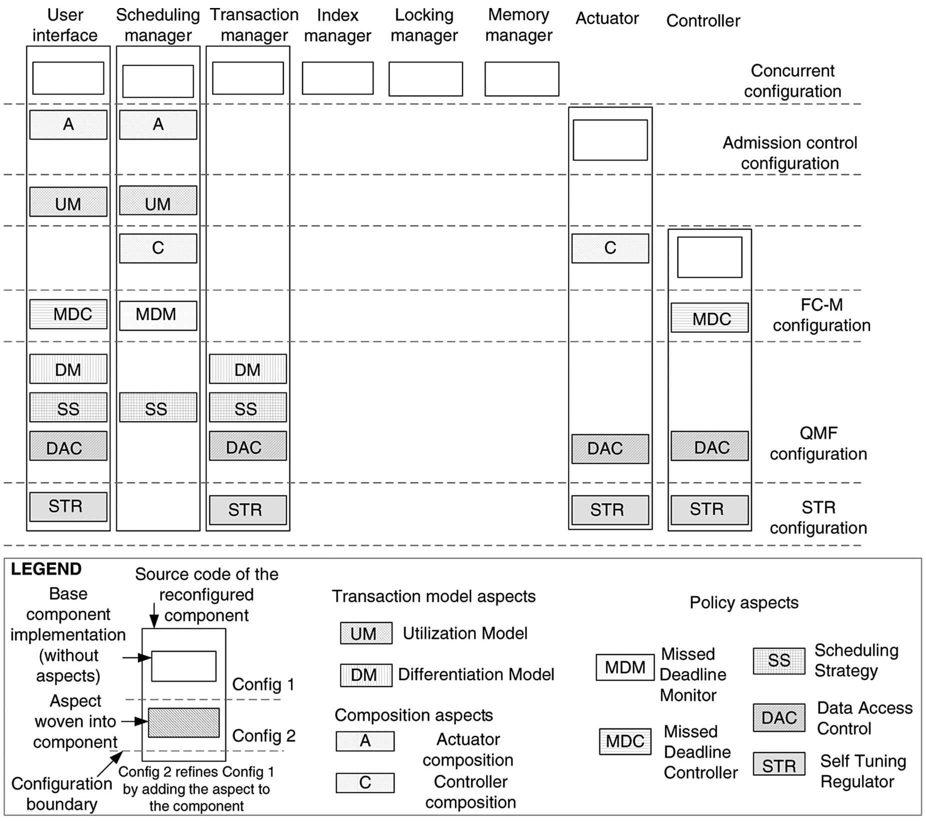


Fig. 6. Creating a family of real-time systems from the COMET QoS aspect package

The miss ratio feedback configuration (COMET FC-M) provides QoS guarantees based on the FC-M policy. The configuration includes the actuator and controller components and their corresponding composition aspects, the utilization transaction model aspect, the missed deadline monitor aspect, and the missed deadline controller aspect (see Table 2). These aspects modify the policy of the scheduling manager component and the controller to ensure that QoS with respect to deadline misses is enforced.

The update scheduling configuration (COMET QMF) provides the QoS guarantees based on the QMF policy. Here the data differentiation aspect and scheduling strategy aspect are used to further enrich the transaction model. Moreover, the data access monitor aspect is required to ensure the metric used in QMF, and the QoS through update scheduling aspect to further adjust the policy of the actuator to suit the QMF algorithm.

The self-tuning regulator configuration (COMET STR) provides adaptive QoS control where the control algorithm parameters are adjusted by using

Table 2. The COMET QoS aspect package constituents and their usage in various COMET QoS configurations

QoS aspect package		COMET configurations				
		Admission control	COMET FC-M	COMET QMF	COMET STR	COMET RM
Policy aspects	Actuator utilization policy	X	X	X	X	X
	Missed deadline monitor		X	X	X	X
	Missed deadline controller		X	X	X	X
	Scheduling strategy			X		
	Data access monitor			X		
	QoS through update scheduling			X		
	Self-tuning regulator				X	
	Adaptive regression model					X
Transaction model aspects	Utilization transaction model	X	X	X	X	X
	Data differentiation			X		
Composition aspects	Actuator composition	X	X	X	X	X
	Controller composition		X	X	X	X
Components	Actuator	X	X	X	X	X
	Controller		X	X	X	X

X in the table means that an aspect (or a component) is a part of a configuration
COMET FCM the miss ratio feedback configuration; *COMET STR* the self-tuning regulator configuration
COMET QMF the update scheduling configuration; *COMET RM* the regression model configuration

the self-tuning regulator. This aspect is added to the aspects and components constituting the COMET FC-M configuration to ensure the adaptability of the control already provided by the COMET FC-M configuration.

The regression model configuration (COMET RM) provides the adaptive QoS control where the control algorithm parameters are adjusted by using the least squares technique and the regression model. This aspect also requires all the aspects needed for the FC-M configuration to ensure adaptability of QoS management.

5.5 Implementation Details

To illustrate the way aspects and components can be implemented within an aspect package, in this section we elaborate on the implementation of the main constituents of the COMET FC-M configuration. Recall that the FC-M configuration includes the actuator and controller components, composition aspects for these components, as well as the utilization transaction model aspect, the missed deadline monitor aspect, and the missed deadline controller aspect.

The actuator is a component that, based on an admission policy, decides whether to allow new transactions into the system. Operations provided by the actuator are `admit()`, which performs the admission test, and `adjust()`, which adjusts the number of transactions that can be admitted. The default admission policy is to allow all transactions to be admitted to the system. This admission policy of the actuator can be changed by weaving specific QoS actuator policy aspects.

The controller is a component that computes the input to the admission policy of the actuator at regular intervals. By default, an input of zero is generated, but by using QoS controlling policy aspects various feedback control policies can be used. The controller provides only one operation, `init()`, that initializes the controller component. The controller calls `adjust()` operation of the actuator after computing the manipulated variable.

The *utilization transaction model aspect* augments the basic COMET transaction model so that it suits the utilization transaction model described in Sect. 5.3. This is done using an intertype declaration that adds new parameters to the basic model, e.g., estimated utilization $u_{\mathrm{E},i}$ and estimated execution time $x_{\mathrm{E},i}$.

```
1: aspect actuator_composition{
2:   // Insert actuator between user interface  andscheduler.
3:   advice call("bool Scheduler_CreateNew(...)") : around() {
4:     if (Actuator_Admit(*(scheduleRecord *)tjp->arg(0)))
5:       tjp->proceed();
6:     else
7:       *(bool *)tjp->result() = false;
8:   }
9: };
```

Fig. 7. The actuator composition aspect

The *actuator composition aspect* enables the actuator to intercept requests to create new transactions that are posed by the user interface to the scheduling manager component. This is done via an advice of type around which is executed when the operation `createNew()` of the scheduler manager is called (lines 3–8 in Fig. 7). Since this operation of the scheduler manager component is in charge of registering a new transaction to the system, the advice ensures that an admission test is made by the actuator before the transaction is actually registered (line 4). If the transaction can be admitted, transaction registration is resumed; the `proceed()` in line 5 enables the normal continuation of the joinpoint `createNew()` (an explicitly declared joinpoint in the scheduling manager component). If the transaction is to be aborted, then the around advice replaces the execution of the transaction registration in full and, thus, ensures that the transaction is rejected from the system (line 7).

The *actuator utilization policy aspect* shown in Fig. 8 replaces, via the around advice (lines 5–13), the default admission policy of actuator with an admission policy based on utilization (lines 9–12). The current transaction examined for admission in the system is denoted `ct` in Fig. 8.

```
aspect actuator_utilization_policy{
  // Add a utilization reference to the system
  advice "SystemParameters" : float utilizationRef;
  // Changes the policy of the actuator to the utilization
 advice execution("% Actuator_Admit(...)") : around() {
   // Get the current estimated total utilization
   totalUtilization = GetTotalEstimatedUtilization();
  // Check if the current transaction ct can be admitted
  if (utilizationTarget > totalUtilization + ct->utilization)
     { (*(bool *)tjp->result()) = true; }
       else
      {  (*(bool *)tjp->result()) = false; }
 }
```

Fig. 8. The actuator utilization policy aspect

The *controller composition aspect* facilitates the composition of the controller with all other components in the system by ensuring that the controller is properly initialized during the system initialization.

```
aspect missed_deadline_monitor {
  advice call("% Scheduler_CreateNew(...)") : after(){
       if (*(bool *)tjp->result()) { admittedTransactions++; }
  }
  advice call("% Scheduler_Completed(...)") : before(){
     ScheduleRecord *sr = (ScheduleRecord *)tjp->arg(0);
      _getTime(&currentTime);
      node = findNode(ActiveQueue_root, sr->id);
      if ((node != NULL) && (_compareTimes(&currentTime,
                                     &(node->data->deadline))))
        { missedTransactions++; }
  }
  advice call("% Scheduler_Aborted(...)") : before(){...
      admittedTransactions--;}
  advice call("% Scheduler_RejectLeastValuableTransaction(...)") :  after(){
      if (*(bool *)tjp->result()) { admittedTransactions--;}
  }
  advice call("% getTimeToDeadline(...)") && within("%
     getNextToExecute(...)") : after() {... missedTransactions++;}
}
```

Fig. 9. Missed deadline monitor aspect

The *missed deadline monitor aspect* modifies the scheduling manager component to keep track of transactions that have missed their deadlines, *missedTransactions*, and transactions that have been admitted to the system, *admittedTransactions*. This is done by having a number of advices of different types intercepting operations of the scheduling manager component that handles completion and abortion of transactions (see Fig. 9). For example, the advice of type after that intercepts the call to `createNew()` increments the number of admitted transactions once the transactions have been admitted to the system (lines 2–4). Similarly, before the transaction has completed, the advice in lines 5–12 checks if the number of transactions with missed deadlines should be incremented, i.e., before invoking the operation `completed()` of the scheduler manager component.

```
1:  aspect missed_deadline_control{
2:   // Initialize the new variables need for contrb
3:   advice call("% UserInterface_init(...)") : after() {
4:        SystemParameters *sp =
5:               (SystemParameters *)tjp->arg(0);
6:     if (*(bool *)tjp->result()) {
7:      missRatioReference = sp->missRatioReference;
8:      missRatioControlVariableP =
9:               sp->missRatioControlVariableP;
10:       ...
11:   }
12:   // Modify the calculation of the control output
13:   advice call("% calculateOutput(...)") : after(){
14:     missRatioOutputHm =
15:        calculateMissRatioOutput(Scheduler_GetDeadlineMissRatio());
16:    *((float *)tjp->result()) = missRatioOutputHm;
17:   }
18: }
```

Fig. 10. Missed deadline controller aspect

The *missed deadline controller aspect*, illustrated in Fig. 10, is an instance of the feedback control policy aspect and it modifies the scheduler manager component to keep track of the deadline miss ratio, using Eq. (1). The aspect does so with two advices of type after. One is executed after the initialization of the user interface (lines 3–11), thus, ensuring that the appropriate variables needed for controller policy are initialized. The other modifies the output of the controller to suit the chosen feedback control policy, which is deadline miss ratio in this case (lines 13–17).

5.6 Experimental Evaluation

In this section we present an experiment made on the COMET platform with and without the QoS aspect package. The goal of the experiment is to show that QoS management in COMET performs as expected and thereby show that, when adding the COMET QoS aspect package, we indeed achieve configurability in QoS management with required performance guarantees. It should be noted that we have performed several other experiments to show that we achieve the desired behavior under different COMET QoS configurations [51].

For doing the experiment we have chosen the following experimental setup. The database consists of eight relations, each containing ten tuples. Note that this relatively low data volume is acceptable for the experiments as the experimental results do not depend on the data volume but the load, i.e., number of transactions, that is imposed on the database. To that end, we ensured that a constant stream of transaction requests is used in the experiments. Update transactions arrive periodically, whereas user transactions arrive aperiodically. To vary the load on the system, the interarrival times between transactions are altered. The deadline miss ratio reference, i.e., the desired QoS, is set to 0.1.

The experiment is applied to the COMET FC-M configuration, where the load applied on the database is varied. This way we can determine the behavior of the system under increasing load. We use the behavior of the concurrent COMET configuration without the QoS aspect package as a baseline. For all the experiment data, we have taken the average of 10 runs, each consisting of 1,500 transactions. We have derived 95% confidence intervals based on the samples obtained from each run using the *t*-distribution [52]. We have found that the ten runs are sufficient as we have obtained tight confidence intervals (shown later in this section). Figure 11 shows the deadline miss ratio of concurrent COMET and COMET with the FC-M configuration. The dotted line indicates the deadline miss ratio reference. We vary the applied load from 0 to 130%. This interval of load captures the transition from an underloaded system to an overloaded system since at 130% applied load we capture the case when the system is overloaded. We refer to [5, 6] for extensive evaluations of how loads greater than 130% affect the performance of the system.

Starting with concurrent COMET, the deadline miss ratio starts increasing at approximately 0.85 load. However, the deadline miss ratio increases more than the desired deadline miss ratio and, hence, concurrent COMET does not provide any QoS guarantees. Studying the results obtained when using the FC-M configuration we see that the deadline miss ratio for loads $0.90, \ldots, 1.30$

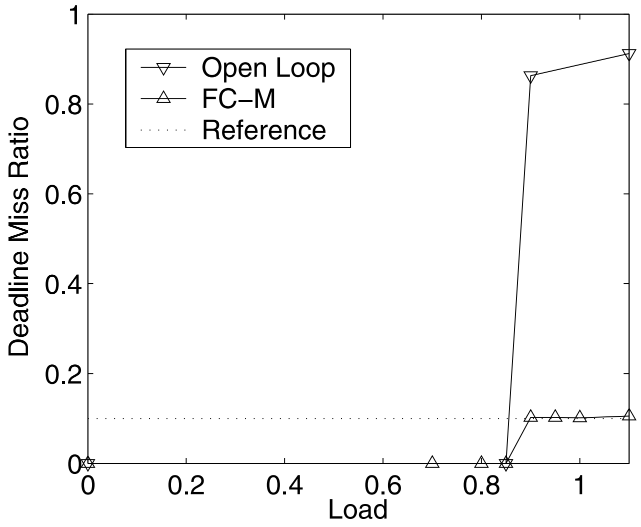


Fig. 11. Deadline miss ratio as a function of load

are $0.1025 \pm 0.0070, 0.1023 \pm 0.0027, 0.1012 \pm 0.0025, 0.1052 \pm 0.0030$, and 0.1082 ± 0.0011. In contrast with concurrent COMET, the added FC-M configuration manages to keep the deadline miss ratio at the reference, even during high loads. This is in line with our earlier observations where feedback control has shown to be very effective in guaranteeing QoS [5, 53, 54]. Hence, the results of our experiment show that the COMET FM-C configuration is able to provide QoS guarantees under varying load.

6 Experience Report

This section contains observations we made with respect to use of aspect-oriented software development in general and the QoS aspect package in particular for ensuring configurability and reusability of QoS management in real-time software.

Lesson 1: *Both aspects and components are needed in the QoS aspect package to ensure configurability.* We already observed in Sect. 1 that many QoS management approaches use a similar QoS infrastructure but provide distinct QoS policies, concluding that this is the reason why a QoS aspect package has both components (providing infrastructure) and aspects (providing QoS policies). Here we would like to reaffirm that without components in a QoS aspect package high configurability and reusability would be difficult to achieve. Namely, if all QoS management approaches were implemented only using aspects, each aspect would have to contain the necessary functionality of the infrastructure for QoS management. This is obviously not conducive to the development of families of real-time systems with distinct needs on the QoS configuration. Furthermore, the footprint of the system would be increased with additional code. Another solution would be to implement one basic aspect that would contain the functionality of an infrastructure and possibly a QoS policy. However, this option is not desirable as it implies dependencies among aspects that could lead to dependency problems in the implementation phase. This in turn could induce decreased configurability (as fewer aspects could be combined). Having components that provide functionality used by aspects decreases dependency issues and the overall memory footprint for the system, and increases reusability of aspects and configurability of QoS management.

Lesson 2: *Explicitly declared joinpoints in component interfaces lead to an efficient and analyzable product-line architecture.* We have observed that for developing a variety of system configurations using the same set of components in combination with various aspects, it is beneficial to have explicitly defined places in the architecture where extensions can be made, i.e., aspects woven. Although this approach restricts the joinpoint model of the aspect language, we obtain clear points in the components and the system architecture where variations can be made. The system developer extending and reconfiguring an existing system now does not need to have a complete insight into the system or component internals to perform successful reconfiguration. Therefore, in our components within a QoS aspect package we enforce that the places where possible extensions can be done are explicitly declared in the component interfaces. It is our experience, confirmed by the experiences of third-party COMET users, that these points are not difficult to identify in the design and implementation phases of component development. For example, the declared joinpoints for the actuator component and the controller component from the COMET QoS aspect package were straightforwardly determined in the design phase by taking into consideration a few possible policies that could be applied to these components.

Hence, relevant joinpoints in the components should be identified in the design phase of the QoS aspect package development with regard to possible policy variations. In development of the COMET database we have experienced that these points are relatively low in number and, once identified, are even suitable for aspects that are developed later on. Moreover, the explicitly declared joinpoints in the component code are desirable in the real-time domain. This is because they provide predefined places where code modifications can be done and, therefore, the system can be analyzed during the design phase to establish if it satisfies temporal constraints and has desired functional characteristics [16, 46]. We can conclude that by using the notion of a QoS aspect package we can efficiently develop an analyzable product-line architecture of a real-time system that has the ability to satisfy specified QoS needs.

Lesson 3: *There is a tradeoff between configurability, reusability, and maintenance.* Having a large number of aspects leads to high demands on maintainability of the aspects and the system, while fewer aspects lead to better maintainability at the expense of limiting configurability and reusability of aspects in the system. This further confirms previous observations [55] where a tradeoff between requirements for configurability and maintenance when using aspects in embedded software systems was identified. In the case of developing a QoS aspect package for COMET, our primary goal was to ensure reuse of QoS-related aspects and to increase system configurability. Therefore, we have chosen to separate concerns such that we have a great number of aspects that each can be used in multiple COMET configurations. For example, the missed deadline monitor aspect is decoupled from the missed deadline controller aspect (both are part of the QoS policy aspects and implement FC-M policy) to allow the missed deadline monitor aspect to be used in a combination with another controller policy. In the case when there is a focus on maintainability, the missed deadline monitor aspect and the missed deadline control aspect could be combined into one aspect that both monitors and controls the deadline misses. The same is true for the scheduling strategy aspect and the QoS through the update scheduling aspect that both implement parts of the QMF algorithm. We have chosen to have these in different aspects to enable them to be exchanged independently from the configuration and from each other.

Hence, if reusability and configurability is of foremost concern, as it is typically the case in the context of creating families of real-time systems, QoS policies should be decomposed into greater number of aspects; thus, trading maintainability for reusability and configurability. To deal with maintainability issues, an efficient way of organizing aspects and components for easier access and modification within a QoS aspect package is needed.

Lesson 4: *Aspects can be reused in various phases of the system development.* We found that aspects implemented in a QoS aspect package can be reused in various phases of the system development. Namely, due to the nature of QoS policies, one or several aspects constituting a policy normally control the load of the system and in some way monitor the system performance. Hence, in addition to reuse of these aspect in a number of QoS configurations, they can be reused in the testing and evaluation phase for performance evaluation and gathering statistics. As a simple illustration, the missed deadline monitor aspect within the COMET QoS aspect package is used in the design and implementation phase of the system as a part of a QoS management to implement a specific QoS policy, and is later used in the evaluation phase of the system development for performance evaluations (presented in Sect. 5.6).

Lesson 5: *Aspect languages are a means of dealing with legacy software.* Since we initially targeted the COMET database system to hard real-time systems in the vehicular industry [4], the programming language used for the development of the basic database functionality (described in Sect. 5.1) needed to be suited for software that already existed in a vehicular control system. Moreover, analysis techniques that have been used in the existing vehicle control system should be applicable to our basic database components. This lead to developing the COMET basic configuration using C programming language. Aspects provided an efficient means for introducing extensions to the system; we used the AspectC++ weaver since a weaver for C language [56] is not yet publicly available. Since existing real-time systems are typically developed in a non-object-oriented language such as C, aspects provide a great value for evolution of the system without reconstructing the code of the system.

Lesson 6: *Less is more when it comes to aspect languages for embedded and real-time systems.* When developing aspect languages for real-time systems operating in resource-constrained environments, the focus should be on providing basic aspect language features that facilitate encapsulating and weaving aspects into the code of components in the simplest and most memory-efficient way possible. We believe that minimizing memory usage should be of primary importance for aspect languages suitable for these types of systems. Given that most real-time computing systems are developed using non-object-oriented languages, the intertype declaration could be kept as simple as possible, e.g., allowing weaving of single members in structs. We also observe that due to the nature of many real-time operating systems, e.g., Rubus [57] and MicroC [58], the advice and pointcut model could be simplified. Namely, the pointcut syntax in most cases does not need to be elaborate as it is in current aspect languages (e.g., AspectJ and AspectC++). We have developed most of the COMET aspects using `call` and `execution` pointcuts, and occasionally `within`.

7 Summary

In this paper we have presented the concept of a QoS aspect package that enables reconfigurability in QoS management of real-time systems. A QoS aspect package facilitates the development of real-time systems with a flexible QoS management architecture consisting of aspects and components, where parts of the architecture can be modified, changed, or added depending on the target application's QoS requirements. Furthermore, QoS policies within a QoS aspect package are encapsulated into aspects and can be exchanged and modified independently of the real-time system. This improves reusability of QoS management and ensures applicability across many applications. Applying the concept of a QoS aspect package enables existing real-time systems, without QoS guarantees, to be used in applications that require specific performance guarantees. By exchanging aspects within the QoS aspect package one can efficiently tailor QoS management of a real-time system based on the application requirements. We have shown how the concept can be applied in practice by describing the way we have adapted the COMET database platform. Initially, COMET was developed without mechanisms for QoS guarantees, but by adding the QoS aspect package COMET has been extended to support a variety of QoS policies.

Acknowledgments

The authors would like to thank anonymous reviewers for valuable comments on the manuscript. This work is financially supported by the Swedish National Graduate School in Computer Science (CUGS) and the Center for Industrial Information Technology (CENIIT) under contract 01.07.

References

[1] Chen X., Cheng A.M.K. An imprecise algorithm for real-time compressed image and video transmission. In: *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)*, pp. 390–397, 1997

[2] Curescu C., Nadjm-Tehrani S. Time-aware utility-based resource allocation in wireless networks. *IEEE Transactions on Parallel and Distributed Systems*, 16:624–636, 2005

[3] Gustafsson T., Hansson J. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In: *Proceedings of Tenth IEEE Real-Time Applications Symposium (RTAS'04)*, IEEE Computer Society Press, pp. 182–191, 2004

[4] Nyström D., Tešanović A., Norström C., Hansson J., Bånkestad N.E. Data management issues in vehicle control systems: a case study. In: *Proceedings of the 14th IEEE Euromicro International Conference on Real-Time Systems (ECRTS'02)*, IEEE Computer Society Press, pp. 249–256, 2002

[5] Amirijoo M., Hansson J., Son S.H., Gunnarsson S. Robust quality management for differentiated imprecise data services. In: *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, IEEE Computer Society Press, pp. 265–275, 2004

[6] Amirijoo M., Hansson J., Gunnarsson S., Son S.H. Enhancing feedback control scheduling performance by on-line quantification and suppression of measurement disturbance. In: *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05)*, IEEE Computer Society Press, pp. 2–11, 2005

[7] Buttazzo G.C. Hard Real-Time Computing Systems, Kluwer Academic, Dordrecht, 1997

[8] Lu C., Stankovic J.A., Tao G., Son S.H. Feedback control real-time scheduling: framework, modeling and algorithms. *Journal of Real-Time Systems*, 23, 2002

[9] Lu Y., Saxena A., Abdelzaher T.F. Differentiated caching services: a control-theoretical approach. In: *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, IEEE Computer Society Press, pp. 615–622, 2001

[10] Parekh S., Gandhi N., Hellerstein J., Tilbury D., Jayram T., Bigus J. Using control theory to achieve service level objectives in performance management. *Journal of Real-Time Systems*, 23, 2002

[11] Cervin A., Eker J., Bernhardsson B., Årzén K. Feedback-feedforward scheduling of control tasks. *Real-Time Systems Journal*, 23, 2002, Special Issue on Control-Theoretical Approaches to Real-Time Computing

[12] Li B., Nahrstedt K. A control theoretical model for quality of service adaptations. In: *Proceedings of the Sixth IEEE International Workshop on Quality of Service*, pp. 145–153, 1998

[13] Stankovic J. VEST: a toolset for constructing and analyzing component based operating systems for embedded and real-time systems. In: *Proceedings of the First International Conference on Embedded Software, (EMSOFT'01), LNCS vol. 2211*, Springer, Berlin Heidelberg New York, pp. 390–402, 2001

[14] Tesanovic A., Amirijoo M., Nilsson D., Norin H., Hansson J. Ensuring real-time perfomance guarantees in dynamically reconfigurable embedded systems. In: *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing, LNCS vol. 3824*, Springer, Berlin Heidelberg New York, pp. 131–141, 2005

[15] Nyström D., Tešanović A., Nolin M., Norström C., Hansson J. COMET: a component-based real-time database for automotive systems. In: *Proceedings of the IEEE Workshop on Software Engineering for Automotive Systems*, pp. 1–8, 2004

[16] Tešanović A., Nyström D., Hansson J., Norström C. Aspects and components in real-time system development: towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1, 2004

[17] Engblom, J. Analysis of the execution time unpredictability caused by dynamic branch prediction. In: *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, pp. 152–159, 2003

[18] van Ommering R. Building product populations with software components. In: *Proceedings of the 24th International Conference on Software Engineering*, ACM, New York, pp. 255–265, 2002

[19] Stewart D.B., Volpe R., Khosla P.K. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23, 1997

[20] Sandström K., Fredriksson J., Åkerholm M. Introducing a component technology for safety critical embedded realtime systems. In: *Proceedings of the International Symposium on Component-Based Software Engineering (CBSE7)*, Scotland, Springer, Berlin Heidelberg New York, pp. 194–208, 2004

[21] Tešanović A., Amirijoo M., Björk M., Hansson J. Empowering configurable QoS management in real-time systems. In: *Proceedings of the Fourth ACM SIG International Conference on Aspect-Oriented Software Development (AOSD'05)*, ACM, New York, pp. 39–50, 2005

[22] Amirijoo M., Hansson J., Son S.H., Gunnarsson S. Generalized performance management of multi class real-time imprecise data services. In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pp. 38–49, 2005

[23] Abdelzaher T.F., Shin K.G., Bhatti N. Performance guarantees for web server end-systems: a control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13:80–96, 2002

[24] Koutsoukos X., Tekumalla R., Natarajan B., Lu C. Hybrid supervisory utilization control of real-time systems. In: *Proceedings of 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05)*, IEEE Computer Society Press, pp. 12–21, 2005

[25] Liu J.W.S., Shih W.K., Lin K.J., Bettati R., Chung J.Y. Imprecise computations. IEEE Comput, 82, 1994

[26] Amirijoo M., Hansson J., Son S.H. Algorithms for managing QoS for real-time data services using imprecise computation. In: *Proceedings of the Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA'03), LNCS vol. 2968*, Springer, Berlin Heidelberg New York, pp. 136–157, 2004

[27] Abdelzaher T.F., Stankovic J.A., Lu C., Zhang R., Lu Y. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23:74–90, 2003

[28] Sha L., Liu X., Lu Y., Abdelzaher T. Queuing model based network server performance control. In: *Proceedings of 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, IEEE Computer Society Press, 2002

[29] Abdelzaher T., Lu Y., Zhang R., Henriksson D. Practical application of control theory to web services. In: *Proceedings of American Control Conference (ACC)*, 2004

[30] Robertson A., Wittenmark B., Kihl M. Analysis and design of admission control in web-server systems. In: *Proceedings of American Control Conference (ACC)*, 2003

[31] Zhu Y., Mueller F. Feedback EDF scheduling exploiting dynamic voltage scaling. In: *Proceedings of the Tenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, IEEE Computer Society Press, pp. 84–93, 2004

[32] Franklin G.F., Powell J.D., Workman M. Digital Control of Dynamic Systems, 3rd edn., Addison-Wesley, New York, 1998

[33] Åström K.J., Wittenmark B. Adaptive Control, 2nd edn., Addison-Wesley, New York, 1995

[34] Kang K.D., Son S.H., Stankovic J.A. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transactions on Knowledge and Data Engineering*, 16:1200–1216, 2004

[35] Kang K.D., Son S.H., Stankovic J.A. Service differentiation in real-time main memory databases. In: *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE Computer Society Press, pp. 119–128, 2002

[36] Sharma V., Thomas A., Abdelzaher T., Skadron K., Lu Z. Power-aware QoS management in web servers. In: *Proceedings of 24th IEEE Real-Time Systems Symposium (RTSS)*, IEEE Computer Society Press, pp. 63–71, 2003

[37] Sanfridson M. Problem formulations for qos management in automatic control. Technical Report TRITA-MMK 2000:3, ISSN 1400-1179, ISRN KTH/MMK–00/3–SE, Mechatronics Lab KTH, Royal Institue of Technology (KTH), Sweden, 2000

[38] Schantz R., Loyall J., Atighetchi M., Pall P. Packaging quality of service control behaviors for reuse. In: *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*, Washington, DC, USA, IEEE Computer Society, pp. 375, 2002

[39] Wang N., Gill C., Schmidt D.C., Subramonian V. Configuring real-time aspects in component middleware. In: *Proceedings of the OTM Confederated International Conferences, LNCS vol. 3291*, Springer, Berlin Heidelberg New York, pp. 1520–1537, 2004

[40] Duzan G., Loyall J., Schantz R., Shapiro R., Zinky J. Building adaptive distributed applications with middleware and aspects. In: *Proceedings of the Third ACM International Conference on Aspect-Oriented Software Development (AOSD'04)*, New York, NY, USA, ACM, pp. 66–73, 2004

[41] Gokhale A.S., Schmidt D.C., Lu T., Natarajan B., Wang N. CoSMIC: an MDA generative tool for distributed real-time and embedded applications. In: *Proceedings of the First Workshop on Model-Driven Approaches to Middleware Applications Development (MAMAD 2003)*, pp. 300–306, 2003

[42] Isovic D., Norström C. Components in real-time systems. In: *Proceedings of the Eighth IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'02)*, Tokyo, Japan, pp. 135–139, 2002

[43] Liu C.L., Layland J.W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973

[44] Tešanović A., Mu P., Hansson J. Development environment for configuration and analysis of embedded real-time systems. In: *Proceedings of the Fourth International Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'05)*, 2005

[45] GME. The generic modeling environment. Institute for Software Integrated Systems, Vanderbilt University, `http://www.isis.vanderbilt.edu/Projects/gme/`, 2004

[46] Tešanović A., Nadjm-Tehrani S., Hansson J. Modular verification of reconfigurable components. In: *Component-Based Software Development for Embedded Systems – An Overview on Current Research Trends, LNCS vol. 3778*, Springer, Berlin Heidelberg New York, pp. 59–81, 2005

[47] Liu C.L., Layland J.W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973

[48] Kang K.D., Son S.H., Stankovic J.A., Abdelzaher T.F. A QoS-sensitive approach for timeliness and freshness guarantees in real-time databases. In: *Proceedings of the 14th IEEE Euromicro Conference on Real-Time Systems (ECRTS'02)*, IEEE Computer Society Press, pp. 203–212, 2002

[49] Ramamritham K. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1993

[50] Spinczyk O., Gal A., Schröder-Preikschat W. AspectC++: an aspect-oriented extension to C++. In: *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'02)*, Sydney, Australia, Australian Computer Society, 2002

[51] Björk M. QoS management in configurable real-time databases. Master's thesis, Department of Computer Science, Linköping University, Sweden, 2004

[52] DeGroot M.H., Schervish M.J. Probability and Statistics, 3rd edn., Addison-Wesley, New York, 2002

[53] Amirijoo M., Hansson J., Son S.H. Error-driven QoS management in imprecise real-time databases. In: *Proceedings of the 15th IEEE Euromicro Conference on Real-Time Systems (ECRTS'03)*, IEEE Computer Society Press, pp. 63–72, 2003

[54] Amirijoo M., Hansson J., Son S.H. Specification and management of QoS in imprecise real-time databases. In: *Proceedings of the Seventh IEEE International Database Engineering and Applications Symposium (IDEAS'03)*, IEEE Computer society Press, pp. 192–201, 2003

[55] Tešanović A., Sheng K., Hansson J. Application-tailored database systems: a case of aspects in an embedded database. In: *Proceedings of the Eighth IEEE International Database Engineering and Applications Symposium (IDEAS'04)*, IEEE Computer Society, pp. 291–301, 2004

[56] Coady Y., Kiczales G. Back to the future: a retroactive study of aspect evolution in operating system code. In: *Proceedings of the Second ACM International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, USA, ACM, pp. 50–59, 2003

[57] Articus Systems. Rubus OS – Reference Manual, 1996

[58] Labrosse J.J. MicroC/OS-II the Real-Time Kernel, CMPBooks, 2002

Author Index