Research Institute for Advanced Computer Science

Technical Report

Proceedings of the Second Dynamic Aspects Workshop (DAW05)

Robert E. Filman
Michael Haupt
Robert Hirschfeld

Technical Report 05.01

March 2005

Proceedings of the Second Dynamic Aspects Workshop (DAW05)

Robert E. Filman, RIACS

Michael Haupt, Darmstadt University of Technology

Robert Hirschfeld, DoCoMo Euro-Labs

Technical Report 05.01

March 2005

Chicago, Illinois

This volume represents the proceedings of the Second Dynamic Aspects Workshop, held at the Aspect-Oriented Software Development conference in Chicago, Illinois in March, 2005. This workshop identifies examples of useful dynamic aspect behavior, suggests appropriate linguistic structures for dynamic aspects, and discusses implementation techniques for dynamic aspects, such as shadow compilation, dynamically changeable hook points, and modifications required in the underlying execution environment.

This work was supported in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC 2-1426 with the Universities Space Research Association (USRA).

Table of Contents

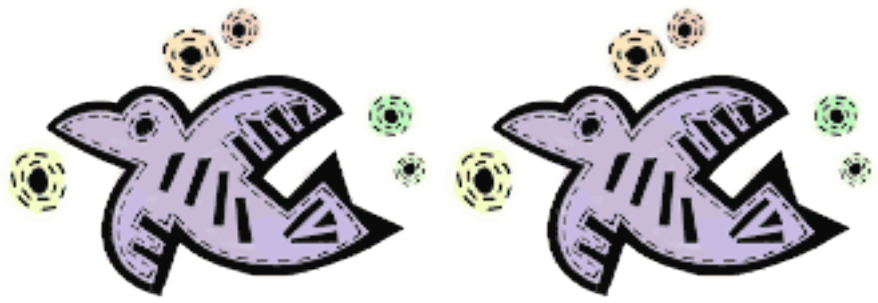
Workshop Organization

Workshop Organizers

- **Robert Filman**
  Research Institute for Advanced Computer Science/NASA Ames Research Center.
- **Michael Haupt**
  Darmstadt University of Technology
- **Robert Hirschfeld**
  DoCoMo Euro-Labs

Program Committee

- **Johan Brichau**
  Vrije Universiteit Brussel
- **Ruzanna Chitchyan**
  Lancaster University
- **Yvonne Coady**
  University of Victoria
- **Pascal Costanza**
  University of Bonn
- **Robert Filman**
  Research Institute for Advanced Computer Science/NASA Ames Research Center.
- **Stefan Hanenberg**
  University of Essen
- **Michael Haupt**
  Darmstadt University of Technology
- **Stephan Herrmann**
  Technical University Berlin
- **Robert Hirschfeld**
  DoCoMo Euro-Labs
- **Guenter Kniesel**
  Universities of Osnabrück and of Bonn
- **Crista Lopes**
  University of California, Irvine
- **Katharina Mehner**
  Technical University Berlin
- **Mario Südholt**
  Ecole des Mines de Nantes



•

Introduction

Join points are the locus of aspect and functional code interaction. Traditional aspect systems define join points in terms of the static structure of programs, such as syntactic method calls, method entry and field access.

AOSD has moved beyond static aspects. In some situations, it is desirable to invoke or change aspect behavior based on the dynamics of program execution. Such situations include changing behavior based on the call-stack context, co-occurrence of predicate triggers, concurrent thread status, or events in the underlying interpreter such as storage reclamation or process scheduling.

This workshop identifies examples of useful dynamic aspect behavior; suggests appropriate linguistic structures for dynamic aspects; and discusses implementation techniques for dynamic aspects, such as shadow compilation, dynamically changeable hook points, and modifications required in the underlying execution environment. The topics of the workshop include:

- Application scenarios for and applications of dynamic aspects
- Linguistic structures for dynamic aspects
- Implementation mechanisms for dynamic aspects
- Enabling technologies and environment support for dynamic aspects (e.g., debuggers, IDEs)
- Models for dynamic aspects
- Validation techniques for dynamic aspects
- Achieving the effect of dynamic aspects in conventional programming environments
- Challenges and research directions

State-Based Join-Points: Motivation and Requirements

[Position paper]

Nelis Boucké
Tom Holvoet
AgentWise, DistriNet, K.U.Leuven
Celestijnenlaan 200A,
B-3001, Leuven, Belgium
{nelis.boucke,tom.holvoet}@cs.kuleuven.ac.be

ABSTRACT
In developing a real-world complex application, we experience the major problem that complex concerns do not easily map onto low-level aspects with join-points based on fixed points in the program code. It is our observation that modularizing concerns and quantification are to be tackled at design-time, using suitable abstractions, with a translation to dynamic weaving at run-time. In particular, we argue that 'abstract states' of software entities (a concern consists of software entities) are a promising instrument for defining higher-level join-points for concerns. The specification and quantification of concerns in terms of abstract states typically result in dynamic weaving, i.e. depending on run-time states of the software entities.

Based on experience, we provide requirements for supporting concern modelling and quantification at design-time, as well as an initial sketch of an approach that we investigate in this perspective. The approach is based on a new type of higher-level join-points, called *state-based join-points* and serves as an example of the necessity for advanced dynamic (state-based) aspect weaving. The approach is motivated and illustrated through a scenario in a real-world application, namely decentralized control software for several automatic guided vehicles in an industrial transportation system.

Categories and Subject Descriptors
D.2.11, D.2.2 [**Software Engineering**]: Software Architectures, Design Tools and Techniques

General Terms
Design

1. INTRODUCTION
Research in Aspect Oriented Software Development (AOSD) focusses on developing systems with several non-orthogonal (e.g. overlapping, crosscutting and interacting) concerns. In the AOSD community great work has been done in building the technology to support separation of concerns, both statically [10, 13, 9, 11] and dynamically [1, 2].

In this position paper we argue that there is more to dynamic aspect approaches than technologies to support variety of implementation code join points. In particular, a software designer should be able to model concerns and explicitly describe quantification [6], which results in dynamic aspect weaving at run-time. To this end, implementation code join points do not provide an appropriate level of abstraction. Here, state-based join-points and quantification are defined, describing concern interference on a higher abstraction level.

We base the arguments and the proposal on our experience in developing a complex decentralized application for controlling automatic guided vehicles (AGVs) in a warehouse management system. For obvious reasons, we aim to apply the principles of separation of concerns throughout the design of the application. For example in the architectural design phase interesting concerns can be identified. However, these concerns do not easily map onto low-level aspects and join-points based on fixed points in the program code. Such a concerns are typically made up by a set of related software entities (e.g. objects or components). We observe that suitable quantification of these concerns can be achieved by considering 'abstract states' of software entities, rather than fixed points in program execution. Such an abstract state are more than the value of a attribute, it is relevant state on a higher abstraction level with a clear semantic meaning, described in terms of operations calls and attributes of software entities. A corresponding new type of higher-level join-points is introduced, called *state-based join-points*. The need for a dynamic join-point model is supported by publications on events in AOP ([7, 4]), the importance of a clear semantic meaning came forward in [15]. High-level quantification based on state-based join-points obviously results in a highly dynamic run-time system, requiring dynamic weaving.

First, in sec. 2, the necessity of higher-level concern quantification is motivated by pointing out a concrete problem scenario that we encounter in a real-world application of AGVs. Next, in sec. 3, an initial list of requirements and an initial sketch for design-level concern modelling and quantification is presented, based on our experience with this application. Finally, in sec. 4, we conclude with a number of open issues and links towards future work.

2. REAL-WORLD APPLICATION: AUTOMATIC GUIDED VEHICLES

This section contains a description of the real-world application, the identification of some important concerns in the architecture and a more detailed problem statement.

2.1 Application

Currently, our group is active in a research project with an industrial partner to decentralize control in this transportation system using a multi-agent system (EMC^2 [5]). An AGV is an unmanned, computer-controlled transportation vehicle using a battery as energy source. AGVs are able to perform transportation tasks, consisting of picking up a load and transporting it to the destination. High-level functional requirements for the AGV system are: (1) allocating transportation tasks to individual AGVs; (2) performing those tasks; (3) preventing conflicts between AGVs on crossroads; and (4) charging the batteries of AGVs on time.

Multi-agent systems (MASs) provide an approach for solving software problems by decomposing a system into a number of autonomous entities (e.g. AGVs), embedded in an environment, which cooperate in order to reach the functional and non-functional requirements of the system [14]. In general, MASs form a family of software architectures [16].

2.2 Architectural concerns

The need for state-based join-points is identified during architectural design of an AGV system. Once the basic structure of the architecture is clear, there are several concerns (specific to this architecture) that are important for the designer of the system. The basic architectural structure used here is a MAS, decomposing the system in agents (AGVs) and an environment (the factory). In essence, a concern is 'an issue that is important for a stakeholder in the system'. Concerns specific to the basic structure of an architecture are denoted with the term *'architectural concerns'* and are 'issues, part of a specific basic architecture, that are important for a designer using this architecture'. A few architectural concerns have been listed here to illustrate our position:

Autonomy One of the fundamentals of an agent is that it operates autonomously and proactively. E.g. an AGV-agents decides for itself what it will do, e.g. perform a task, reload its battery. Autonomy is an architectural concern because the way it is filled in has an important influence on the remainder of the architecture.

Individual capabilities Each agent has a collection of capabilities it can perform. Both externally visible actions (like driving, picking or dropping down a load) and internal calculation (like calculating the shortest path) belong to this concern. The capabilities are an important factor in determining the internal structure of an agent, asking for attention on an architectural level.

Coordination A MAS is built using several agents and these agents have to coordinate to meet the overall goals. For example in the AGV system, agents coordinate their behavior to prevent collisions on crossroads. Coordination of entities is a non-trivial problem that asks for an architectural solution. It is not our intent to focus on coordination itself, but rather on how coordination can be combined with other concerns.

Clearly, these concerns cross-cut and overlap each other. For example, consider the concerns autonomy and individual capabilities. Autonomy has an important influence on how individual capabilities are filled in. Both concerns overlap and crosscut. From the designer point of view, it is good to keep these concerns separated throughout the remaining building process, following the philosophy of SoC.

2.3 Problem statement

A simple scenario of the AGV system is used to illustrate our position. Consider two AGVs approaching a crossroad, leading to a possible collision between them. To enforce that only one AGV at a time can enter the crossroad, coordinating the behavior of both AGVs is needed. From the AGV system designer's point of view, there are several Architectural concerns involved, we consider 'individual capabilities' and 'coordination' as an example. If an AGV is approaching a crossroad (with 'approaching crossroad' as abstract state) and if there is another AGV approaching this crossroad leading to a possible conflicting situation (with 'other approaching' as abstract state), the concern 'coordination' crosscuts the concern 'individual capabilities'.

It is very hard, if not impossible, to specify quantification of these concerns based on fixed points in the program code of software entities. Simply because the quantification is not dependent on an operation call or attribute value, but rather on an abstract state (e.g. 'approaching crossroad'). As stated before, such an abstract state is more then a value of one or two attributes, it is relevant state on a logical level with a clear semantic meaning, described in terms of operations calls and attributes of software entities. E.g. to determine if an AGV is approaching a crossroad, the position of the AGV and its driving direction are compared with an internal layout of the factory. This type of behavior can not easily be expressed with current dynamic aspect approaches using the classical type of join-point, leading towards a design where architectural concerns are not clearly separated.

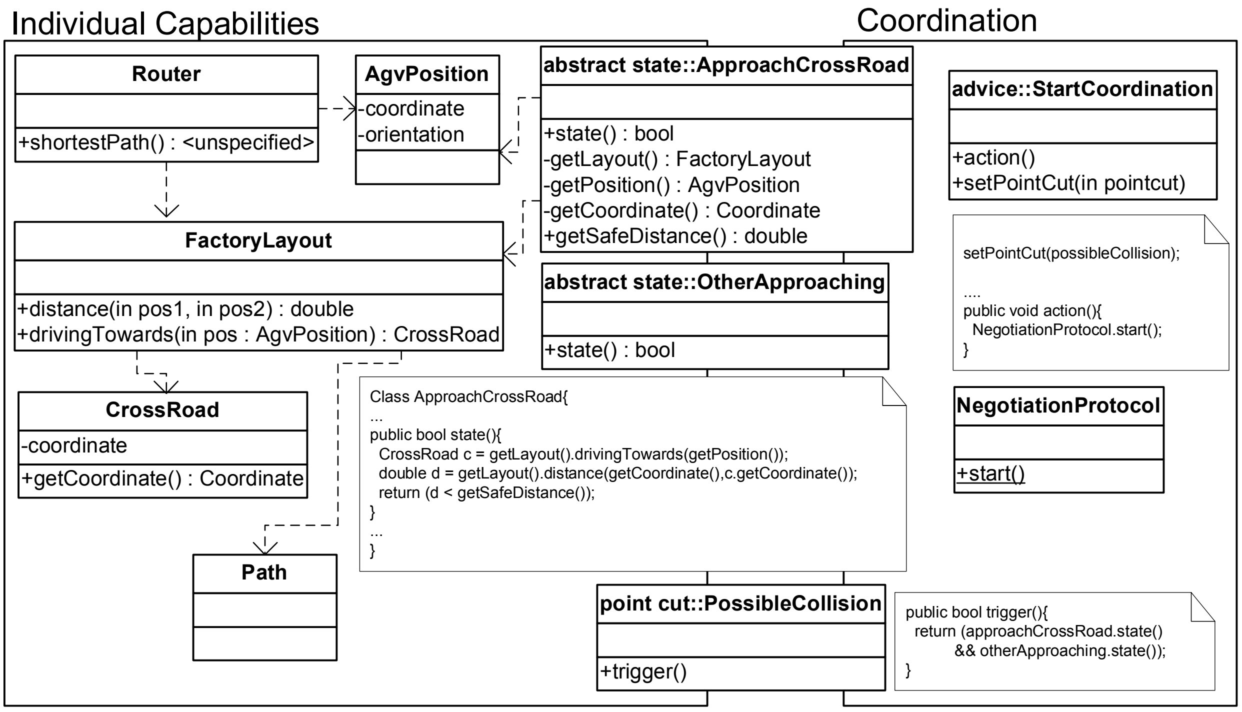
3. STATE-BASED JOIN-POINTS

It is our position that concern based approaches must support a level of abstraction that is higher than implementation code join-points. They must go beyond changing behavior based on the call-stack context, co-occurrence of predicate triggers or concurrent thread status (CFP of this workshop [3]), moving towards a higher level of abstraction.

3.1 Requirements for state-based join-points

An initial list of requirements for a mechanism based on state-based join-points is presented here, making up our research challenges for the next few years.

First, it must be possible to **define** a state-based join-point (in addition to implementation code join-points). For this definition, a rich language is needed to determine the abstract state.



Individual Capabilities
Coordination
Router
+shortestPath() : <unspecified>
AgvPosition
-coordinate
-orientation
abstract state::ApproachCrossRoad
+state() : bool
-getLayout() : FactoryLayout
-getPosition() : AgvPosition
-getCoordinate() : Coordinate
+getSafeDistance() : double
advice::StartCoordination
+action()
+setPointCut(in pointcut)
FactoryLayout
+distance(in pos1, in pos2) : double
+drivingTowards(in pos : AgvPosition) : CrossRoad
abstract state::OtherApproaching
+state() : bool
NegotiationProtocol
+start()
CrossRoad
-coordinate
+getCoordinate() : Coordinate
Path
point cut::PossibleCollision
+trigger()

```
setPointCut(possibleCollision);

....
public void action(){
  NegotiationProtocol.start();
}
```

```
Class ApproachCrossRoad{
...
public bool state(){
  CrossRoad c = getLayout().drivingTowards(getPosition());
  double d = getLayout().distance(getCoordinate(),c.getCoordinate());
  return (d < getSafeDistance());
}
...
}
```

```
public bool trigger(){
  return (approachCrossRoad.state()
        && otherApproaching.state());
}
```

Figure 1: A graphical representation of the Individual Capabilities concern.

A second requirement is that a **declarative quantification** language is needed, backing away from enumeration or pattern matching based on names [8, 12].

A third requirement is that a state-based join-point mechanism must respect the **obliviousness property** [6]. The obliviousness property states that the development of a software entity must be possible without being aware of the aspects that eventually will crosscut it. This implies that it must be possible to (1) define state-based join-points separately from the involved software entity; (2) after it is fully developed; and (3) without enhancements to it.

The fourth requirement is that both the definition of state-based join-points and the quantification of these join-points must be **translatable to a concrete execution** of a program. Thus, there is a need for a dynamic weaver, capable of working with state-based join-points.

The fifth requirement is **prevention of semantic mismatch conflicts**. Semantic mismatch conflicts between concerns are a typical problem in AOSD approaches [15]. Because every state-based join-point needs a clear semantic meaning, prevention of semantic mismatches is an important requirement.

3.2 Initial sketch

In this section we present a preliminary attempt towards supporting state-based join-points. The approach consists of three steps: seperation/modularization, defining state-based join-points and quantification. In this initial sketch, the approach is presented as a UML-diagram.

First, architectural concerns must be separated and modularized. Although modularization is an important issue, we only briefly describe it because our focus is on state-based join-points. In this initial sketch an UML-diagram corresponds with a concern and the software entities making up the concern are classes and objects. In Figure 1 an example is depicted, containing a few classes of the concern 'individual capabilities' and 'coordination' to illustrate our approach.

Next, the state-based join-points must be defined. First, abstract states are defined (e.g. `ApproachCrossRoad` and `OtherApproaching`). The `state` operation represents the condition for this abstract state. Next, a pointcut is defined, based on these abstract states (e.g. `PossibleCollision`). The `trigger` operation contains a condition when the quantification should be triggered (enumerated here, the final system should use a declarative description). The variables `approachCrossRoad` and `otherApproaching` represent the instantiations of the abstract states 'approaching crossroad' and 'other approaching' respectively. Every pointcut is associated with a particular concern and thus the concern definition is extended with a new, externally visible, join-point.

Finally, quantification is specified using state-based join-points. For this, we write an advice for a particular pointcut (e.g. `StartCoordination`). The `setPointCut` operation sets the pointcut for this advice, the `action` operation contains the action for this advice.

The results of these three steps serve as input for the dynamic weaver. Run-time weaving based on state-based join-points will probably be one of the most challenging problems for our future research. Especially, determining when state checks must be performed and thus where concerns should be woven is one of the main problems. Using current aspect technology could be useful in this step. Until know it is unclear what the exact implications are of run-time weaving and state checks. But it is certain that the amount of state checks and the checks itself will have an influence on the performance of the system.

Remark that using the terminology of [6], our initial sketch of state-based join-points is a event-based, publish and subscribe (EBPS), black-box AOP-mechanism. Differences with classical EBPS systems are that both the events (state-based join-points) and the subscription (quantification statements) are described separately. The main differences with quantification of events (in [7]) and Event-based AOP (EAOP [4]) are that our "events" are: (1) high-level state changes with a clear meaning, in contrast with primitive events, similar to fixed points in the program execution; and (2) defined explicitly and separately. In addition, [4] differs in its aspect definition (an aspect is a transformation on an event) and the way the weaving takes place (using a central execution monitor, catching every event and applying every aspect).

4. CONCLUSION

In this paper we outlined our position that dynamic aspect approaches must support an abstraction level that is higher than implementation code join-points and illustrated this in an industrial transportation system. High-level quantification based on state-based join-points has been introduced. This obviously results in a highly dynamic run-time system, requiring dynamic weaving.

What the exact implications of using state-based join-points are and how the join-points will translate to dynamic weaving are still important open issues. Especially, where exactly the weaving should take place and which current technology for dynamic aspect can be (re)used for defining dynamic weaving based on state-based join-points are important research tracks for the near future. A crucial step will be to determine a set of requirements for the dynamic weaver working with state based join-points and to consider the possible trade-offs.

5. ACKNOWLEDGMENTS

Thanks to the AgentWise taskforce, the EMC^2 members and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

6. REFERENCES

[1] Lodewijk Bergmans and Mehmet Aksit. Principles and design rationale of composition filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development.* Addison-Wesley, 2004. ISBN 0-32-121976-.

[2] I. Sommerville Chitchyan, R. Comparing dynamic ao systems. *Dynamic Aspects Workshop (at AOSD04).*, pages 23–36, 2004.

[3] DAW Organizing Committee. Call for papers of the dynamic aspects workshop (daw05). `http://aosd.net/2005/workshops/daw/cfp.html`. Checked on 13 Januari 2005.

[4] Rémi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (eaop). Technical report, Ecole des Mines de Nantes, 2002.

[5] Egemin and DistriNet. Emc^2: Egemin modular controls concept. IWT-funded project. From 1 March 2004, until 28 February 2006.

[6] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the workshop on Advanced Separation of Concerns, OOPSLA*, 2000.

[7] R. E. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *Foundations of Aspect-Oriented Languages (FOAL'02)*, 2002.

[8] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[9] IBM. Concern manipulation environment (CME). `http://www.research.ibm.com/cme/`. Checked on 4 januari 2005.

[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming, Springer-Verslag*, 1241:220–242, 1997.

[11] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[12] István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Declarative aspect composition. `http://trese.cs.utwente.nl/publications/files/0201Nagy.pdf`, 2004. Checked 13 januari 2005.

[13] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[14] H. V. D. Parunak. Agents in overalls: Experiences and issues in the development and deployment of industrial agent-based systems. *International Journal of Cooperative Information Systems*, 9(3):209–227, 2000.

[15] Peri Tarr, Maja D'Hondt, Lodewijk Bergmans, and Cristina Videira Lopes. Workshop on aspects and dimensions of concern: Requirements on, and challenge problems for, advanced separation of concerns. *Lecture Notes in Computer Science*, 1964:203–243, 2001.

[16] D. Weyns, A. Helleboogh, E. Steegmans, T. De Wolf, K. Mertens, N. Boucke, and T. Holvoet. Agents are not part of the problem, agents can solve the problem. In *Proceedings of the OOPSLA 2004 Workshop on Agent-oriented Methodologies*, 2004.

An application of dynamic AOP to medical image generation

Thomas Fritz[a,1], Marc Ségura[b], Mario Südholt[b], Egon Wuchner[c], Jean-Marc Menaud[b]

[a]Institut für Informatik, Gruppe PST
Ludwig-Maximilians-Universität
Oettingenstr. 67
80538 München, Deutschland
fritz@informatik.uni-muenchen.de

[b]Équipe OBASCO
EMN-INRIA, LINA
École des Mines de Nantes
4, rue Alfred Kastler
44307 Nantes cedex 3, France
{msegura,sudholt,jmenaud}@emn.fr

[c]Corporate Technology, SE2
Siemens AG
Otto-Hahn-Ring 6
81739 München, Deutschland
Egon.Wuchner@siemens.com

ABSTRACT

Medical image generation, *e.g.*, in computer tomographs, requires the use of sophisticated algorithms in a highly sensitive application domain. These algorithms are characterized (i) by a large variability to enable generation of different types of images and (ii) a strong need for dynamic reconfiguration to adapt image generation to individual patients. These two characteristics suggest the use of AOP techniques to manage variability which is akin to a crosscutting functionality and to enable dynamic reconfiguration.

In this paper we present three results related to AOP and medical imaging in the context of medical devices from Siemens AG, Germany: (i) a motivation why imaging software for medical tomographs can benefit from dynamic AOP, (ii) a case study of how system software for medical devices can be adapted using the Arachne system for dynamic AOP in C, and (iii) a detailed presentation of the underlying Arachne implementation and the design of its extension to C++.

1. MOTIVATION: MEDICAL IMAGE GENERATION AND AOP

Many medical devices (*e.g.*, magnetic resonance or computer tomography devices) require the generation of images based on measurements from the human body. The corresponding signal processing consists in the decomposition of the input signals yielded at certain points of time into signals corresponding to all the positions within a space cube representing the scanned 3-dimensional image (see Fig. 1). The signal specific to a particular position within the cube is characterized by its periodicity (*i.e.* cosine and sine waves), frequency, phase and amplitude.

[1]Part of this work has been done during the author's stay at École des Mines de Nantes.

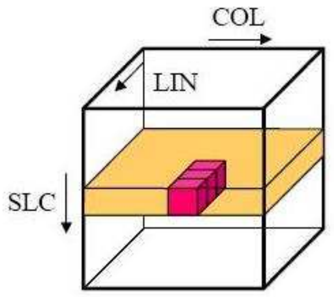


Figure 1: Space cube of measured signals associated with a human body part

The mathematical tool underlying such image generation tasks is Fourier transformation. In medical scanners, measurement data is typically stored within a raw data cube consisting of lines, columns and slices. A position in this cube is determined by its three dimensions and represents one measured signal. On the other hand this cube also represents the part of the human body which is examined. The state of human tissue at a position, is calculated by application of a sequence of basic image calculation steps to the measured raw data. These steps filter and adjust received signals, calculate images and post-process images. In the existing software for devices of Siemens AG the entire image generation transformations are constructed from approx. 60 different basic image processing functors. The set of valid transformations, *i.e.*, valid orderings according to which these basic functors may be combined, can be conceptually represented using a graph with one start and end node. The start node receives the measured signals and the end node yields a generated image.

In practice, the devices are used as follows. A concrete transformation needs to be configured before the start of a measurement for a patient. Currently, doctors execute one of a set of complete functor sequences generating an image for a patient, followed by other complete sequences, if necessary, for the same patient. However, based on corresponding customer requests, an evaluation is performed within Siemens medical devices unit of explicit support for dynamic adaptations of such functor sequences. With such techniques medical staff would be able to interactively adapt image generation during a measurement session depending on an initial set of calculated images. This would be highly useful in order to optimize the final images *w.r.t.* the individual patient. Such adaptations would enable, *e.g.*, using a higher resolution for parts belonging to tumors and are expected to speed up generation of the images taken for each patient.

During execution, code is executed corresponding to sequential and parallel functor sequences, the latter implementing, *e.g.*, calculations of different parts using different resolutions. This means execution can be represented by another graph, henceforth called the functor graph, which is a subgraph of the graph of all valid transformations discussed above.

Adaptation of functor sequences constitutes a software engineering problem that has three main characteristics:

1. The changes required by these adaptations are *scattered* over the functor graph and require a partial, but possibly rather comprehensive, transformation of the original processing graph.

2. The modifications to the functor graph during a measurement *cannot be anticipated*.

3. Modifications to the functor graph must be *reversible* so that new measurements can be performed based on parts of the image information previously generated.

Aspect-Oriented Programming (AOP) [11] has been instrumental in the adaptation of complex legacy software (for an example in the domain of operating systems programming, see [1]). An application of AOP techniques for the implementation of such medical image generation software seems therefore promising. In particular, an AO approach realizing this adaptation problem through (relatively) small changes to an existing code base seems advantageous, *e.g.*, concerning development effort and correctness validation, compared to approaches incurring larger changes, such as restructuring of the code base into an interpreter over the functor graph.

In this paper we present initial results of how to address the adaptation problems for image generation software for medical scanning devices from Siemens AG. We show how to apply the Arachne model and tool [6] for dynamic AOP in C in order to directly address the three above-mentioned characteristics: Arachne enables (i) the concise modular definition of the changes to the functor graph, (ii) dynamic modification of functor graphs without access to the source code, and (iii) unweaving of functor modifications. Furthermore, we give an overview of Arachne's implementation as well as its on-going extension to C++.

The remainder of this paper is structured as follows. Section 2 presents examples of the C++-based legacy code base used for a medical device from Siemens AG. In Sec. 3, we detail two fundamental transformations of the functor graph used for adaptation of image generation. Section 4 shows how such manipulations can be defined using Arachne. In Sec. 5 we give an overview of the architecture of Arachne and present its on-going implementation in C++. In Sec. 6 related work is discussed. Finally, Sec. 7 gives a conclusion and presents future work.

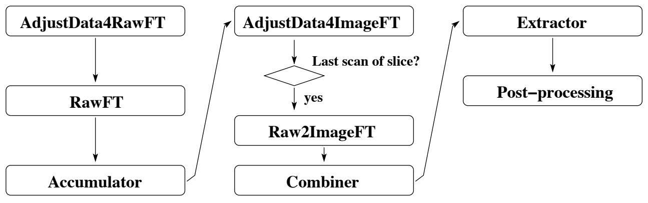


Figure 2: Basic steps for image generation

2. IMAGING CODE BASE

Let us first have a closer look at parts of the C++ code base used for medical image generation in Siemens devices, its overall structure and some specific generation steps.

Fig. 2 shows a sequence of basic processing steps for image generation. The functor `AdjustData4RawFT` (as well as the functors `Accumulator` and `AdjustData4ImageFT`) receives a line of measured data and makes some adjustments for the following Fourier transformations. The Fourier transformation `RawFT` works on the columns of the current line measurement of the current slice and derives some intermediary values for each column per receiver channel. (A channel corresponds to, *e.g.*, sensors situated at different locations of the medical device and receives its own signal during measurement.) The functor `Raw2ImageFT` takes the values of all these line calculations and computes a signal consisting of frequency and amplitude for each matrix position of the current slice. This way an image per receiver channel is calculated. The `Combiner` functor then takes the computed slice images corresponding to several receiver channels and calculates a weighted combination of them. The functor `Extractor` converts the complex values making up the image into corresponding human-readable information (*e.g.*, amplitude information) allowing conclusions about the kind of human tissue. Finally, `Post-processing` performs graphical manipulations, such as coloring of image parts, to the generated image.

The implementation of the image generation algorithms forming the basic steps are based on the cube containing raw data measurements introduced previously. On the code level, this cube does not have only three spatial dimensions but, in fact, up to 16 dimensions. For instance, its fourth dimension consists of the above-mentioned channels. Slice images can be calculated for each channel and combined afterwards. The following statement constructs such a multi-dimensional cube:

```
RawCube* cube =
  CubeFactory::create( LINE, 256, COLUMN, 256,
                       SLICE, 512, CHANNEL, 8,
                       ...);
```

Similarly, there is an `ImageCube` class allowing to store a multi-dimensional array of (intermediate or final) images.

Functors essentially implement algorithms iterating over the multi-dimensional data cubes. Each generation step requires access to data from a certain set of dimensions. As an example, the functor `RawFt` accesses the current line and slice and executes the Fourier transformation on each column for each receiver channel by initializing an iterator accordingly (see the iterator `rawFtIter` in lines 20–23 of List. 1).

```
1  class RawFT : public Functor {
2  public:
3    void addNextFunctor( Functor* );
4    FunctorList* getNextFunctors();
5    ...
6
7    virtual void computeScan( CtrlInfo& ctrl, CubeIterator& iter );
8  };
9
10 void RawFT::computeScan( CtrlInfo& ctrl, CubeIterator& iter ) {
11   // copy input iterator referring to raw data cube
12   CubeIterator rawFtIter( iter );
13
14   // set the cube dimensions and ranges the RawFT should work on
15   rawFtIter.init( COLUMN, 0, ctrl.getNumberOfColumns(),
16                   LINE, ctrl.getCurrentLine(), ctrl.getCurrentLine(),
17                   SLICE, ctrl.getCurrentSlice(), ctrl.getCurrentSlice(),
18                   CHANNEL, 0, ctrl.getNumberOfChannels());
19
20   // call RawFT
21   Imager::FT( iter, rawFtIter );
22
23   // call next Functors
24   for(int i, i<FunctorList.size(), i++){
25     this->getNextFunctor(i)->computeScan( ctrl, iter );
26   }
27 };
```

Listing 1: RawFT implementation skeleton

Functors inherit from a base class `Functor` as shown for `RawFT` in List. 1. This functor provides public methods `addNextFunctor` and `getNextFunctors` (lines 3, 4) to manage a list of functors following `RawFT` within a sequence of generation steps forming a transformation. These methods realize the functor graphs at runtime. The algorithm represented by the functor is implemented by the method `computeScan` (lines 7, 10–27). This methods first initializes the iterator `rawFtIter` and applies it (lines 12–21). Finally, all functors following `RawFT` in the current functor graph are called using the method `getNetFunctors` (lines 24–26).

3. ADAPTATION SCENARIOS

We now present two fundamental adaptations of the imaging process required to enable interactive control by the medical staff. Technically, these adaptations take the form of transformations of the graph defining the functor sequences which generate images from raw data.

The first scenario for interactive adaptation of the image generation process consists in *adjoining a new parallel functor chain* to an existing chain of the current functor graph, as illustrated by Fig. 3 (which shows an application of a transformation to a graph consisting of two parallel functor sequences). For instance, a doctor using a tomograph could find some indication of a tumor covering some part of the human body which is currently being scanned. Thus, he decides to examine the corresponding region further without loosing the currently calculated image information and without interference with other image parts. This can be done by adding a new sequence of functors performing a very detailed image calculation for the smaller body section in addition to the original calculation of the initial body part. Both chains are then executed in a pseudo-parallel fashion and the resulting images of both functor chains are combined to one image per slice.

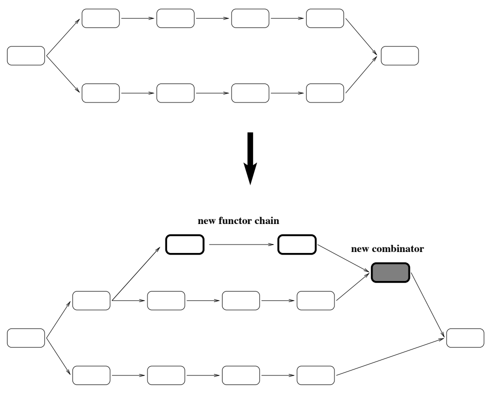


Figure 3: Adding a functor chain in parallel

The second adaptation scenario is to *replace a part of a functor chain* by another one as shown in Fig. 4. This scenario is used, *e.g.*, to modify the section of an image a transformation is applied to.

To conclude the discussion of image processing adaptations, note that functors affect several connection points of the original chain as illustrated by the two transformations above. Furthermore, many adaptations are performed during a tomography examination. Such adaptations may be applied to the initial functor chain as well as to chains which have been dynamically added previously. As a consequence the corresponding transformations are spatially and temporally scattered over the entire functor graphs.

4. APPLYING DYNAMIC AOP

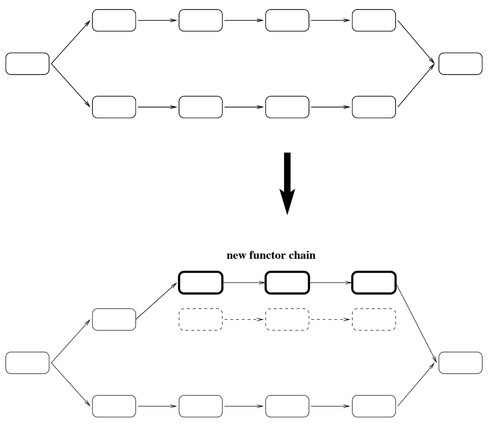


Figure 4: Replacing a functor chain

We now turn to the problem how to express the adaptation scenarios using the aspect language of Arachne.

Let us consider a sequence of functors, where each functor implements a method `computeScan`, as introduced in the previous section. Since the current version of Arachne supports only C, the functors have been mapped from C++ to C for the purposes of the evaluation presented in this paper. Note that this is quite simple because the functors, as exemplified by the code shown in the previous section, do not make extensive use of the object-oriented features of C++. As shown in List. 2 functors are mapped to C code by representing each functor class as a structure containing a pointer to a list of subsequent functors in the functor graph and a pointer to a function `computeScan`.

Arachne's aspect language. Let us first briefly consider Arachne's aspect language (see [6] for a more detailed presentation). Arachne provides analogues for C to AspectJ's main Java-oriented [10] features: pointcuts can be used in Arachne to match calls to C functions and match nested calls on the execution stack, a form of "cflow". (Note that Arachne also provides pointcuts others than those related to calls, *e.g.*, variable access join points. Since these are not used in this paper, we refer the interested reader to [6].)

Arachne distinguishes itself from most aspect languages by providing a construct for explicit sequencing on the language level, which is of the following form:

`seq(` *Prim*`,` *Prim* `[*], ... ,` *Prim* `[*],` *Prim* `)`

where *Prim* is a primitive aspect, such as

```
call(m(a, b)) then m'(b);
```

associating a primitive pointcut, which matches, *e.g.*, a call, to an action, typically a function call, which is to be executed instead of the matched call and that can itself call the original function. The construct is executed by first creating a new instance of the sequence aspect (with a fresh state) each time the first primitive aspect in the sequence matches. Then the other primitive aspects of the sequence are applied (repeatedly in case a star is used) as early as possible, *i.e.*, a primitive aspect a has priority over its predecessor: a is executed when the pointcut of a matches.

Adjoining a new functor sequence. A new functor sequence can be adjoined to an existing one using Arachne's aspect language by means of a single sequence aspect. Assume we want to add a chain of functors (`f3'` to `f4'`) along with a new functor combining the data of the parallel chains to a chain of functors `f1` to `f6`. An aspect that adds the new chain at functors `f3` and `f6`, and that can also be unwoven without any side effects is presented in List. 3.

The `computeScan` functions of `f1` and `f2` are executed as usual, but once `f3` is reached (lines 6–8 in List 3) in this sequence, the new subchain will be executed and the resulting data is stored temporarily. Then the original chain is executed and before `computeScan` of `f6` is executed, the image data is combined (lines 9, 10). The second and third step in the sequence aspect contain if-conditions to ensure that the `computeScan` methods work on the same image and iterator. This is necessary because there might be several identical chains to be matched that work on different iterators.

Replacing a functor sequence. Replacement of a functor sequence by another one can be expressed using a single sequence aspect, too. Assume we want to replace functors `f3` to `f5` with new functors `f3'` to `f5'` (cf. Fig. 4). The aspect in List. 4 achieves the replacement.

The `computeScan` functions of `f1` and `f2` will be executed as usual, but once `f3` is reached, the new subchain will be executed and the `computeScan` function of `f5'` will call the one of `f6` that then proceeds as usual. As in the preceding aspect, we use if-conditions to ensure that the steps in the sequence work on the same image and iterator.

Using Arachne, the functors replacing the ones in the original chain can be added dynamically.

5. ARACHNE: ARCHITECTURE AND IMPLEMENTATION

In this section we describe how Arachne enables dynamic weaving and unweaving of aspects into running legacy C applications. We present, in particular, a description of Arachne's structure which is improved w.r.t. the one in [6], a new detailed discussion of the consistency of Arachne's rewriting of native code, and a design of Arachne for C++ developed in order to apply Arachne (as future work) to Siemens AG's original code base, which has been developed over 8 years and whose use without modification is an important cost criterion for Siemens. By targeting Arachne directly to C++ code we obviate the need for the mapping to C introduced in the preceding sections.

5.1 Arachne's Architecture

Arachne's architecture is shown in Fig. 5: it is composed of three parts: the aspect runtime environment, a kernel manager, and an aspect compiler.

5.1.1 Arachne's runtime environment

The main component of Arachne's runtime environment is *Arachne's kernel dynamic link library* (DLL). As it is responsible for weaving aspects in the base program at runtime, it has to be able to rewrite the binary code of the base program and thus needs to be loaded in the same address space (as discussed in Section 5.1.2). Once the kernel DLL is loaded in the address space, it creates a thread in the base program

```
typedef struct functor *Functor;
struct functor{
    Functor* nextFunctors;
    void (*computeScan)(CtrlInfo* ctrl, CubeIterator* iter);
}

void computeScan_RawFT(CtrlInfo* ctrl, CubeIterator* iter){...}
Functor rawFT; rawFT.computeScan = computeScan_RawFT;
```

Listing 2: Mapping `RawFT` to C

```
CtrlInfo* ctrlNew; CubeIterator* iterNew;

seq(call(void computeScan_f1(CtrlInfo*, CubeIterator*)) && args(ctrl,iter);
    call(void computeScan_f2(CtrlInfo*,CubeIterator*)) && args(ctrl2,iter2)
     && if(ctrl == ctrl2) && if(iter == iter2);
    call(void computeScan_f3(CtrlInfo*, CubeIterator*)) && args(ctrl3,iter3)
     && if(ctrl == ctrl3) && if(iter == iter3)
      then executeOldAndNewChain(ctrl,iter);
    call(void computeScan_f6(CtrlInfo*, CubeIterator*))
      && args(ctrl6,iter6) && if(ctrl == ctrl6) && if(iter == iter2)
        then combineDataAndExecutef6(ctrl6,iter6); )

void executeOldAndNewChain(CtrlInfo* ctrl, CubeIterator* iter) {
    ctrlNew = ctrl.clone(); iterNew = iter.clone();
    executeNewChainf3'tof5'(ctrlNew,iterNew);
    computeScan_f3(ctrl,iter); // execute original chain
}

void combineDataAndExecutef6(CtrlInfo* ctrl, CubeIterator* iter) {
    combine(); // combines the data (ctrlNew,ctrl,iterNew,iter) and
               // stores result in ctrl and iter
    computeScan_f6(ctrl,iter); // execute last functor with
                               // combined data
}
```

Listing 3: Sequence aspect for adjoining a chain

```
seq(call(void computeScan_f1(CtrlInfo*, CubeIterator*)) && args(ctrl,iter);
    call(void computeScan_f2(CtrlInfo*, CubeIterator*)) && args(ctrl2,iter2)
     && if(ctrl == ctrl2) && if(iter == iter2);
    call(void computeScan_f3(CtrlInfo*, CubeIterator*)) && args(ctrl3,iter3)
     && if(ctrl == ctrl3) && if(iter == iter3)
      then replace(ctrl3,iter); )

void replace(CtrlInfo* ctrl, CubeIterator* iter) {
    executeNewChainf3'tof5'(ctrl,iter);
    // computeScanf5' will call computeScanf6 that
    // then proceeds as usual
}
```

Listing 4: Sequence Aspect for replacing a subchain

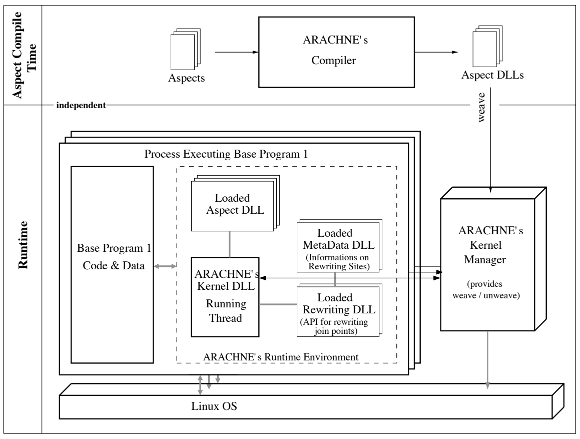

Aspect Compile Time
Runtime
independent
Aspects
ARACHNE's Compiler
Aspect DLLs
weave
Process Executing Base Program 1
Loaded Aspect DLL
Loaded MetaData DLL (Informations on Rewriting Sites)
Base Program 1 Code & Data
ARACHNE's Kernel DLL
Running Thread
Loaded Rewriting DLL (API for rewriting join points)
ARACHNE's Runtime Environment
ARACHNE's Kernel Manager (provides weave / unweave)
Linux OS

Figure 5: Arachne's Architecture

process, waiting on weaving and unweaving requests. By using a thread for processing the requests and weaving/unweaving, the execution of the base program does not have to be suspended. Upon reception of a weaving request, the Arachne kernel loads the corresponding aspect DLL. Then the kernel loads the rewriting DLLs required by the aspect if necessary. A *rewriting DLL* serves as an API that provides functions to rewrite/instrument one kind of join point. A rewriting DLL does not contain the information about the places to be instrumented in the binary code of the base program, *i.e.* the shadows. This information is stored in the meta data DLLs. *Metadata DLLs* are used to ensure the independence between Arachne's aspect system and the base program. They contain a mapping between the symbolic description of rewriting sites and the actual rewriting sites in the binary code of the base program. To rewrite join points, the rewriting DLL will request the Arachne kernel to load its corresponding metadata DLL. If this metadata DLL has not yet been loaded, the kernel will attempt to generate it by parsing the symbol tables and the object code contained in the base program executable file. Once the corresponding metadata DLL is loaded, the rewriting DLL will use the metadata to resolve the symbolic references in the aspect DLL and the binary code of the base program will be rewritten. For an unweaving request the kernel will instruct the rewriting DLLs referenced by the aspect to restore the original code before unloading the aspect DLL.

5.1.2 Arachne's kernel manager

Arachne's kernel manager serves as an intermediate between the user that wants to weave and unweave aspects into different applications, and aspect kernels that actually weave and unweave the aspects into a specific base program. The kernel manager is embodied in a set of shell commands for the weaving and unweaving of aspects. Once a (un)weaving command is issued by a user, the kernel manager communicates through sockets with the kernel of the specified base program to initiate the (un)weaving. In case a user wants to weave an aspect into a running application that does not yet contain an Arachne kernel, the kernel manager will "inject" an Arachne kernel in the application's address space before instructing it to weave the aspect. First, the Arachne kernel manager suspends the base program execution. Then it uses debugging APIs (`ptrace` [ref POSIX]) to rewrite the first bytes of the memory image of the base program and restarts the execution of the base program at the beginning of its memory image. Upon execution, the rewritten code loads the Arachne kernel DLL into the address space of the base program and sends a signal to the Arachne kernel manager. Upon reception of the signal, Arachne's kernel manager suspends the execution of the base program, restores the first bytes and finally triggers the continuation of the execution at the place where it interrupted the program execution beforehand. From this point on, the base program contains an Arachne kernel loaded in its address space that can handle weaving and unweaving requests. This dynamic "injection" technique does not require a new start of the application but might just lead to a suspension of the base program execution for up to 500ms.

5.1.3 Arachne's compiler

The compiler is a combination of a lexical analyzer written with Flex and a parser and program generator written with Bison. The compiler first translates an aspect written in the aspect language into a C source file that contains advice in executable functions and dynamic predicates, *i.e.* the residue. Once a join point is rewritten by the aspect runtime, it will automatically trigger the execution of the predicates and in case the aspect applies at the join point, run the appropriate function holding the aspect advice. In a second step, the compiler generates a compiled aspect DLL by using a regular C compiler (`gcc`). As the same aspect can be woven into two different base programs, the information about the rewriting sites, *i.e.* the shadows, of an aspect is not available at aspect compile-time, and thus all references to join points in a base program and rewriting DLLs are in a symbolic form. Once woven, an aspect DLL instructs the aspect runtime environment to instrument the base program at the appropriate places.

5.2 Nuts and bolts of binary code weaving

Arachne's dynamic weaving approach raises a number of issues that have to be considered. First, the base program code must remain executable and stay consistent at all times, and second, the resources (memory and sockets) used by the Arachne kernel should not interfere with the base program.

5.2.1 Consistency of the base program execution

Atomic rewriting of a site. As the rewriting is done at runtime without suspension of the base program execution, Arachne has to ensure that the rewriting does not interfere with the execution. A shadow in the binary code of the base program is rewritten by a jump to the residue of the aspect. However, such a jump instruction is too long for replacing it with one of the atomic memory write operations of the x86 processor. Therefore Arachne uses a rewriting strategy that ensures a consistent execution of the base program. It does so by first inserting a self-referencing loop (short enough to be written atomically) at the beginning of the instruction, so that in case the base program wants to execute the code at the rewriting site, it will just loop. (Note that this synchronization mechanism requires minimal execution time and is compatible with all common higher-level synchronization mechanisms.) Then Arachne writes the end of the jump instruction behind the loop before finally rewriting the loop instruction with the beginning of the jump instruction.

Rewriting consistency. In case a join point is composed of a set of assembly instructions, there might be jumps in the base program to an instruction in the set. To conserve the consistency of these jumps, a weaver may not rewrite the whole set of instructions as is done by Kerninst and Dyninst [7, 17, 3], or it might change the jump addresses beforehand. But latter one is nearly impossible to achieve at runtime where jump targets may be determined by an address held in a register or in memory. Therefore Arachne ensures to instrument only the first instruction of the assembly instructions belonging to the join point. At the same time, the instruction that will be rewritten by Arachne has to be big enough to fit a jump instruction. This however is ensured by the selection of the join points that Arachne provides.

Atomic weaving and unweaving. Arachne treats an aspect DLL as a collection of related aspects, potentially collaborating. To ensure a consistent execution of the base program, all rewriting sites addressed by one aspect DLL have to be woven atomically (the same counts for unweaving). This atomicity is provided by executing a dynamic check on the progress of the weaving before the residue and eventually the advice of an aspect are called. Thus only in case all addressed sites are rewritten, the aspect code is executed.

5.2.2 *Resource consumption*

Rewriting the binary code of the base program at runtime requires the Arachne kernel to share its memory with the base program. To allocate the memory for Arachne, the Linux function `mmap` is called within the base program execution. This function associates a portion of the caller address space with physical memory and treats it as freshly allocated. Since the base program does not know of the enlargement of its address space, it can not interfere with the memory used by the Arachne kernel[1]. In addition, to isolate the socket used by Arachne from the base program and thus avoiding interference between these two, the kernel thread is created through the Linux specific function `clone`, so that only the memory is shared but not the used sockets.

5.3 Extension of Arachne to C++

We conclude this implementation section by considering the design of how to extend Arachne for dynamic weaving of C++ applications, such that Siemens AG's code base for medical image generation can be used without previously mapping it to C.

Essentially, C++ is a typed object-oriented extension of C providing function overloading and overriding, instance variables and compile-time code generation facilities (*i.e.* `template`). To ensure proper interoperability between compilers, the compiled representation of a C++ file has been normalized [5, 15]. Except from the language features specific to C++, this standard closely follows the ANSI C specification. Therefore, the techniques used in Arachne to instrument C programs are directly applicable to C++ programs and only the features specific to C++ require further considerations and will be discussed in the following.

C++ implements function overloading by encoding the types of the signature in the function name. This encoding process is defined by the standard and allows tools such as `GNU nm` to retrieve the exact, source level name from the encoded, binary level function name. By using this property, Arachne will be able to properly handle overloaded functions.

Function overriding in C++ is implemented using `vtable`s [5, 15]. The C++ compiler translates the invocation of `virtual` functions into binary code that will first retrieve the address to the function to be executed from the `vtable` before actually executing it. In addition the C++ compiler holds each `vtable` as a global variable. Therefore, to trigger the execution of an action upon a virtual function, Arachne can just replace the addresses stored in the `vtable` by the address of the action.

Because of the standardization of the memory layout of object instances [5, 15], the techniques used by Arachne to track global and local C variables can easily be adapted to cover instance variables.

Finally, C++ compile-time generation facilities will not interfere with Arachne for C++. Arachne will however not be able to trigger advices on the metacomputation performed by `template` functions, since these computations are performed at compile-time and their results are inlined in the compiled executable. For the computations performed at runtime, *i.e.* template mechanisms used to parameterize a class or function, the necessary information is encoded within the binary names of the functions and variables and may be used by Arachne.

6. RELATED WORK

Image generation by medical devices is an active research field [12, 2, 18, 14]. Despite rapid evolutions, industrial medical software offers a fixed, closed set of features (functors). Hence, the state of the functor graph required for each image processing could be fixed at compile-time. However, the combinatorial explosion makes this approach unsuitable without appropriate tools.

Partial evaluation systems could be used to master such an combinatorial explosion. To be successful, such an approach would require a design where all image treatments will be derived from a most generic one. Ideally, in a partial evaluation approach, the application should use only a single functor graph capable of performing any image processing. Partial evaluation techniques and tools [8, 13] could then be used to automatically prune the unused functors from the functor graph depending on its use in the different parts of the program. But this generic and complete graph does not exist for Siemens AG's medical devices and tools for partial evaluation are rather unwieldy compared to, *e.g.*, Arachne.

To our knowledge, Arachne is the only dynamic aspect weaving system for C. AspectC [4] (for which no tool support is available) and AspectC++ [16] extend C and C++, respectively, by an aspect model very similar to AspectJ's [9]. Both of these provide static weaving and therefore do not meet Siemens AG's requirements of dynamic adaptability. Furthermore static approaches would require the implementation of sophisticated (and probably complex) undo-

[1]Contrary to `mmap`, regular memory allocators typically have some sort of side effect. For example, in case the Arachne kernel was built with the GNU `malloc` command, the base program could have used `sbrk(0)` to detect the memory allocations performed by the Arachne kernel.

mechanisms to support a notion of reversibility, similar to that built-in into Arachne.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented part of the existing code base of Siemens AG, Germany, for the generation of images by medical devices. We have presented some interactive adaptation scenarios arising in practice. We have also motivated that the corresponding transformation of the structure of image generation algorithms should benefit from AOP techniques. We have then outlined a solution realizing the adaptation scenarios in form of aspects using the aspect language of Arachne, a dynamic weaver for C programs. Finally, binary code weaving for C has been detailed and the design of an extension of Arachne for C++ has been presented.

There are several direct leads to pursue the work presented in this paper. Most prominently, the set of adaptation scenarios should be completed. Second, an implementation of the C++-version and the corresponding transformations is to be done. Finally, the relation between partial evaluation techniques and our AOP-based approach should be investigated.

8. REFERENCES

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an os kernel using temporal logic and aop. In *Proceedings of Automated Software Engineering (ASE'03)*, pages 196–204. IEEE, 2003.

[2] J. Ashburner and K.J. Friston. Why voxel-based morphometry should be used. *NeuroImage*, 14(6):1238–1243, 2001.

[3] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[4] Y. Coady, G. Kiczales, J.S. Ong, A. Warfield, and M. Feeley. Brittle systems will break – not bend: Can aspect-oriented programming help? In *Proceedings of the Tenth ACM SIGOPS European Workshop*, pages 79–86, St. Emilion, France, September 2002.

[5] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SG, editors. *Itanium C++ ABI.* CodeSourcery, November 2003. published on-line http://www.codesourcery.com/cxx-abi/abi.html.

[6] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *Proc. of 4th International Conference on Aspect-Oriented Software Development (AOSD'05).* ACM Press, March 2005. To appear.

[7] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A language and compiler for dynamic program instrumentation. In *IEEE Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 201–213, November 1997.

[8] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, Sep. 1996.

[9] G. Kiczales et al. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Berlin, June 2001.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, et al. An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *LNCS*, pages 327–353. Springer Verlag, Berlin, June 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, et al. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[12] G. Lohmann, K. Muller, V. Bosch, H. Mentzel, S. Hessler, L. Chen, S. Zysset, and D.Y. von Cramon. Lipsia a new software system for the evaluation of functional magnetic resonance images of the human brain. *Computerized Medical Imaging and Graphics*, 25(6):449–457, 2001.

[13] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2):217–251, May 2001.

[14] W.D. Penny, N. Trujillo-Bareto, and K.J. Friston. Bayesian fMRI time series analysis with spatial priors. *NeuroImage*, 2004. Accepted.

[15] Nathan Sidwell. A common vendor c++ abi. In *Proceedings of the Association of the C and C++ Users conference*, April 2003.

[16] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002.

[17] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

[18] D. Veltman, A. Mechelli, K.J. Friston, and C.J. Price. The importance of distributed sampling in blocked functional magnetic resonance imaging designs. *NeuroImage*, 17(3):1203–1206, 2002.

Dynamic Business Rules for Web Service Composition

María Agustina Cibrán
Vrije Universiteit Brussel
Pleinlaan 2
1050 Elsene
++32(2)629.29.64
mcibran@vub.ac.be

Bart Verheecke
Vrije Universiteit Brussel
Pleinlaan 2
1050 Elsene
++32(2)629.38.13
Bart.Verheecke@vub.ac.be

1. INTRODUCTION

The domains of many software applications are inherently knowledge-intensive. Examples of such domains are e-commerce, the financial industry, television and radio broadcasting, hospital management and rental business. Part of this knowledge is rule-based, typically representing knowledge about policies, preferences, decisions, advice and recommendations. The current software engineering practices result in software applications that contain implicit rule-based knowledge, which is tangled with the object-oriented core functionality. Nowadays, rule-based knowledge has become a hot topic and is also referred to as business rules [8, 12, 17].

On one hand we have conducted previous work in the field of business rules [5, 6] in which we observe the crosscutting nature of business rules connectors and the suitability of AOP for their implementation.

On the other hand, we are working on the design and development of the Web Services Management Layer (WSML) [7, 16], management middleware in between client applications and the web services. Such a management layer allows the definition of web services compositions to provide the functionality requested by client applications. Service compositions are expressed using a process-based language. An example of such a process-based language is WS-BPEL [2], a service choreography and orchestration language that allows the definition of business processes as interactions between web services in order to achieve a certain goal.

In this paper we focus on business rules present in the domain of web services compositions. We observe that composition business rules govern different aspects of the composition: *how* services need to be composed together, *how* suitable services can be discovered to fill in the roles of the composition, *which* services should be chosen at deployment time to carry out the activities of the process, and *how* the composition should adapt to the changing business environment. In this paper we only focus on the last kind of composition business rules.

We observe that support for explicitly defining business rules is inexistent or hardly supported in current business processes languages. Business rules are only implicitly expressed and their implementation is tangled with the core business process. As a consequence it is difficult to reason about and to evolve both parts independently, the core composition and the business rules. WS-BPEL, as a representative example of process-based languages, fails at providing support for explicitly representing the business rules in a clean and decoupled way.

Building on top of our previous work on business rules and AOP, we observe that composition business rules crosscut the service compositions and thus AOP can contribute to achieve their decoupling.

In the rest of the paper, we identify different categories of business rules that govern *how* the compositions should adapt to the changing business environment. In particular we focus on *dynamic business rules*, rules that take decisions on how to modify the core composition based on advanced patterns of execution history. We analyze the different kinds of AOP features needed for the realization of such rules. The JAsCo AOP language [13] is used as an example AOP technology. In particular, this paper shows how JAsCo stateful aspects [15] are suitable for realizing the identified dynamic business rules.

This paper aims to contribute by providing useful examples of dynamic aspect behavior, meaning in this context, the invocation or change of aspect behavior based on the dynamics of program execution.

2. BUSINESS RULES

The Business Rules Group defines a business rule as a statement that defines or constrains some aspect of a business. It is intended to assert business structure or to control the behavior of the business [14]. A significant characteristic of business rules is that they tend to change whenever the business policies they embody change, which is more often than the core application functionality does [1][11][17]. Examples of business rules are "*If a customer has purchased more than 20 books, then he or she becomes a frequent customer*" and "*If a customer is a frequent customer, then he or she gets a 10% discount*". Business rules are applied at events which are well-defined point in the execution of the core application functionality. Example events are "*before the price of a product is retrieved*" and "*after the customer has checked out*", at which the two examples rules are applied.

As business domains become more and more complex, it is fundamental to explicitly capture business processes and policies as business rules. The Business Rules Approach [17] states that it is crucial to implement them adhering to four objectives: separate business rules from the core application, trace business rules to business policies and decisions, externalize business rules for a business audience, and position business rules for change.

3. PROBLEM STATEMENT

Using a process-based language, a business process can be realized by specifying how different services interact to carry out a certain goal. In the definition of a business process a set of activities is identified. Each activity is associated with a role, which is mapped to a concrete web service or web services composition at deployment time in order to perform the functionality of the business process.

Currently, WS-BPEL [2] is one of the most promising process-based languages, candidate for standardization. WS-BPEL (Business Process Execution Language) for Web services is an XML-based language designed to enable task-sharing for a distributed computing - even across multiple organizations - using a combination of web services. Using BPEL, a programmer describes a business process that will take place across the Web in such a way that any cooperating entity can perform one or more steps in the process the same way. In a supply chain process, for example, a BPEL program might describe a business protocol that formalizes what pieces of information a product order consists of, and what exceptions may have to be handled. The BPEL program would not, however, specify how a concrete web service should process a given order internally.

We observe that the problems encountered when implementing business rules in object-oriented languages [6, 10] also arise in process-oriented languages. These problems occur due to the impossibility of achieving the following objectives in an oblivious way, i.e. without having to manually change the core application: (1) connect business rules to core application events which depend on run-time properties, (2) retrieve the needed information and make it available at those events when the rules are applied, (3) reuse the rules by connecting them at different events, (4) combine, prioritize and resolve rule interferences and (5) achieve all this preferably dynamically. The reason of this impossibility is that the rules are implicitly represented as *if-then* statements, and as result they appear tangled and scattered in the core application.

A business process written in WS-BPEL is one monolithic specification. It does not support the definition of business rules in a clean, modularized and reusable way and the specification of the rules gets tangled with the main process itself. Changes in the workflow due to changes in business requirements need to be done manually and invasively. The only support WS-BPEL offers is a limited kind of rules such as alternatives between tasks and repetitions based on business logic. Only limited workarounds or no support at all is provided for certain business rules such as time or order constraints between activities.

4. OUR APPROACH

We aim at defining service composition driven by explicit business rules. It is important to explicitly represent them since they tend to change faster than the core business processes. They are very volatile since they need to adapt to business requirements and thus should be decoupled. In order to achieve highly flexible service compositions we observe the need for defining rules explicitly and decoupled from the service composition itself. In the remainder of this section, we will illustrate how aspects are useful in this regard.

As mentioned before, in this paper we focus on composition rules that govern *how* the core composition needs to be adapted according to changing business knowledge. These rules will decide whether to add, replace, change or remove activities that are present in the core composition. In particular we consider ***dynamic business rules***, rules whose triggering events and/or conditions are based on the dynamics of the execution of the core application. To illustrate this kind of composition business rules, consider the following example application. It describes the business process of buying books online. In this context, customers of the shop can send in a quote request. If the customer is a valid customer then a quote is sent back and remains valid for a two-day period in which the customer can purchase an item. On the contrary, if the customer is not registered in the shop, then an error message is returned. During the time the quote is valid, the customers can place orders for buying books which are accepted by the shop. The next step in the business process is the payment of the ordered books. Depending on the results of this activity, the workflow continues with the shipping of the goods or with the refusal of the order, if the card payment is not authorized for instance.

In the following sections we identify different categories of dynamic business rules in the domain of service compositions. Examples of these categories are provided based on the introduced scenario. These rules are triggered depending on the dynamics of core compositions execution flow. Note that we are not presenting an exhaustive categorization, but the intention is to provide significantly different example categories of service compositions business rules that serve as a basis for the identification of AOP solutions.

Examples of a possible implementation in JAsCo will be presented for each category. JAsCo is an AOP language tailored for the component based context. JAsCo builds on top of Java and introduces two additional entities: aspect beans and connectors. An aspect bean is an extended version of a regular Java bean and allows describing crosscutting behaviour by means of a special kind of inner class, called a hook. Aspect beans are specified independently of concrete component types and API's, making them highly reusable. A connector on the other hand, is used for deploying one or more aspect beans within a concrete component context. In addition, connectors are able to specify explicit precedence and combination strategies in order to manage the cooperation among several aspects that are applicable onto the same join point. In addition, the JAsCo technology provides an extensive run-time infrastructure. Using this infrastructure, aspects remain first-class entities at run-time and dynamic aspect addition and removal becomes possible.

Sections 4.1, 4.2 and 4.3 discuss different example categories of dynamic business rules.

4.1 Category 1

Condition: occurrence of behaviors in a specific sequence. Additionally, checks on properties of business objects can be specified on the identified sequence

Action: addition of extra activity to the core process

This category of rules is considered when certain activities can be executed in different orders in the core composition. For instance, consider the case where activities a, b and c are part of the workflow, and the execution sequences a→b→c, b→c→a and c→b→a can occur in the core composition. The rule checks whether a particular path is followed, for instance the path a→b→c. The condition can additionally define extra checks on properties of business objects to be done at any of the points in the identified path. As an action the rule defines the insertion of an extra activity in the core composition, which needs to be added at an execution point posterior to the identified path on which the condition is checked.

Note that the activities a, b and c can be either consecutive (they occur one after the other) or not (other behaviors can be interleaved in between the identified activities in the path). Moreover, they can occur within the control flow of each other or outside. The path a → b means that the execution of activity b should be triggered after the execution of a is triggered, either within the control flow of a or after the execution of a is completed.

Example: suppose that the payment of the books can be done electronically by credit card or by cash. Consider a business rule that specifies that "*if not trustworthy customer and cash payment selected then check whether the payment has been received before shipping the products*". Then, before the actual shipping we first need to check whether the money was effectively received before dispatching the items. This extra check will determine whether to continue or not with the original workflow. This check is not done if the electronic payment branch was followed instead. Moreover, we only want to add this extra checking if the customer is registered in the system as *not trustworthy.*

In this example, the condition needs to check whether the sequence *a:login* → *b:cashPayment* is executed and check the trustworthiness of the customer when *a:login* is performed. The action identifies the addition of an activity that would verify the reception of the cash and act accordingly. This activity is added *before c:shipping*.

Solution: A naïve solution using current process-based languages would opt for adding variables in the core composition to keep track of whether the customer is trustful and the execution path that was taken (either the electronic payment or the cash one). Next, before allowing the shipment of the purchased products, those variables would need to be consulted in order to decide whether the extra checking for receiving the money is needed. However, this solution implies tangling the core composition with code for the implementation of the business rule. Moreover, the business rule results scattered in the composition and its identity is lost. Aspects help to avoid tangling the core composition as the variables and extra checking code would be encapsulated in a single module, the aspect, outside the main composition. However, keeping track of these variables inside the aspect code might be tedious and unclean.

```
class UnknownPaymentAspect {
  hook UnknownPaymentHook {
    UnknownPaymentHook(
      logIn(String username, String password),
      cashPayment(Order order),
      shippingOrder(Order order)) {
        start > logInCustomer;
        logInCustomer: execute(logIn) > payment;
        payment: execute(cashPayment) > shipping;
        shipping: execute(shippingOrder);
    }
    isApplicable logInCustomer(){
        return !store.trustworthyCustomer
                          (username, password);
    }
    replace shipping() {
      if (PurchaseDepartment.cashReceived(order))
        then proceed();
      else System.out.println
              ( "Shipping cannot proceed");
    }
  }
}
```

Code fragment 1 – Stateful aspect for payment check

JAsCo supports the definition of stateful aspects [15], aspects that are triggered on protocol history conditions. Stateful aspects allow achieving a cleaner implementation of this business rule, it is possible to specify the desired execution path of interest and plug in the crosscutting functionality at any time in the execution of the identified path. In this case, the path of interest is the execution of the sequence of activities *a:log-in* → *b:cashPayment* → *c:shipping*. The crosscutting code is the extra check for receiving the money that needs to be done before allowing the actual shipping of the purchased goods. The JAsCo stateful aspect is shown in code fragment 1.

Note that this aspect should be instantiated *perThread* to avoid inconsistencies when concurrent access.

This solution based on aspects that are triggered on protocol history conditions is much cleaner since the execution path of interest for the pluggability of the aspect behavior is explicitly captured. Conditions can be checked at the different transitions in the path as well as the extra/replacing behavior can be plugged at each transition. This solution is illustrated in Figure 1.

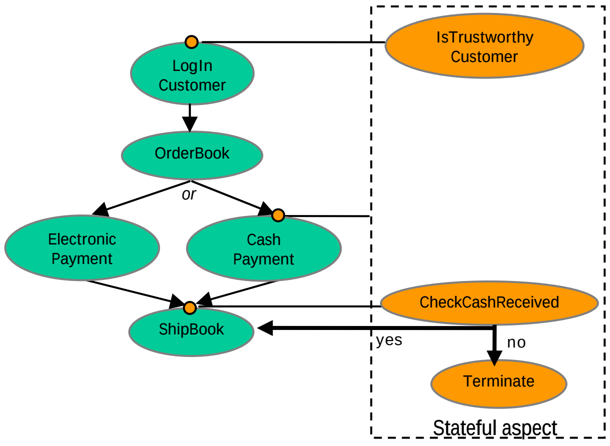


Figure 1 – Stateful aspect for adding additional payment checking to e-commerce business process

4.2 Category 2

Condition: checks on properties of business objects at a specific point in time during the process execution

Action: addition of extra activity in the process that needs to be applied on the execution of behaviors that occur in a specific sequence

At a certain point in the execution of the composition a condition that involves checking a certain property of a business object is checked. Based on the result of this checking, it is decided whether to add an extra crosscutting behavior. In order to execute, this behavior operates on different execution points of a defined sequence of events that occur later on, when future tasks in the execution path are executed.

Example: This example represents a conditional time constraint: "*If a purchase order is received and labeled as urgent then the whole process payment-shipping-delivery should occur within a maximum of 3 days. Otherwise, the customer is not charged for the purchase*".

Solution: In this example, the condition of the rule is checked at a certain point in time, which is the event *a:orderBook*. The result of this check will determine the pluggability of the *monitoringTime* activity. This extra behavior is applied on a sequence of events, *b:payment* → *c:shipping*, since it needs to measure the execution time of these two activities.

The solution using stateful aspect in JAsCo looks as follows:

```
class ConditionalTimeConstraintAspect {
  hook ConditionalTimeConstraintHook {
    ConditionalTimeConstraintHook(
     orderBook(CustomerId customer, BookId book,
                Priority priority),
     paymentOrder(Order order),
     shippingOrder(Order order)) {
       start > placeOrder;
       placeOrder: execute(orderBook) > payment;
       payment: execute(paymentOrder) > shipping;
       shipping: execute(shippingOrder);
    }
    isApplicable orderBook(){
        return priority.isUrgent();
    }
    before payment() {
      timestampbefore=System.currentTimeMillis();
    }
    after shipping() {
      timestampafter=System.currentTimeMillis();
      if (//time difference not OK)
        store.reimbourseCustomerOfOrder(order);
    }
  }
}
```

Code fragment 2 – Stateful aspect for timing constraint

Figure 2 illustrates this example.

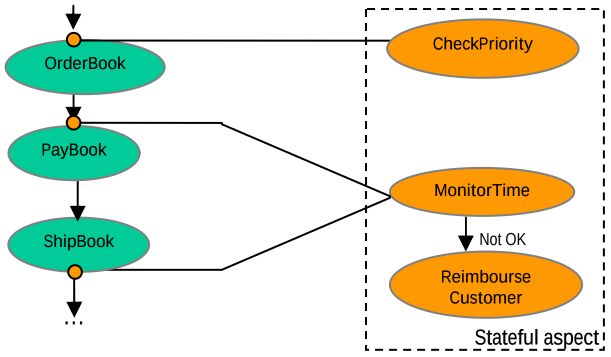


Figure 2 – Stateful aspect for adding additional time monitoring to payment-shipping activities in e-commerce workflow

4.3 Category 3

Condition: execution of behaviors in a sequence that is not allowed

Action: perform extra behavior

When parallel threads of tasks executions are possible in core composition, it might be desirable to restrict certain paths depending on a certain condition. A constraint rule restricts which paths are allowed, filtering out the not desired ones.

Example: An execution order constraint between tasks is enforced by a business rule. Imagine a scenario where the shipment and payment activities belong to different execution threads of parallel activities. Suppose a business rule that states that "*goods can only be shipped after payment has been received*". If the opposite occurs, then notify the manager of the shop.

Solution: JAsCo stateful aspects support triggering crosscutting behavior on the opposite of a protocol using the *complement* keyword. In this example such a feature is useful since the manager needs to be notified whenever the protocol is not respected. The solution looks as follows:

```
class ExecutionOrderAspect {
  hook ExecutionOrderConstraintHook {
    ExecutionOrderConstraintHook(
      paymentOrder(Order order),
      shippingOrder(Order order),
      methodsContext(..args)) {

        complement[execute(methodsContext)]:
        start > payment;
        payment: execute(paymentOrder) > shipping;
        shipping: execute(shippingOrder);
    }
    replace complement() {
       Manager.notify
         ("Shipping done before payment");
    }
  }
}
```

Code fragment 3 – Stateful aspect for payment execution order

```
static connector ExecutionOrderConnector {
  ExecutionOrderAspect.ExecutionOrderHook checker
  = new ExecutionOrderAspect.ExecutionOrderHook (
   void paymentService.pay(Order order),
   void shippingService.ship(Order order), {
      void paymentService.pay(Order order),
      void shippingService.ship(Order order)
   }
 );
}
```

Code fragment 4 – Connector for payment execution order

The `ExecutionOrderAspect` in code fragment 3 states that if the specified protocol is not followed (e.g. the shipping is done before the payment) the original behavior of the core application is replaced by a notification which is sent to the manager. By using the complement keyword, the specified advice is only executed when the protocol is *not* followed.

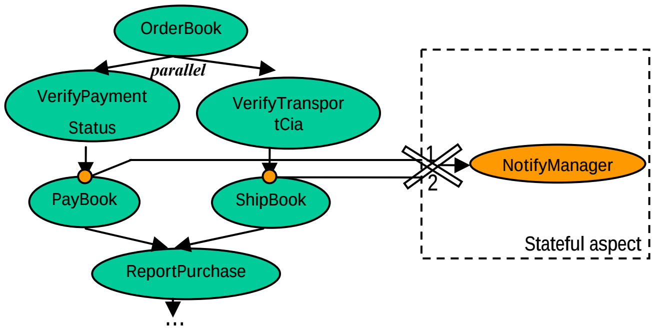


Figure 3 – Stateful aspect for triggering notification when a certain path is not respected in e-commerce workflow

5. Related work

Previous research focused on the applicability of AOP for business rules. Experiments have been done in AspectJ and JAsCo illustrating the connection of business rules with the core object-oriented application [5][6]. Other related previous research focused more deeply on the implementation level, more specific on the integration between the object-oriented paradigm and rule-based languages as a rule-based approach is a more suitable paradigm to implement business rules in [9]. Hybrid aspects were proposed to achieve an oblivious integration at the language and programming level.

However, in some situations, applications cannot afford to use a rule-based programming language for implementing their business rules since it might be too costly to incorporate support for a rule engine for the kind and amount of rules under consideration. Rule engines are proprietary, expensive products and the learning curve can be steep and unaffordable for small projects. Thus, a more lightweight approach is needed for the definition and implementation of the business rules. In this context, ongoing work is being carried out that envisions the definition of a high level business rule language for the specification of the rules, very close to natural language. This language allows the definition of business rules independently of any implementation detail. This way, business rules are defined using the concepts defined in a business model. A business model contains the different elements of the domain under consideration. We are investigating the definition of a very configurable and extensible business model in order to be extended with different terms present in different domains such as, in this case, the web service composition context. An automatic translation of the rules defined in this high level language to a possible object-oriented representation is under development as well as the transparent and automatic generation of the connectivity JAsCo aspects for their integration with the core application.

In [3], an AOP extension for BPEL is proposed. Examples of business rules written in AO4BPEL are given in [4], illustrating the use of AOP for decoupling business rules. This approach is dynamic in the sense that it is possible to plug-in and out the aspects at run-time. As it is an interpretation based approach, aspects can be plugged in at run-time and triggered when the interpreter reaches their pointcut definition. Pointcuts in AO4BPEL are based on Aspect's pointcut model. As most current mainstream AOP languages, AspectJ's pointcuts (with the exception of *cflow()*) cannot refer to the history of previously matched pointcuts in their specification. Thus, AO4BPEL does not allow the triggering of aspects depending on protocol history conditions.

6. CONCLUSION

In this paper we have presented different example categories of business rules that are applicable in the web service composition context. Business rules that are based on the dynamics of the core composition are addressed meaning that either the triggering of the rules, their conditions or actions are based on protocol history. As a consequence of their application, crosscutting behavior is plugged in which results in the addition, change or removal of activities in the base composition. We provide example categories of dynamic business rules in web services compositions. To illustrate our results, examples of a possible implementation in the JAsCo AOP language are given. JAsCo supports stateful aspects which allow the definition of stateful pointcut expressions. The use of JAsCo stateful aspects allows a more seamless integration of the identified dynamic business rules in web service compositions.

The business rules presented in this work are examples of rules that guide *how* the composition should adapt to the changing business environment. In order to fully cover the whole domain of service composition business rules this work can be continued by addressing the other kinds of composition rules already identified in these paper.

To enhance the specification and implementation of business rules, our current line of research focuses on creating a high-level business rules language. This language allows specifying rules in a very declarative way, without having to be aware of specific AOP constructs. AOP is used as an underlying layer to realize the connection of the business rules with the core applications in an oblivious way.

7. REFERENCES

[1] Arsanjani, A.. Rule object 2001: A pattern language for adaptive and scalable business rule construction.

[2] Business Process Execution Language for Web Services (WS-BPEL), Specification Version 1.1, www-128.ibm.com/developerworks/library/ws-bpel/

[3] Charfi, A., Mezini, M., Aspect-Oriented Web Service Composition with AO4BPEL, LNCS 3250 , 2004

[4] Charfi, A., Mezini, M., Hybrid Web Service Composition: Business Processes Meet Business Rules, 2nd International Conference on Service Oriented Computing, New York City, USA, November 2004

[5] Cibrán M. A., D'Hondt M., Jonckers V.: Aspect-Oriented Programming for Connecting Business Rules. In Proceedings BIS, Colorado Springs, USA (2003)

[6] Cibrán M. A., D'Hondt M., Suvée D., Vanderperren W., Jonckers V.: JAsCo for Linking Business Rules to Object-Oriented Software. In Proceedings CSITeA, Rio de Janeiro, Brazil (2003)

[7] Cibrán, M. A., Verheecke, B. and Jonckers, V. Modularizing Client-Side Web Service Management Aspects. In Proceedings of the second Nordic Conference on Web Services. Vaxjo, Sweden, November 2003.

[8] Date C.: What not How: The Business Rules Approach to Application Development. Addison-Wesley Publishing Company (2000)

[9] D'Hondt M., Jonckers V.: Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge. In Proceedings of AOSD, Lancaster, UK (2004)

[10] M. D’hondt: Hybrid Aspects for integrating Rule-based Knowledge and Object-Oriented Functionality, Phd Thesis, Vrije Universiteit Brussel, May 2004.

[11] Kappel, G., Rausch-Schott, S., Retschitzegger, W. and Sakkinen, M., From rules to rule patterns. In Conference on Advanced Information Systems Engineering, pages 99--115, 1996.

[12] Ross R. G.: Principles of the Business Rule Approach. Addison-Wesley (2003)

[13] Suvée, D. and Vanderperren, W. “JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development,” in Proc of Second International Conference on Aspect-Oriented Software Development, Boston, USA, March 2003.

[14] The Business Rules Group. Defining Business Rules: What Are They Really?, July 2000. http://www.businessrulesgroup.org/

[15] Vanderperren, W., Suvee, D., Cibrán, M. A. and De Fraine, B. Stateful Aspects in JAsCo. To be published at the Software Composition Workshop (LNCS), ETAPS 2005, Edinburgh, Scotland, April 2005.

[16] Verheecke, B., Cibrán, M. A. and Jonckers, V. AOP for Dynamic Configuration and Management of Web services in Client-Applications. In Proceedings of 2003 International Conference on Web Services. Erfurt, Germany, September 2003.

[17] Von Halle B.: Business Rules Applied. Wiley (2001)

Flexible Call-by-call Settlement
An Opportunity for Dynamic AOP

Christian Hofmann
TU Ilmenau
Ehrenbergstraße 29
98693 Ilmenau, Germany
hofmann_ch@gmx.de

Robert Hirschfeld
DoCoMo Euro-Labs
Landsberger Straße 312
80687 Munich, Germany
hirschfeld@acm.org

Jeff Eastman
Windward Solutions
1081 Valley View Ct.
Los Altos, CA 94024, USA
jeff@windwardsolutions.com

ABSTRACT
Dynamic aspect-oriented programming is gaining interest due to its ability to provide attractive solutions to challenging technical problems. Most scenarios presented to date are motivated by the technical capabilities of a particular platform rather than application-level requirements to be addressed. In this paper, we describe a scenario taken from telecommunications where settlement systems of operators and call-by-call providers need to be integrated in a flexible manner after the system's initial deployment. By comparing static and dynamic object-oriented and aspect-oriented approaches, we present a case for dynamic AOP.

1. CALL-BY-CALL SCENARIO
In the telephony domain, rate plans vary from provider to provider. There are many different tariffs, for for example: local, long distance, international, or toll-free calls. There are also variations in how and when customers are billed. Besides other ways to compete in this market, there are call-by-call providers offering dial-around services. Here customers can choose their actual connectivity provider for each individual call by dialing a specific prefix or toll-free number prior to the actual destination phone number to be called (Figure 1).

The great flexibility offered to customers to select the best (and in most cases cheapest) connectivity provider demands a corresponding flexibility in the core telephony network and its supporting systems. Network operators and call-by-call providers can choose to start or terminate business relationships at any time. Such decisions are rarely synchronized with the design, development, or deployment of the responsible software systems. Most of these partnerships are formed and terminated while their systems are being operated long after their initial deployments. Entering into a partnership requires both operators and providers to integrate their operations environments. Such integration requires agreement on data and data formats to be exchanged as well as the way this data is to be transmitted.

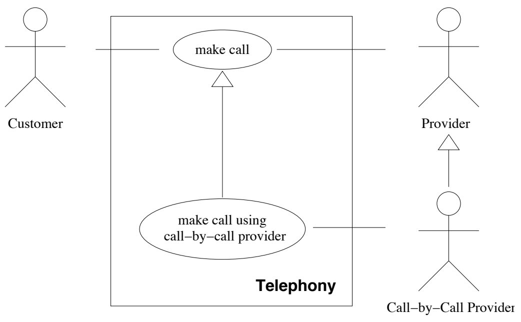


Figure 1: Dial-Around Services

Settlement is one such area to be addressed (Figure 3). Both operators and providers run their own settlement pipelines that are unlikely to be exactly compatible in the formats of call data records to be processed. Nor are the interfaces required to interchange raw or processed call data records standardized. Finally, security policies to be enforced and communication protocols and endpoints to be utilized have to be agreed upon. For customer convenience, some operators offer combined invoices to simplify customer billing. Items charged for by a call-by-call provider need to be listed separately from the ones charged for by the operator to make cost distribution explicit and transparent to the customer. The way items are listed may also vary from partnership to partnership.

As it might have become apparent from the problem description so far, integrating settlement systems is a rather complex task. Complexity increases by the general requirement to keep system down time minimal, on both the operator and the provider side. An operator can partner with several call-by-call providers over a period of time, as similarly a call-by-call provider might partner with several operators to render its services to its customers.

In our paper we discuss selected system functionality as indicated by the use-cases marked gray in Figure 3. We identify variation points needed to support the aforementioned establishment and termination of partnerships, and describe

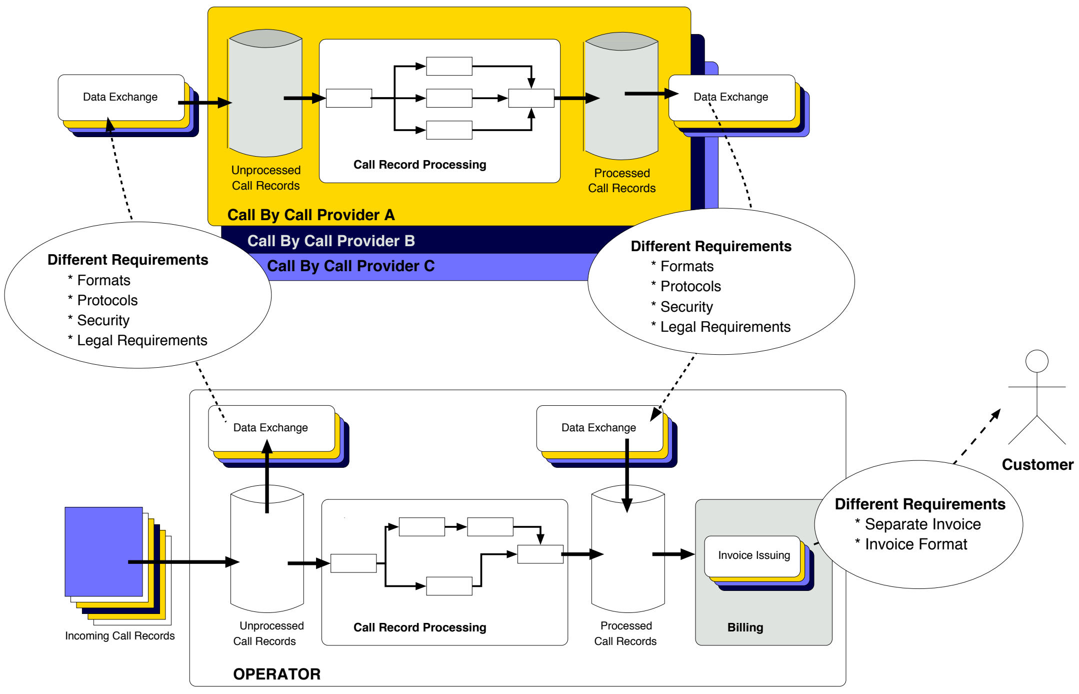


Figure 2: Variation Points

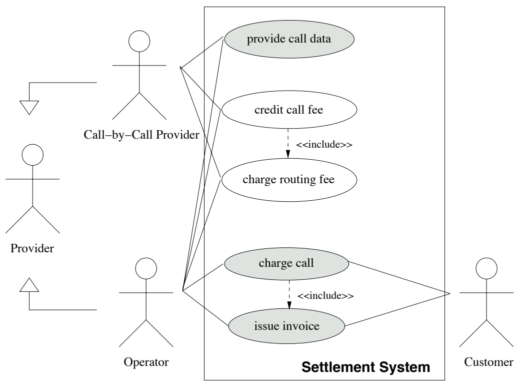


Figure 3: Call-by-call Settlement

how these variation points can be realized by applying both object-oriented and aspect-oriented approaches. We show how dynamic composition is of benefit in both cases; recognizing the dynamic aspect-oriented approach as the most beneficial one.

2. SELECTED VARIATION POINTS

As indicated in the previous section, integrating settlement systems offer interesting challenges. From the point of view of each participant, its settlement system is considered the stable part in such an activity. The settlement of calls can be described as the processing of call data records. It is typically performed as a sequence of processing steps that can be conveniently arranged within a settlement pipeline. Each step is responsible for one particular task. Example tasks are data record collection, correlation of ingress and egress records, fraud detection, duplicate removal, and billing. In general, a settlement system converts raw data records into processed records that in turn are used for billing all participating parties (Figure 2).

While operators process their call data records themselves, records of call-by-call providers are processed preferably at a provider's site. After sorting and collecting records for every individual provider (for example according to the dialing prefix used to initiate the call), all such records are transmitted to each corresponding provider. There they are handled similarly to the processing performed at the operator's site. At the end of a provider's settlement pipeline, calls are settled either by the provider directly or, for customer convenience, by the operators customers originated their calls. In the former case, the provider uses its own billing system. In the latter case, all processed call data records need to be transferred back to the operator where they are phased back into the operator's settlement system. There this data is eventually used to prepare the customer's bill that enlists provider charges separately and differently.

From this description we can infer at least three variation points to be present to allow for operator-provider settlement system integration:

- Raw call record transfer from the operator to the provider,
- Processed call record transfer from the provider back to the operator, and
- Provider-specific billing by an operator.

Both the first and the second item most likely require data conversion from one system's representation to the other's. Also, it has to be decided which security policies (including encryption mechanisms) to enforce and which transmission protocol suites to use (Figure 4).

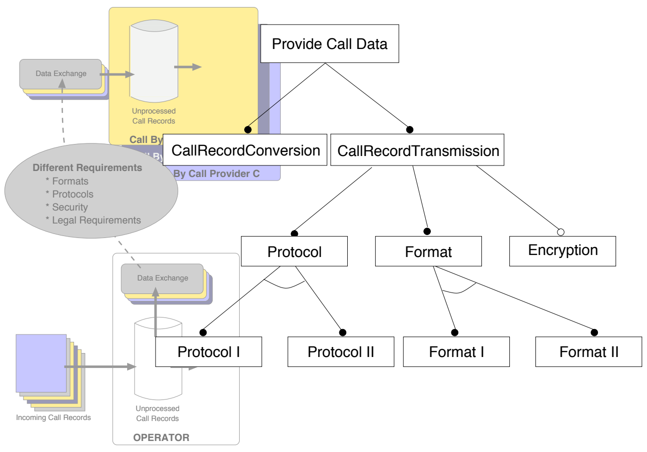


Figure 4: Call Record Exchange

The third item, provider specific billing needs to reflect legal constraints and requirements (such as privacy concerns) or company guidelines (such as style guides for bill rendering). Also, customers may often still select to receive their invoices by mail instead of viewing them online via the Web (Figure 5).

To integrate two such settlement systems, at least one if not both of them need to be adapted, at all of the variation points mentioned above. In addition to that, system downtime must be kept at a minimum. This requirement makes runtime adaptation very attractive for system integration. Partnership changes need not then equate to an off-line system modification. The following sections discuss four approaches to address system integration and show how system downtime can be avoided.

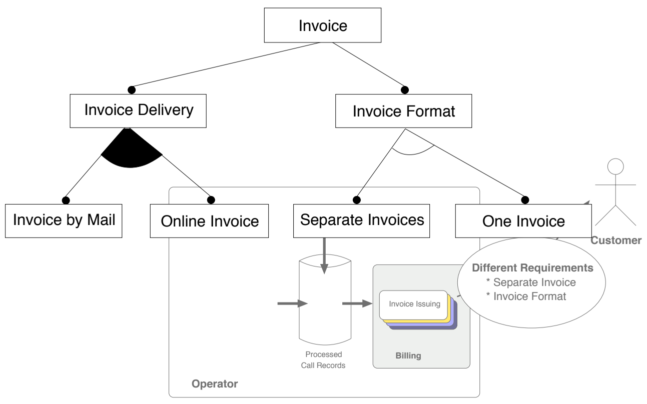


Figure 5: Invoice

3. OBJECT-ORIENTED VARIATION POINTS

First we show how some of the variation points outlined in the previous section can be implemented with object-oriented technology, relying on polymorphism or dynamic registries with explicit dispatch.

3.1 Static Object-Oriented

Figure 6 models call records, different kinds of calls, and the specifics needed by individual providers via single inheritance. Here we can see that the kind of call or call-type (in our example only long-distance and international are addressed) acts as the dominant decomposition criterion. Both implementations can offer different ways of charging, or - from the operator's point of view - different ways of creating provider bills. To further distinguish long-distance and international call records by provider, we need to subclass each of them and enhance every such subclass with provider-specific behavior that might be duplicated across neighboring branches of the class tree. Such behavior can comprise call-record conversions or the selection of transmission protocols.

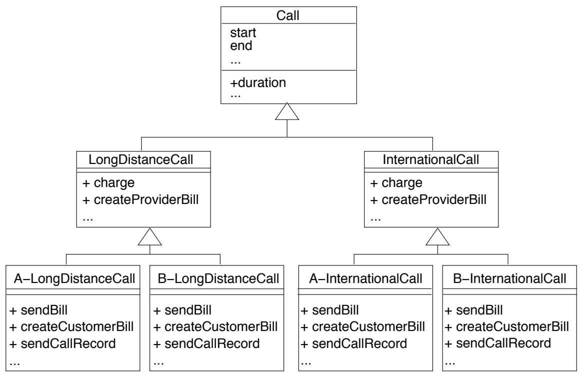


Figure 6: Dominant Decomposition by Call-type

If we decide to use provider-specifics as the dominant decomposition criteria, we might end up with a class tree as shown in Figure 7. Information specific to long-distance or international calls needs to be modeled by further sub-classing our provider-specific classes. This modeling step will introduce duplication across neighboring branches of the class tree similar to the dominant decomposition by call-type described above.

In languages providing multiple inheritance or mix-in behaviors, we might end up with a class model as depicted in Figure 8. While with mix-ins there can still be a dominant decomposition (here kind of call, or call-type), mix-ins are a means to help us avoiding code duplications by combining crosscutting concerns in our implementation model.

Note that all models described above aim not only for implementation reuse, but also for simplicity of method dispatch by using polymorphism. Such dispatch is necessary to select the appropriate implementation that matches, at each variation point, the correct provider and call type. Dispatch is also needed when transmitting raw call records from an

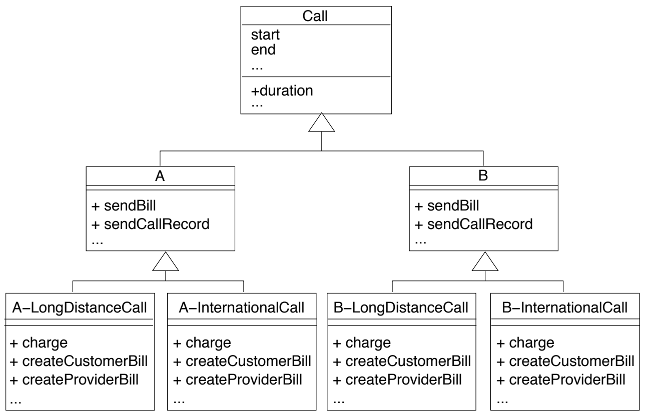


Figure 7: Dominant Decomposition by Provider-specifics

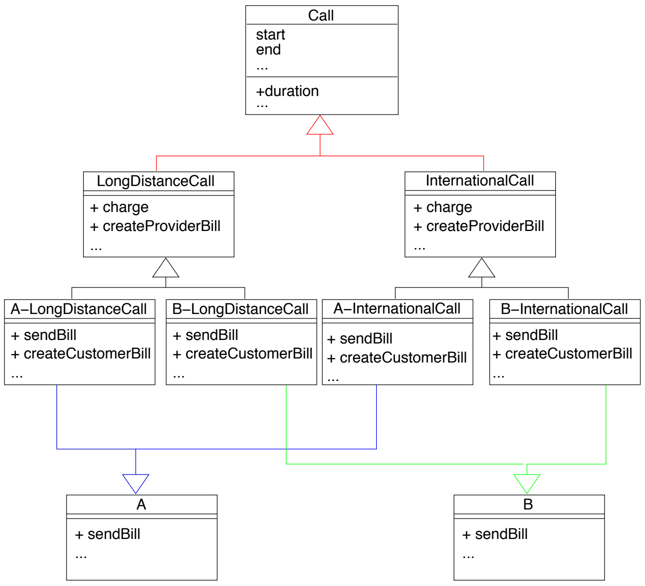


Figure 8: Mix-in Solution

operator to one of its partnering call-by-call providers, and when transmitting processed call records from a provider back to to one of its partnering operators. Finally, similar selection criteria also apply for provider-specific customer billing at the operator's site.

When using a static object-oriented approach, we need to know all possible partners in advance at development time in order to provide a complete set of implementations covering all possible dispatches that might be necessary during system operation. Since, in a static object-oriented approach, code cannot change at runtime, all necessary dispatch code needs to be provided initially. Code that turns out to be incomplete then requires the exchange of deployed system components. This requires expensive hardware and software fail-over solutions to avoid system outages and down-time in the high availability (99.999%) telecom world.

3.2 Dynamic Object-Oriented

A dynamic object-oriented solution benefits from language platforms that allow to load and integrate additional code into or remove code from a running system. Examples of such platforms range from Java with its dynamic class loaders to Smalltalk or Lisp. In the latter two there is no distinction between code and data, thus any data made available at runtime can also be interpreted as computation. This allows partnership-specific processing data to be incorporated into the system at runtime, or removed to storage for use at another time. Since such behavior is difficult to be visualized in rather abstract models, we will provide code to describe one such solution. Due to its flexibility, usability and open source availability, we opted for Squeak/Smalltalk [4, 2] to do so.

Smalltalk has a powerful mechanism for expressing small units of computation as blocks and block contexts. A block is an object that embodies a sequence of operations. It is only executed after a value message is received by the block. A block context holds the dynamic state, such as parameter values, for execution of a block. Since blocks are, as anything else in Smalltalk, regular objects, they can be manipulated and stored as any other object.

A simple dynamic and customizable dispatch mechanism can be implemented by using a dictionary (similar to a hash table in Java) as a dispatch table. The dictionary allows us to store a set of associations where we use an association's key part to store the information necessary for dispatch selection, and the association's value part to store the code to be activated in the context of a dispatch. The code is provided as a block. On dispatch we would simply use the dispatch criteria to look up an associated block. Once obtained, this block is executed by simply sending a value message as mentioned above.

In our scenario, we can create a dispatch dictionary for each variation point with call prefixes or identifiers as keys and the appropriate provider-specific sequence of operations as values.

```
Object subclass: #RawCallRecordExchange
  instanceVariables: 'dispatchTable
                      defaultAction'
```

```
RawCallRecordExchange>>
initialize
  dispatchTable := Dictionary new.
  defaultAction := [:anObject | self error].

RawCallRecordExchange>>
addSelector: anArray
  action: aBlockContext
   dispatchTable add: anArray -> aBlockContext.

RawCallRecordExchange>>
removeSelector: anArray
  dispatchTable removeKey: anArray
                ifAbsent: [].

RawCallRecordExchange>>
dispatchOn: anArray with: aCallRecord
   (dispatchTable at: anArray
    ifAbsent: [defaultAction])
   value: aCallRecord.

RawCallRecordExchange>>
sendCallRecord: aCallRecord
   | prefix |
   prefix := aCallRecord callPrefix.
   self dispatchOn: { prefix. #before.
                      #sendCallRecord:. }
         with: aCallRecord.
   ...
   self dispatchOn: { prefix. #after.
                       #sendCallRecord:. }
        with: aCallRecord.

rawExchange := RawCallRecordExchange new.
rawExchange
 addSelector:
  { 01071 . #before. #sendCallRecord:. }
 action:
  [:aCallRecord |
   aCallRecord server
   authenticateUsingKerberos.
   aCallRecord authenticated
    ifTrue: [aCallRecord server connect]].
rawExchange
 addSelector:
  { 01071. #after. #sendCallRecord:. }
 action:
  [:aCallRecord |
   aCallRecord server disconnect]].
...
rawExchange sendCallRecord: aCallRecord.
```

Every time a new partner (an operator or a provider) needs to be added to the system, all dictionaries at our variation points are populated with the call prefixes or identifiers of the new partner and the code block specific to the new partner and the concerned variation point. If a dispatch for the new partner needs to be carried out, we simply look up and evaluate the code block associated with that partner. Terminating a partnership only involves removing all dictionary entries associated with the partner separated from. Starting or terminating relationships does not require a system rebuilt and exchange as in the static object-oriented case. With this dynamic registry and dispatch mechanism we can modify the running system without requiring elaborate fail-over mechanisms required by static methods.

4. ASPECT-ORIENTED VARIATION POINTS

As with the static and dynamic object-oriented implementations of our variation points, there are static and dynamic aspect-oriented implementations as well. In the following we see how aspect-orientation helps us to avoid the coding of explicit selections or explicit dispatches by using the dispatch mechanisms already built-in into an aspect-oriented composition platform.

4.1 Static Aspect-oriented

Figure 9 shows an aspect-oriented model of call records, kinds of calls, and the specifics needed by individual providers. In contrast to our object-oriented models, for instance the one in Figure 6, we decided to model only kinds of calls within a call records hierarchy. This hierarchy acts as the base system of our aspect-oriented model in which we express call record exchange and issuing invoices to customers.

Besides the reduction of code duplication, our aspect-oriented model also frees us from explicitly selecting a particular implementation for a specific operator or provider since such conditionals are hidden within the static compositions or residual test of the aspect composition.

Within a static aspect-oriented model the solution space must be known completely at compile-time, requiring us to know all possible combinations and their implications on our integrated system when it is deployed. If, after the deployment of our integrated system, we discover that we were wrong, we would have to perform similar corrective actions as the ones needed to update a statically modeled object-oriented system.

4.2 Dynamic Aspect-oriented

As with the dynamic object-oriented solution, the dynamic aspect-oriented solution benefits from the malleability of dynamic programming and composition platforms at runtime. Examples of such platforms are Steamloom, Prose, or AspectS. Since AspectS [3] is our research platform, we provide our sample code for this system.

We do not describe a different aspect model for the dynamic case since the static one discussed previously will do just fine. To implement our variation point for exchanging raw call-records, we first extend AspectS so that we can easily express aspects and their associated advice constructs that are active or inactive depending on a particular provider. This requires nothing else than allowing aspects to be provider-specific. We implement a provider-specific activation block and make it accessible to an advice qualifier object via the #providerSpecific attribute.

```
AsMethodWrapper class>>
providerSpecificActivator
  ^[:aspect :baseSender |
    | result receiver |
    receiver := baseSender receiver.
    result := aspect hasProvider:
    receiver callerNumber prefix.
    aspect := baseSender := nil.
    result] copy fixTemps
```

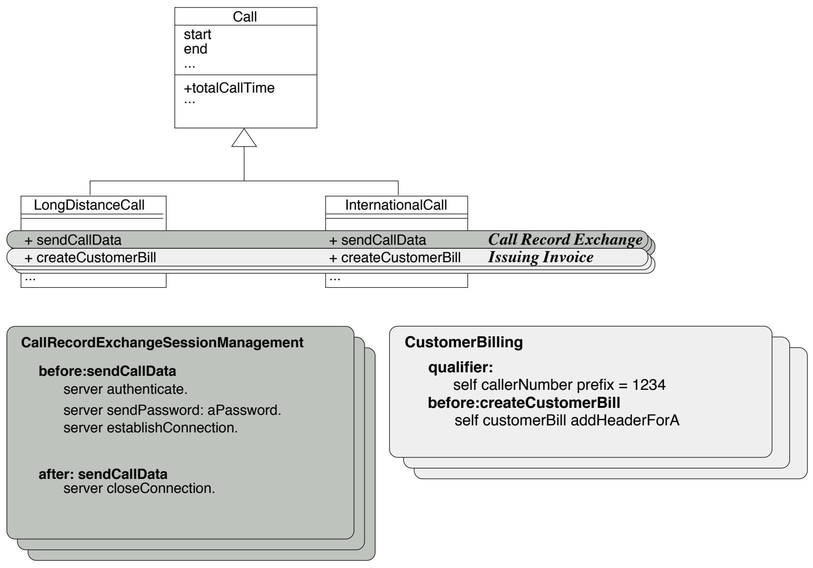


Figure 9: Aspect-Oriented Composition

More code is necessary to make the provider-specific advice qualifier attribute work. It requires the following changes to class AsAspect:

```
Object subclass: #AsAspect
  instanceVariableNames:
   'receivers senders
    senderClasses projects providers
    clientAnnotations advice installed'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'AspectS-Aspects'
```

AsAspect>>
providers
```
  ^providers
```

AsAspect>>
providers: anIdentitySet
```
  providers := anIdentitySet.
```

AsAspect>>
initialize
```
  self receivers: IdentitySet new;
  senders: IdentitySet new;
  senderClasses: IdentitySet new;
  projects: IdentitySet new;
  providers: IdentitySet new;
  clientAnnotations: IdentityDictionary new;
  advice: nil;
  installed: false
```

AsAspect>>
addProvider: aProvider
```
  ^self providers add: aProvider
```

AsAspect>>
removeProvider: aProvider
```
  ^self providers remove: aProcess
                  ifAbsent: []
```

AsAspect>>
hasProvider: aProvider
```
  ^self providers includes: aProvider
```

Now we can implement our raw call record exchange, making use of our newly defined advice qualifier attribute. Here it is interesting to note that the association key we used in the dynamic object-oriented example to select the appropriate dispatch is represented as a provider specific activator in our dynamic aspect-oriented example. The values of the previously used associations (code blocks) can now be found in our AspectS code as the before and after blocks of the respective advice constructs.

```
AsAspect subclass: #RawCallRecordExchangeAspect
```

SessionManagementAspect class>>
prefix: anInteger
```
   ^self new
    addProviderPrefix: anInteger
```

RawCallRecordExchangeAspect>>
adviceSendCallDataProvider
```
   ^AsBeforeAfterAdvice
     qualifier: (AsAdviceQualifier
       attributes: { #receiverClassSpecific.
                    #providerSpecific.})
     pointcut: [{ AsJoinPointDescriptor
       targetClass: InternationalCall
       targetSelector: #sendCallData.
      AsJoinPointDescriptor
       targetClass: LongDistanceCall
       targetSelector: #sendCallData. }]
     beforeBlock: [receiver server
         authenticateUsingKerberos.
       receiver server authenticated
         ifTrue: [receiver server connect]]
     afterBlock: [receiver server disconnect]
```

```
aspect := RawCallRecordExchangeAspect
   prefix: 12345.
aspect install.
```

Looking at the code above we can see that the dynamic aspect-oriented solution does not require an explicit dispatch to be provided by a developer since this dispatch is intrinsic to all aspect-oriented platforms. Whenever necessary, the underlying aspect system accesses the activation block provided by us, evaluates this block, and, depending on the outcome of this evaluation, activates the associated advice code or not. So, in addition to all the flexibility gained by our dynamic object-oriented solution, we also achieve simplification of our code by the utilization of a hidden but well known and proven system-provided dispatch mechanism.

5. SUMMARY

In this paper we provide a scenario taken from telecommunications to motivate the need for dynamic aspect-oriented programming languages and systems. Our scenario describes how constantly changing relationships between operators and call-by-call providers affect their system integration requirements (here in the context of settlement), and how one of the most important of these requirements – keeping system downtime to a minimum – can be supported by employing dynamic composition in general, and dynamic aspect-oriented composition in specific. While focusing on dynamic aspect composition, we do not argue for or against the merits of aspect-orientation in general; this is done adequately elsewhere [1].

	OO	AO
static	implicit dispatch fixed set of providers explicit selection	implicit dispatch fixed set of providers
dynamic	explicit dispatch	implicit dispatch

Table 1: Properties of the Proposed Solutions

Table 1 summarizes the properties of the proposed solutions ranging from static and dynamic object oriented techniques to dynamic aspect-oriented composition. It is no surprise that all decisions made in advance of building a software system can be modeled, implemented, and optimized via early-bound object- or aspect-oriented systems. Later decisions, more precisely decisions made after the construction and deployment of a software system can be modeled, implemented, and, most importantly to us, adapted only in late-bound object- and aspect-oriented systems. In addition to adaptability, an aspect-oriented system has many other advantages enabling the support of unanticipated software evolution[5]. In our example, use of the built-in dynamic dispatch mechanism allows us to avoid explicit and critical dispatching code.

6. ACKNOWLEDGEMENTS

We would like to thank Matthias Wagner and Monika Fuchs for their valuable discussions and contributions.

7. REFERENCES

[1] http://www.aosd.net.

[2] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[3] R. Hirschfeld. AspectS - aspect-oriented programming with squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays 2002*, LNCS 2591, pages 216–232, Erfurt, 2003. Springer.

[4] D. Ingalls, T. Kaehler, J. Maloney, W. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices*, 32(10):318–326, Oct. 1997.

[5] G. Kniesel, J. Noppen, T. Mens, and J. Buckley. Unanticipated software evolution. *Lecture Notes in Computer Science*, 2548:92–107, 2002.

Dynamic Aspects for Runtime Fault Determination and Recovery

Jeremy Manson, Jan Vitek, Suresh Jagannathan
Department of Computer Science
Purdue University
{jmanson,jv,suresh}@cs.purdue.edu

ABSTRACT

One of the most promising applications of Aspect Oriented Programming (AOP) is the area of fault tolerance and recovery. In traditional programming languages, error handling code must be closely interwoven with program logic. AOP allows the programmer to take a more modular approach - error handling code can be woven into the code by expressing it as an aspect.

One major impediment to handling error code in this way is that while errors are a dynamic, runtime property, most research on AOP has focused on static properties. In this paper, we propose a method for handling a variety of run-time faults as dynamic aspects. First, we separate fault handling into two different notions: *fault determination*, or the discovery of faults within a program, and and *fault recovery*, or the logic used to recover from a fault.

Our position is that fault determination can be expressed effectively as dynamic aspects. We propose a system, called Rescue, that exposes underlying features of the virtual machine in order to express faults as variety of run-time constraints. We show how our methodology can be used to address several of the flaws in state of the art fault fault handling techniques. This includes their limitations in handling parallel and distributed faults, their obfuscated nature and their overly simplistic notion of what a "fault" actually may comprise.

1. INTRODUCTION

It is extremely difficult to write code that handles faults efficiently and effectively. They are a reasonably high percentage of source code generated: one study indicates that up to 5% of program text is contained in fault handling code, and that anywhere up to 46% of a given program is reachable from that code [9].

We take an even broader view of faults: instead of saying a fault is a situation where something drastic has occurred, we simply define it as a situation that arises because of unexpected change in resource availability or performance assumptions. This notion encompasses the traditional notion of program/node failures, as well as quality of service and other non-functional characteristics of distributed program behavior. For example, under this model, excessively high load on a processor might be characterized as a fault.

The current state of the art mechanism for dealing with faults in software treats them as *exceptions* which can be *thrown* by a line of code, and then *caught* by an exception handler whose dynamic scope encloses the code that threw the exception. Some form of exception handling is available in most modern languages, including Java, C++, C#, Visual Basic, Ada, ML and Haskell. Their use in this context stretches back to the CLU programming language [4]. However, exceptions have a number of limitations.

The first problem that we encounter with the use of language-level exceptions is that they tend to foster a binary view of faults: either the program is out of memory (for example), or it is not. This does not allow for graceful resolution of resource exhaustion – if we want to perform some emergency cleanup task when a certain percentage of memory is available, we are out of luck.

Exceptions also obfuscate code greatly. One or two thrown exceptions are simple to catch, and the resulting code is simple to read. However, in large systems, with calls to many different libraries, code can throw many different exceptions. This leads to a software engineering challenge: there will be many clauses in the program that catch these exceptions, all of which are interlaced with program logic that has nothing to do with fault handling and tolerance.

This problem is exacerbated when dealing with faults; faults incorporate unexpected as well as erroneous conditions. Since fault criteria can be arbitrarily complex, using standard exception handling mechanisms to express more general faults leads to further complexity.

It is also unclear how exceptions raised on one segment of a distributed application can be handled cleanly in another. If, for example, code running on one node fails, an exception is propagated up the stack for that node, but not necessarily propagated to other nodes which require the information.

Our conclusion is that is desirable to have a way *to separate* *the definition of an application-specific fault from the mechanics of fault book-keeping.* Once a fault is identified, the infrastructure should provide *facilities for orderly recovery and compensation.*

This paper discusses a novel programming language based approach to determining the presence of faults, called RESCUE. The language is designed to be implemented on both stand-alone and distributed systems; it is an extension to AspectJ [3], the most widely used AOP language for Java. RESCUE would provide a wide spectrum of ways for users to define mechanisms for determining when faults occur in their programs. Once they have declared how the faults occur, they can either define their own compensatory code, or use one of a number of built-in mechanisms (such as transactions, or distributed replication). The code they define may change behavior across the entire infrastructure; thus, the resulting system would provide a pervasive system for defining and handling faults in a clear, clean and efficient way.

2. FAULT DETERMINATION IN RESCUE

In distributed systems, just as in a parallel program, there is a large set of events that need to be monitored in order to detect "exceptional" situations. RESCUE provides infrastructural support for *fault determination*, the act of recognizing such conditions. An informal taxonomy of the faults RESCUE addresses is given below:

- *Internal v. External.* An external event (for example, memory exhaustion or a network outage) exists outside of the control of the program and can be monitored either within the virtual machine or in middleware. An internal event corresponds to a fault in the application: for example, when a thread raises an exception as a result of a condition that compromises the integrity of other concurrently executing tasks.
- *Fine- v. Coarse-Grained.* Some events, such as a measurement of CPU usage or a hardware interrupt, have high frequencies. Other events, such as node failure or file operations, are less frequent. The granularity impacts the performance of monitoring.
- *Builtin v. User-defined.* Builtin events include events supported by the VM, like CPU or memory usage. User-defined events may include something as simple as a counter incremented every time an element is added to an array; a fault might occur if the count gets too high.
- *Local v. Distributed.* An example of a local event would include a change in CPU usage on a given node. An example of a distributed event would be a failure on a different node.

In order to support the full taxonomy listed above, RESCUE requires functionality that goes well beyond the current state-of-the-art. Faults can be complex to express, and the logic to keep track of them may not be easily localized. In the case of fine-grained event monitoring, such as timers or hardware interrupts, it is essential to guarantee that the overhead of fault determination can be bounded.

2.1 Meters and Plans

RESCUE includes a declarative language for fault determination; it is integrated into Java and designed to allow for an efficient implementation. The two key concepts in the language are *plans* and *meters*. Plans are specifications of fault conditions as a predicate over meters. Meters provide an interface to monitoring events. Typically, plans are written by an application developer, as they describe the condition which requires intervention. Meters, on the other hand, are designed to be provided by the infrastructure: i.e. either the virtual machine or libraries. User-defined meters can also be defined using a programmatic API.

2.1.1 Meters

Meters provide the flexibility of join points for faults. A meter is attached to a resource, and can be used to tell the user how much of that resource remains. For example, a meter may give information as to how much memory remains, how many threads are active in a system, or how much network bandwidth is available.

```
plan NetworkUnavailable:
  meter bandwidth < 128kbps or
  meter latency < 100ms;
```

Built in memory meters can also be used to determine that an application is running out of memory. For example, the following plan can be used to trigger a memory fault:

```
plan MemoryFault:
  meter memory < 100k;
```

The obvious alternative to this approach would be to have application code poll for memory at regular intervals. Doing this in an automated way is clearly not very efficient in the case where many applications are running on the same host, trying to monitor the same fine-grained fault. Meters, on the other hand, have compiler support and are registered with the environment. So, for instance, in the case of memory, the virtual machine could insert checks in the memory allocator (and try to optimize these checks).

Additionally, meters may be defined in library code. A meter in an I/O class may, for example, tell the user how much space is left in a buffer, so that it can avoid buffer overflow/underflow. In this case, the programmer explicitly controls the frequency of checks. Obviously, this would also be a case where more traditional aspect-oriented approaches might suffice.

2.1.2 Plans

Once a meter is in place, it can be used to indicate the presence of a fault by placing it in a *plan*, possibly aggregated with other meters. In the following example, the plan is triggered when network bandwidth is low or the latency (perhaps of the last remote request) is high.

```
after(): within(Task) && NetworkUnavailable {
  spawnTasksOnLocalHost = true;
}
```

The above code deals with the spawning of tasks on a distributed platform. It specifies that when the control flow is in the Task class and the network becomes unavailable, the task should be spawned locally. The example above is not quite complete, as we should specify the *rate* at which plans are evaluated. Without a rate, the `MemoryFault` plan would be triggered at every allocation. The user can allow the compiler to determine the rate (as mentioned earlier), or can control it with explicit annotation. This can be done with a `period` argument, e.g. `period == 5s`, or with the `once` predicate which ensure that plan can only evaluate to true once.

The expressive power of the meter predicates is determined by the meters themselves. So, for instance, the memory meter only allows for constant arguments and simple inequalities. This is because the implementation of the meter needs to order all meter predicates and be guaranteed that the value of the meter expression does not change.

User-defined meters can include a `evaluate(exp)` predicate to evaluate user code. For instance, it is possible to associate a meter with the filesystem so that every I/O operation evaluates the user defined `check()` function. To reduce the granularity of events, inclusion of the `period` predicate ensures that `check()` will be invoked at most once per second.

```
plan Tick:
  meter file evaluate(check()) &&
        period == 1s;
```

We can directly associate the execution of method with a plan by writing the following (`when` is a new keyword introduced by RESCUE as a shorthand for a more thorough aspect-like syntax):

```
void takeAction() when Tick {
  ... implementation ...
}
```

2.2 Discussion

Meters monitor thread and system state. When a plan is triggered, it may in turn affect the threads affected by the fault, and update system resources as part of a recovery process.

Not all plans need to be built-in runtime environment constructs like network bandwidth or CPU usage. The user may also define her or his own plan. As an example, consider the fact that in distributed and cluster environments, there may be intermittent failures in one or more nodes. It is frequently useful to provide a "timeout" function: a component will register the fact that it is alive, and the timeout will expire if they have failed, or are unreachable. In the Java-based network services specification Jini [6], this timeout is called a *lease*. Unlike Jini, leases in RESCUE are simply user-defined instantiations of plans and meters, and not primitive.

Using RESCUE, a library that holds a resource may implement leases by simply placing a meter on the time elapsed

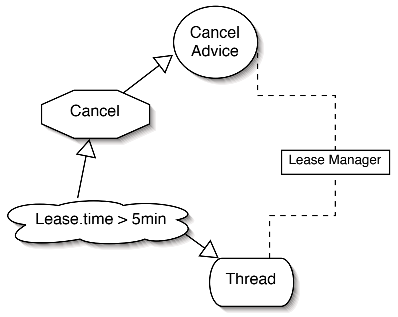


Figure 1: Using plans and advice to express a lease.

since it was contacted by a given lease holder. An example of this can be seen in Figure 1. A meter monitors the acquisition of leases by a thread. Whenever an acquired lease exceeds a bound (in this case 5 minutes) a cancel plan is triggered, and the executed code notifies the lease manager to cancel the lease. This simple example could be effectively implemented by incorporating timer information within the lease manager itself. The advantage of using meters and plans, however, lies in added flexibility: the criteria for canceling a lease can be application-specific, and may include conditions beyond simple timer expiration, e.g., thread priority.

Meters and plans are partially inspired by *event-condition-action* (ECA) rules ("triggers"), an approach used mainly by database systems [11]. ECA rules check for the occurrence of events; at the event, if the condition is satisfied, the action takes places. Both academic and commercial databases use trigger mechanisms.

2.3 Synchronization

One of the problems with this form of aspect is that execution is somewhat non-deterministic; specifically, it is difficult to predict exactly when (if ever) the advice will be executed. On the other hand, advice should not execute unless any structures it is operating on are in a stable state. For example, if the advice is going to alter the representation of a data structure, it is important to ensure that no other code tries to alter that data structure while the advice executes. To put it another way, because the advice is executed asynchronously, our techniques brings along with it all of the difficulties of concurrency.

One way to formulate this problem is as an attempt to execute advice asynchronously *while also ensuring that the execution of the advice is transparent to the rest of the program.* There has been some work in ensuring that asynchronously executing code can be executed transparently [8], but only for introducing additional concurrency to single-threaded code, not for code that was originally intended to execute asynchronously.

AspectJ does not typically encourage synchronization in aspects. It offers a `Coordinator` package, which allow traditional Java synchronization to be applied to a pointcut of the program. Our proposed functionality is similar to this: the `mutex` qualifier on a plan prevents that plan from executing concurrently with its parameter, a join point.

Our system uses a **mutex** qualifier to ensure that it is not executing concurrently with a given join point (or set of join points). It should be stressed that **mutex** is not the same as `synchronized`; the `synchronized` modifier would not only prevent the advice from executing concurrently with the join points, it would also prevent the join points from executing concurrently with each other.

The following code is an example of a plan that can only occur when the methods `o.foo` and `p.bar` do not execute:

```
plan Tick:
  meter file evaluate(check());
  mutex o.foo(), p.bar();
```

3. WEAVING

Obviously, there is a major problem with weaving an aspect that is intimately tied to the dynamic state of the program. Few of the meters we have discussed here can be resolved statically; it is impossible, for example, to determine high load levels when a program is not executing. Meters, therefore, may be triggered at any point in the program. However, it is obviously undesirable to check every meter after any instruction is executed.

This problem is not exclusive to our work. More traditional dynamic aspects can use (for example) `cflow` pointcuts, which execute advice based on whether the control flow of the program is within a given join point, require checks to indicate whether or not they are, in fact, in that join point. Exceptions in Java provide a similar problem: because most bytecodes can throw exceptions, it is necessary to check for exceptional conditions frequently.

Our first implementation is ongoing work; it involves weaving dynamically. Typically, static optimization is used to reduce the number of checks in a program. For dynamic aspects, Masuhara *et al.* [5] employ a technique based on partial evaluation that can limit the number of unnecessary run-time checks. There has also been a great deal of work in reducing the number of necessary checks for exceptions in Java [2, 10].

Unfortunately, the properties that we are checking for cannot, in general, even be approximated statically. For example, a plan that executes on high CPU load might be triggered at any point in the program. Fortunately, most of the potential meters do not require such broad support. For example, a check for low memory can simply be triggered on each allocation (when a check for low memory must be performed in any case).

4. FAULT RECOVERY IN Rescue

As insufficient as the approaches for fault determination in modern programming languages are, the mechanisms available for fault recovery are worse: in fact, they are generally non-existent. All of the responsibility for failure recovery is in the hands of the programmer. However, it is generally accepted that programmers are very bad at planning for failure recovery [9]. This has a dramatic effect on software quality; in real environments, errors occur, and programs are not prepared to deal with them adequately. Other than doing nothing or killing a computation, applications have relatively few choices on how to recover from faults, especially those triggered by outside events. In distributed and grid environments, load balancing monitors can trigger offloading computation from one node to another. A more aggressive strategy, and one rarely supported by most implementations, is to alter the computation so that it adapts to the fault. For example, if a program has a memory-intensive data structure, and is running low on memory, it may be worthwhile to alter the data representation so that it is more compact (perhaps at the expense of another resource, such as CPU usage). Little support for these recovery mechanisms is available in modern programming languages. In fact, the programmer usually must implement them manually. This provides a great deal of flexibility. For example, if a computation must be undone, it is usually possible to undo only what needs to be undone, saving a great deal of time with checkpointing and logging the program's effects.

We provide some ideas here about how to support taking the burden of recovery off of the programmer, in combination with the programmatic techniques discussed in Section 2. Our research focuses on several different strategies: (i) support for efficient checkpointing; (ii) transactional features; (iii) task replication. We discuss (i) and (ii) below; the description extends naturally to (iii). We discuss this work in the context of the Ovm virtual machine [7], an implementation of the Real-Time Specification for Java [1].

4.0.0.1 Checkpointing.

One way to provide fault recovery is to checkpoint program state periodically. When done naïvely, checkpointing can be expensive, potentially requiring the copying of large amounts of state. Aspects provide one way to checkpoint relevant data *selectively.* For example, consider an application performing an iterative fixpoint calculation over a complex data structure D. Modifications to D may occur by different threads and in different parts of the program. Nonetheless, every such modification is guaranteed to be relevant. Checkpointing D whenever it is updated would thus allow tasks that fail when faults arise to restart with latest checkpointed version of D.

The code fragment shown below captures this functionality. A *write barrier* allows the execution of specified code whenever a memory store occurs. Write barriers can be prefixed with the name of the object to be monitored. A meter can be associated with a write barrier. When a write occurs on file `f` of a particular class, but before it takes effect, the current version of the structure is saved; this version can be restored if the computation must be restarted:

```
onwrite int ClassName::f when
  within(fixpointTask)  {
    save(old(f));
    return f = new(f);
}
```

This interface is already available in the Ovm customizable virtual machine, but it requires deep knowledge of the internals of the VM to use. We will make it more accessible. Read barriers are also supported. In both cases users should be extremely careful with the use of this API due to obvious performance implications. Another example of barriers for implementing replication is as follows:

```
onwrite int *::_ {
  buffer[i++]= old(_);
  if (i==buffer.length) {
    atomic { flush(); }
  }
}
```

This barrier is triggered for any integer field write. The function `flush()` tries to propagate the changes to other machines in the cluster. It is only called when the buffer is full. The `atomic` keyword is a low-level feature of the VM which essentially turns off scheduling.

4.0.0.2 Transactions

A transaction is a sequence of operations that is performed atomically: either all of it is seen to have been performed, or none of it is. If it completes successfully, it *commits*; otherwise, it *aborts*, in which case none of the updates it made are seen.

Rescue should provide direct support for transactions. In the scope of the transaction, if a write occurs, the original value of the heap location is written to a log. When a fault occurs, an earlier program state can be re-established by restoring the original values from the log. Program flow can be resumed either from the beginning of the transaction (with some new compensatory code in place) or continued as if the error did not occur.

Transactional execution is supported in Rescue by using lightweight language-based transactions. A Rescue program could use aspects to attach transactional support to a computation and has programmatic control over their semantics. To give an example, consider a method `evaluate()` in a class `Task`. This method is called once for every task and encapsulates most of the computation it performs. In the case that the system runs out of memory, we would like to abort the task. This can be done by a combination of transactions and plans. The first piece of the puzzle is to attach a transaction to the code. This can be done by attaching pre- and post- actions to the `evaluate()` method.

```
rescue.transaction transaction;

before(): Task.evaluate() {
   // start a transaction before each call
   transaction = new Transaction(solution);
   // to evaluate()
}

after(): Task.evaluate() {
  if (!transaction.aborted())
    transaction.commit();
}
```

These methods start a transaction. Note that we give an argument to the constructor, the object `solution`, to indicate the transactional root. All objects transitively reachable from that root will be logged. We then define a method that gets triggered when the system gets low on memory. This method obtains the transactional log, *extracts* the contents from the log (i.e. the objects that have been modified during the transaction), and inspects their values. Finally the transaction is aborted.

```
void handleOOM() when} LowOnMemory {
 Object root = transaction.inspect();
   // obtain the contents of the log
 copyPartialResults(root);
   // user procedure to capture
   // partial results
 transaction.abort();
   // throw away changes
}
```

The default semantics of our transactional mechanism is to log all reads and writes, modulo compiler optimizations. More discriminate policies can be implemented with the low-level read/write barrier interface.

Fig. 2 illustrates how transactional support interacts with Rescue's recovery mechanism. The figure depicts several meters and plans that monitor memory usage. When available memory becomes low (less than 5MB), thread actions are monitored are recorded in a transaction log. If memory use continues to increase, and falls below a critical threshold, an abort action is triggered, and the effects of the thread are discarded using the contents of the transaction log to restore original values. If memory use falls and exceeds a safety threshold (here $>$ 10MB), the contents of the log are committed. Rescue's design uses plans and advice, combined with transactional mechanisms, to allow computation effects to be propagated only when safe to do so; this is a distinguishing characteristic and central to the contributions of the proposed research.

Our proposal is designed to support fault recovery in Rescue by using transactions in conjunction with programming language support for plans and meters. We will also investigate a number of other fault recovery mechanisms, including replication and task migration.

5. CONCLUSION

In this paper we have presented a method for expressing hooks into a virtual machine to provide fault determination and fault tolerance for user code. The resulting language infrastructure allows for a more nuanced approach to handling faults; in essence, we provide an asynchronous construct that can be woven into code at runtime and gracefully handle exceptional conditions. We discuss how to combine this approach with fault recovery mechanisms that may take much of the burden of fault recovery off of the programmer.

6. REFERENCES

[1] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The*

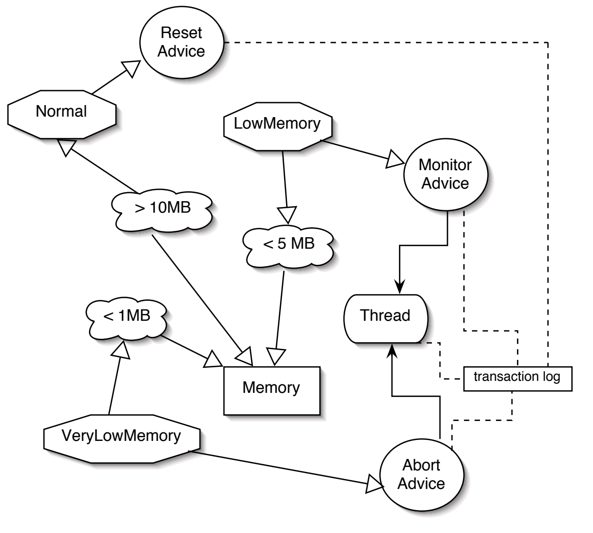


Figure 2: Advice and transactions. Clouds represent meters that monitor resources for particular conditions. Octagons represent plans that are triggered when their associated meters are satisfied. Circles denote advice associated with plans.

Real-Time Specification for Java. Java Series. Addison-Wesley, June 2000. `www.javaseries.com/rtj.pdf`.

[2] Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap. *SIGPLAN Notices*, 35(11):139–149, 2000.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary.* Springer-Verlag, June 2001.

[4] Barbara Liskov and Alan Snyder. Exception Handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979.

[5] Kidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 17–26. Department of Computer Science, Iowa State University, April 2002.

[6] Sun Microsystems. Jini Network Technology version 2.0, http://www.jini.org, June 2003.

[7] Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi, and Jan Vitek. The OVM Customizable Intermediate Representation. *To appear in The Science of Computer Programming*, 2005.

[8] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for Java futures. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Appilcations (OOPSLA)*, pages 206–223, October 2004.

[9] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 419–431, October 2004.

[10] John Whaley. Dynamic Optimization through the Use of Automatic Runtime Specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[11] J. Widom and S. Ceri. Introduction to active database systems. In J. Widom and S. Ceri, editors, *Active Database Systems - Triggers and Rules for Advanced Database Processing*, pages 2–41. Springer, Berlin,, 1996.

Dynamic Weaving in CAM/DAOP: An Application Architecture Driven Approach

Lidia Fuentes
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga, SPAIN
lff@lcc.uma.es

Mónica Pinto
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga, SPAIN
pinto@lcc.uma.es

Pablo Sánchez
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga, SPAIN
granpablo@lycos.es

ABSTRACT

Dynamic weaving is much more flexible than static weaving because the separation of concerns remains at runtime. This results in highly configurable and adaptable applications, since the rules that govern the weaving of aspects can evolve during the application execution, according to different criteria – i.e. user preferences, execution context, etc. In this paper we describe the dynamic weaving mechanism offered by CAM/DAOP, our own component and aspect platform. The most relevant feature is that the plugging of components and aspects is driven by the application architectural information, which is loaded into the internal structures of the CAM/DAOP platform to be consulted at runtime.

1. INTRODUCTION

During the last years, Aspect-Oriented Software Development (AOSD) [2] became a more and more consolidated software technology. Hundreds of new aspect-oriented approaches appeared that cope with the separation of concerns principle in different ways. In this paper, our main interest rely on the different weaving processes, which can be static (performed during compilation) or dynamic (performed at runtime).

In those approaches where the weaving process is static [5, 8, 10] the object/component and aspect code is mixed at compile-time. Static composition provides high performance, but separation of concerns is lost at runtime. Although they normally use introspection to provide reflective information about join points at runtime, the number and type of join points affected by an aspect cannot be modified after compilation.

Dynamic weaving is an interesting alternative to static weaving. It is much more flexible than static weaving because the separation of concerns remains at runtime, enabling, in some cases, the late binding between objects/components and aspects. Approaches that offer a dynamic weaving mechanism [17, 11] are mainly based on a reflection mechanism that offers the ability to modify the application semantics while the application is running. This adaptability is commonly achieved by implementing a Meta Object Protocol (MOP) as part of the language interpreter that specifies the way a program may be modified at runtime.

In this paper we describe the dynamic weaving mechanism of CAM/DAOP [13]. CAM (Component-Aspect Model) is a new component and aspect model that defines components and aspects as first-order entities. The underlying infrastructure supporting the CAM model is a Component-Aspect Platform (DAOP, a Dynamic Aspect-Oriented Platform) where the plugging of software aspects into components is performed at runtime.

One of the most relevant features of our approach is that the dynamic weaving among components and aspects is driven by the information about the software architecture of the application. Concretely, CAM/DAOP uses an XML-based architectural description language (DAOP-ADL) [14] to describe components and aspects, together with the composition rules (i.e. the declaration of aspect pointcuts) that govern the weaving of components and aspects. The platform weaving mechanism loads and consults this information at runtime to establish the connections among components and aspects. This is particularly useful because we make components and aspects much more reusable, isolating the dependencies between them in the platform internal structures. In addition, this information can be easily adapted at runtime, improving the flexibility and adaptability of the final application.

The complete description of CAM/DAOP and the DAOP-ADL language is beyond the scope of this paper and can be found in [13, 14]. In this paper we focus on describing the dynamic weaving mechanism offered by the DAOP platform, and the main advantages obtained from consulting the application architectural information at runtime. After this introduction, the paper is organized as follows. Next section compares other AOSD approaches offering dynamic weaving. Then, section 3 describes the architecture of the DAOP platform and section 4 the DAOP dynamic weaving mechanism. In order to cope with the limitations introduced by dynamic weaving, in section 5 we describe our approach to cope with these limitations in CAM/DAOP. Finally, we present our main conclusions in section 6.

2. RELATED WORK

Table 1 contains a brief description of several AO frameworks and platforms providing dynamic composition. Regarding the main features of CAM/DAOP, we have analyzed the different works particularly interested in: (i) how they incorporate the component concepts; (ii) the separation (or not) of advice and pointcuts in isolated entities; allowing the reuse of aspects, (iii) the mechanisms to express pointcuts; with special interest in if they describe in some way architectural information (iv) if they use an invasive or non-invasive model, and (v) the mechanisms they use to perform dynamic weaving.

Table 1: Related Work in Dynamic AOSD Approaches

	PROSE	**JAC**	**Aspect Werkz**	**JAsCo**	**Lasagne**	**AOP JBoss**	**CAM/ DAOP**
I	No	No	No	Yes (Beans)	Yes	J2EE	Yes
II	No	Yes	Yes	Yes	Yes	Yes	Yes
III	No	.acc or XML files	XML files	No	Composition policy file	XML files	XML files
IV	Yes	Yes	Yes	No	No	Yes	No
V	JVM Class Loader/ HotSwap2	Wrappers + Container	HotSwap / JSR-163	Hotswap + Connector Registry	Decorator-like wrappers	Container	Middleware Layer

(I) CBSD concepts (II) Separation od advice and pointcuts (III) External Configuration of pointcuts (IV) Invasive model (V) Dynamism

PROSE [15] is an AO platform with dynamic composition, for using aspects with objects. Aspects can intercept points that are part of the internal behavior of objects. Its main contribution is that the platform weaves and unweaves aspects directly in the Java Virtual Machine (JVM), inserting the aspect advice directly into the native code generated by the just-in-time (JIT) compiler. In addition, pointcuts and advice are implemented in PROSE in the class representing the aspect, with the drawback of reducing the (re)use of the aspect advice.

Another similar approach is JAC [11], an AO framework that uses the reflexive API BCEL for adding aspects. Aspects in JAC are dynamically deployed and undeployed on top of running application objects using wrappers and aspect containers. JAC pointcuts are not specified as part of the aspect definition but in a third-party entity available at runtime, making aspects more reusable. In addition, JAC uses AspectComponent configuration files (.acc files) or XML files to configure externally the aspect evaluation rules (or pointcuts). It does not really define a component platform and its components cannot be considered software components in the CBSD sense. JAC distributed protocols are introduced as an extra mechanism to be able to distribute aspects in different hosts.

Another AO framework that performs dynamic composition of objects and aspects is AspectWerkz [3], which implements several weaving techniques. It offers static weaving like AspectJ [5], and other based on different mechanisms: JSR-163 JVMTI, hotswap, and bootclasspath. Using these technologies, aspects are composed with objects at runtime, by modifying objects byte code after class loading. Uses XML files to define pointcuts separately from aspect implementations. However, aspects in AspectWerkz are applied to objects and not to components, defining an invasive model. Its weaving mechanism completely relies on the Java technology.

JAsCo [17] is an aspect oriented implementation language that defines a new component model compatible with the JavaBeans component model. Aspects in JAsCo can be applied, adapted and removed at runtime. JAsCo introduces two concepts: aspect beans that encapsulate advice, and connectors that define pointcuts. Both advice and pointcuts can evolve separately, increasing the reuse of advice. Dynamic connector loading and unloading is possible in the JAsCo connector registry. However, JAsCo connectors have to be compiled, reducing their runtime adaptation. Aspects interception is performed only before or after a method execution, defining a non-invasive model, although an aspect can replace the normal execution of a method.

Lasagne [18] defines a platform-independent architecture for dynamic customization of component-based distributed systems using decorator-like wrappers, according with a non-invasive model, that only intercept incoming/outcoming messages. In Lasagne, the composition logic is completely separated from the code of the components, and of the extensions as well, increasing their reuse. This information is specified in composition policy files that can be dynamically attached to the system. Lasagne composes extensions at the instance-level instead of at the class-level, giving Lasagne a runtime performance overhead, although the composition mechanism is much more flexible.

Finally, the JBoss AOP [1] framework is built on top of the JBoss J2EE application server, and tries to solve the limitation of providing just a set of built-in services. Aspect advice in JBoss AOP is implemented using interceptors, according with a invasiveness model, which is not required for components models. The JBoss AOP framework has the advantage that it separates advice and pointcuts in different entities, where pointcuts are configured using XML descriptor files.

There are other dynamic approaches not covered in this section due to the lack of space such as JMangler [6], Caesar [9], EAOP [4], Rapier-LOOM.NET [16] and Weave.NET [7], among others.

3. THE DAOP PLATFORM

The DAOP platform is a distributed component-aspect middleware platform for running applications conforming to the CAM model [13]. Figure 1 shows its architecture, which contains information about the services and facilities it offers to components and aspects (elements that appear above the DAOP Platform class in figure 1), together with the information the platform stores to provide such services (classes below the DAOP Platform class in figure 1).

Similar to other component platforms, the DAOP platform provides a set of common services to develop distributed applications, such as the instantiation of components (Com-

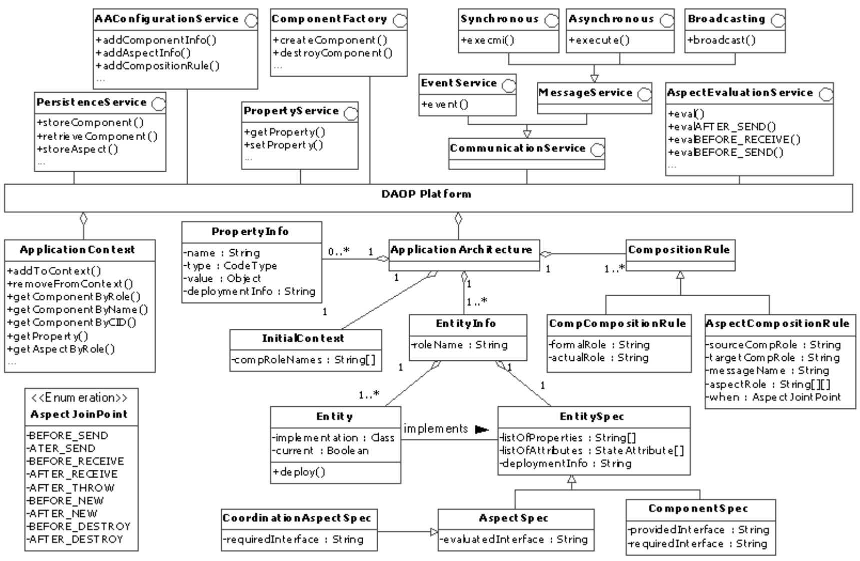


Figure 1: The DAOP Platform Architecture

ponentFactory interface), the communication of components (CommunicationService interface), the evaluation of aspects (AspectEvaluationService interface), the storage of properties (PropertyService interface), the persistence service (PersistenceService interface) and the dynamic adaptation of the application architecture (AAConfigurationServices interface).

Regarding the internal infrastructure of the DAOP platform its information is arranged basically in two objects. The ApplicationArchitecture object, which stores the architectural description of the application; and the ApplicationContext object, which holds the current list of component, aspect and property instances.

The DAOP model in figure 1 is a platform independent model that may be implemented using different middleware technologies, such as .NET, CORBA or Java/RMI. Currently, we have a Java implementation that uses Java/RMI as the base communication mechanism, and the reflective package to help us to implement dynamic composition.

Regarding other AOSD approaches the main advantage of CAM/DAOP is that it supports the development of peer-to-peer distributed applications, since DAOP is neither a client/server approach nor uses a central manager to be notified of messages and events that occur within the application. Instead, the DAOP platform is a distributed platform that do not need to define extra mechanisms to distribute aspects in different hosts. An application in DAOP is distributed among different hosts, where a local instance of the DAOP platform is running. These DAOP platform instances communicate among themselves, being possible for all the components and aspects in a DAOP application to communicate and collaborate amongst themselves. During the deployment of the application it is determined the number of instances that the DAOP platform creates for each aspect and how these instances are distributed. Also components are distributed through the different nodes of the DAOP application.

Other relevant advantage of DAOP is that components and aspects are plain code in the language where the platform was implemented. For instance, they are plain Java code in the current Java/RMI implementation. Neither the use of new constructions nor the generation of stubs and skeletons are needed in order to implement DAOP components and aspects. Only the use of the services offered by the DAOP platform are needed. Other advantages of CAM/DAOP are shared with other AOSD approaches, such as: (1) the application of aspects to components [1, 17, 18] instead of objects [3, 11, 15]; (2) the definition of a non-invasive model similar to most component-based approaches [17, 18], where it is not possible to intercept join points that are part of the internal behavior of a component. Instead, only the behavior exposed through the explicit interfaces of components can be intercepted, considering components as black-box entities, and (3) aspects are applied to components at runtime, and the information needed to perform the dynamic weaving of components and aspects is described using a declarative language such as in [3, 18] and is not hard coded as part of the application implementation classes.

4. DAOP DYNAMIC WEAVING

In this section we will explain the main features of the DAOP dynamic weaving mechanism. The weaving in DAOP is dynamic since components and aspects remain as separate entities during the application execution. By aspect weaving we mean the execution of the corresponding aspect advice when a join point is intercepted by the DAOP platform.

Even though the weaving mechanism in DAOP is based on the interception of messages and events, there is an important difference between DAOP and traditional component platforms, such as CORBA, CCM/CORBA and EJB/J2EE. Whereas these platforms offer a concrete number of services that cannot be extended by users, in our approach it is possible to separate any crosscutting property. The difference can be found in how both approaches manage these properties. In our approach the provision of these properties does not rely on the platform provider, like in CORBA, CCM/CORBA and EJB/J2EE. Instead, components and aspects in DAOP are first-class entities that coexist at the application level. Consequently, using CAM/DAOP application's developers decides how to divide the application functionality into components and aspects, with no limit to the kinds of aspects.

In the rest of this section we will give specific details about the DAOP dynamic weaving mechanism. For that, we will use the example showed in figure 2.

This example is taken from a virtual office application we have developed. In this application users (User component) join a shared space (VirtualOffice component) to collaborate with other users. The figure shows that the user is authenticated before joining the office, by evaluating the Authentication aspect before the *join* message is sent by the User component (BEFORE_SEND join point). Additionally, when the user leaves the office, the Persistence aspect is evaluated after the *leave* message is received by the VirtualOffice component (AFTER_RECEIVE join point), to make persistence the status of the office for that user. This status can be restored the next time the user joins the office, by evaluating the Persistence aspect after the *join* message is received by the VirtualOffice component (AFTER_RECEIVE join point).

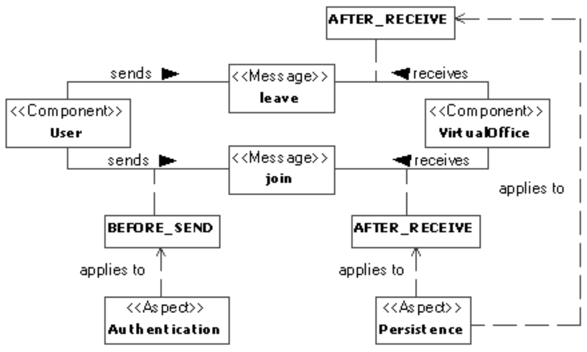


Figure 2: A CAM/DAOP Example

We have used the CAM model to design the example in figure 2. Although the CAM model has not been described in this paper, in order to understand the example we only need to know that the CAM model describes components (with stereotype ⟨⟨Component⟩⟩), aspects (with stereotype ⟨⟨Aspect⟩⟩) and the composition among them, expressed in terms of applies to relationships. These relationships describes, in this example, which aspects are applied when a message (with stereotype ⟨⟨Message⟩⟩) is sent or received. The join points are the same described in the previous section.

From the point of view of the application developer, the main step in the development of a CAM/DAOP application – i.e. related to the dynamic weaving mechanism, is the description of the application architecture. From the point of view of the platform, the main step is the use of the information about the application architecture to perform the dynamic weaving of components and aspects.

4.1 Description of the Application Architecture

Step 1: Describe the application architecture information, including the declaration of aspect's pointcuts

As we mentioned in the introduction, an important feature of our approach is that the DAOP platform stores a description of the application architecture (AA), which is consulted by the platform to perform component and aspect instantiation and dynamic weaving. This information is specified during the design and architecture phases using the DAOP-ADL [14] language, an XML-based architectural description language. Then, during the execution, when an instance of the DAOP platform is created, the XML document is parsed and the structure of the CAM/DAOP application is stored in the ApplicationArchitecture class and its subclasses (see figure 1).

With this language, firstly, all the components and aspects that could be instantiated in the application are described; each of them identified by its role name. Role names are identifiers that we use in CAM/DAOP to uniquely identify both components and aspects. These role names are architectural names that are used for component-aspect composition and interaction, allowing loosely coupled communication among them – i.e. no hard-coded references need to be used for exchanging information, but just a role name identifying the source and the target of a message.

Then, the aspect composition rules, – ie. pointcuts, defining when and how to apply aspects to components are described, where components and aspects are referred by their corresponding role names. In this example, we have two components with role names "user " and "office", and two aspects with role names "authentication" and "persistence". Aspect composition rules are stored in the AspectCompositionRule class in figure 1.

There are three kinds of aspect composition rules. In the following code, we have expressed these rules in EBNF in order to better explain which kind of information can be provided to define pointcuts and, even more important, which kind of information can be adapted to modify component and aspect weaving at runtime.

The first kind of rule describes which aspects are applied when components communicate by sending messages:

```
1 <message_pc>   ::= <message_jp> <message_description>
                     '{'<aspect_composition>'}'
2 <message_jp>   ::= BEFORE_SEND | AFTER_SEND | BEFORE_RECEIVE |
                     AFTER_RECEIVE | AFTER_THROW
3 <message_description> ::= <source> <target> <message>
4 <message>      ::= [<TYPE>] <Ident> '(' (<TYPE>)* ')'
5 <source>       ::= <rolename> [<message>]
6 <target>       ::= <rolename>
7 <aspect_composition> ::= '{' sequential_aspects '}' |
                           '{' sequential_aspects '}'
                           <aspect_composition>
8 <sequential_aspects> ::= (<aspect>)+
```

Notice that the description of the intercepted message (see line 3) includes the source and the target components, identified by their role names. In order to make the definition of pointcuts much more flexible, the source of a message includes not only the role name of the source component but, optionally, the definition of the method from which that message was sent (see line 5).

The aspects to be evaluated are described using a bi-dimensional array of strings with the format $\{\{A_1\},\{A_2\}, \{\{A_3,A_4\}\}$ (see lines 7 and 8) where every A_i is an aspect role name. This bi-dimensional structure allows us to specify two kinds of aspect evaluation: sequential evaluation and parallel evaluation. Aspects enclosed in the outer brackets, for instance A_1 and A_2, are evaluated sequentially. On the other hand, aspects in the inner brackets, for instance A_3 and A_4, will be evaluated concurrently.

Coming back to our example there are three pointcut declarations that correspond with this kind of rule:

```
1 BEFORE_SEND   user * office join(String)  {{authentication}}
2 AFTER_RECEIVE user * office join(String)  {{persistence}}
3 AFTER_RECEIVE user * office leave(String) {{persistence}}
```

which can be interpreted as follows: (1) *before sending* the message *join(String)* from the component with role name *user* to the component with role name *office* the aspect with role name *authentication* must be evaluated. The wildcard "*" indicates that this rule is applicable independently of the method in the source component from which the *join(String)* message is sent; (2) *after receiving* the same message with the same source and target components the aspect with role name *persistence* has to be evaluated, and finally (3) *after receiving* the *leave(String)* message, being the source and target components the same, the aspect with role name *persistence* is applied once again.

There are other kind of rule to describe which aspects are applied when components communicate by throwing events:

```
 9 <event_pc>      ::= <event_jp> <event_description>
                       '{' <aspect_composition> '}'
10 <event_jp>      ::= SEND_EVENT
11 <event_description> ::= <source> <message>
```

This kind of rule only differs from the previous one in that the description of events (see line 11) does not include its target.

Finally, the last kind of rule describes which aspects are applied when components are instantiated or eliminated from the system:

```
12 <component_pc> ::= <component_jp> <component_description>
                      '{' <aspect_composition> '}'
13 <component_jp> ::= BEFORE_NEW | AFTER_NEW | BEFORE_DESTROY |
                      AFTER_DESTROY
14 <component_description> ::= <source> <rolename>
```

The main advantages of describing the AA information as done in CAM/DAOP are the following:

Advantages

1. *The moment of the evaluation (when creating/destroying components and when sending and/or receiving messages and events), the kind of evaluation (sequential or parallel), and the information about which components are affected by aspects is not hard-coded as part of the component or aspect implementations. Instead, this information is taken out of components and aspects and stored in the* ApplicationArchitecture *class inside the DAOP platform, achieving a higher degree of component and aspect independence.*
2. *Pointcut declarations are not distributed through the different aspects in the application. Instead they are centralized in the document describing the AA. In consequence, designers and programmers are able to comprehend the structure of the application, facilitating the understanding and evolution of final applications.*

4.2 Dynamic Weaving of Components and Aspects

Step 2: At runtime, the DAOP platform consults the AA information that was previously stored in its internal structures

During the deployment phase, the document describing the AA is stored together with all the other application resources, i.e. component and aspect implementations, images, etc. Later at runtime, when a user join an application, the information about its AA is downloaded as part of the application specific applet, is de-serialized at the user site and the information stored in the platform.

Finally, this information is used by the platform to perform the dynamic plugging of components and aspects. This occurs when components create or destroy other components using the corresponding methods of the ComponentFactory interface (see figure 1). Also, it occurs when components communicate between them using the component communication primitives offered by the platform (see the CommunicationService interface and all its subinterfaces in figure 1). As in other component platforms (e.g., CORBA), DAOP allows components to send synchronous and asynchronous messages, as well as to broadcast a message to several target components. DAOP also allows components to throw events to other components.

By intercepting the throwing of events, CAM/DAOP provides a join point that occurs within the execution of a component method, similar to the *around* join point in other approaches like AspectJ or PROSE. The difference with these white-box approaches is that the DAOP platform can only intercept the points that the component makes visible throughout the throwing of events, considering it still a black-box component. The handling of events is resolved at runtime by a *coordination aspect*, which is out of the scope of this paper [12].

Coming back to our example, figure 3 describes how components and aspects are dynamically plugged when the component with role name user sends the join(String) message to the component with role name office; using for that the DAOP execmi() communication primitive (step 1 in figure 3).

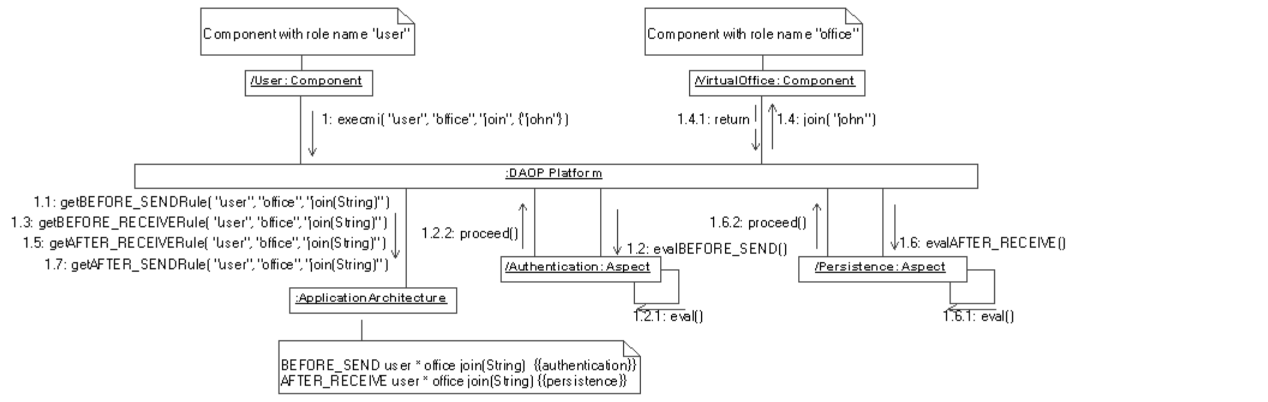

Component with role name "user"
/User : Component
1: execmi("user", "office", "join", {"john"})
Component with role name "office"
/VirtualOffice : Component
1.4.1: return
1.4: join("john")
:DAOP Platform
1.1: getBEFORE_SENDRule("user", "office", "join(String)")
1.3: getBEFORE_RECEIVERule("user", "office", "join(String)")
1.5: getAFTER_RECEIVERule("user", "office", "join(String)")
1.7: getAFTER_SENDRule("user", "office", "join(String)")
1.2.2: proceed()
1.2: evalBEFORE_SEND()
1.6.2: proceed()
1.6: evalAFTER_RECEIVE()
/Authentication : Aspect
/Persistence : Aspect
1.2.1: eval()
1.6.1: eval()
:ApplicationArchitecture
BEFORE_SEND user * office join(String) {{authentication}}
AFTER_RECEIVE user * office join(String) {{persistence}}

Figure 3: Dynamic Weaving Mechanism

At this moment, the DAOP platform takes the control and, by each intercepted join point, consults the information about the aspect pointcuts, which are stored in the :ApplicationArchitecture object (steps 1.1, 1.3, 1.5 and 1.7). As specified by these pointcuts, the platform invokes the evalBEFORE_SEND() method in the Authentication aspect (steps 1.2, 1.2.1, and 1.2.2), then it invokes the join("john") method in the target component (steps 1.4 and 1.4.1) and, finally, it invokes the evalAFTER_RECEIVE() method in the Persistence aspect (steps 1.6, 1.6.1 and 1.6.2).

In order to perform this evaluation, DAOP aspects must implement the AspectEvaluationService interface (see figure 1). This is mandatory since, as described above, the platform will invoke one of the methods in this interface to evaluate an aspect – i.e. to execute the aspect advice. There is a method in the AspectEvaluationService interface for each CAM/DAOP join points. All these join points are enumerated in the AspectJoinPoint enumeration class shown in figure 1. Additionally, CAM/DAOP also provides the eval() method, which is a general method that allows to make aspects completely independent from the intercepted join point. For instance, this is useful for a *Trace* aspect that records information about the application execution. Its behavior is always the same, independently of the join point where the aspect is evaluated.

The main advantages of the CAM/DAOP weaving mechanism are the following:

Advantages

1. *We close the usual "gap", or loss of information, between design and implementation levels, since exactly the same information generated during the description of the application architecture is then used at runtime*
2. *This is a real "runtime" weaving mechanism, not a "load time" one*
3. *During the execution it is possible to adapt the behavior of the application by adding, removing or modifying the information about the application architecture. The platform will assure that this information remains consistent if changes are performed by some user*

4.3 Dynamic adaptability of the Application Behavior

Step 3: Optionally, the application behavior can be adapted at runtime by modifying the information about the application architecture stored in the DAOP platform

Finally, if required by the application, the information about the application architecture that is stored in the internal structures of the DAOP platform can be adapted at runtime in order to modify the behavior of the application. This can be done without stopping or recompiling the application. In the Java implementation this also means that classes do not need to be loaded again with the Java class loader. New components and aspects can be incorporated to the application, or existing ones removed from it. Also, aspect composition rules can be modified, added or removed from the description of the application architecture, changing how the plugging of components and aspects is performed.

In order to do that the DAOP platform offers the AAConfigurationService interface (see figure 1). Currently, we are developing a system tool that will support any kind of application that desire to adapt its application architecture. Additionally, applications can incorporate their own tools to adapt it according to their specific necessities. In both cases, the AAConfigurationService interface must be used.

Advantages

1. *Final applications are more adaptable and evolvable*

5. IMPROVING PERFORMANCE: COMBINING STATIC AND DYNAMIC WEAVING

The main drawback of dynamic approaches is that in many cases static weaving is faster and therefore offers better performance than dynamic weaving. In this sense, in CAM/DAOP the use of reflection and the dynamic composition mechanism may introduce some overhead at runtime.

The performance study we have performed with CAM/DAOP has been using our Java/RMI implementation. Currently, we have a virtual office application[1], and after one year of evaluation we can state that the performance is satisfactory. Even when Java poses significant drawbacks related to efficiency, we find that the overload of dynamic evaluation of aspects is not so critical in distributed systems based on Java. For instance, component and aspect creation through the platform takes 30 ms, and the time to incorporate the evaluation of the aspect at runtime is insignificant (around 20 ms). Comparing this evaluation time with the time spent loading a web page from the same host, they are insignificant.

Nevertheless, and taking into account that there are aspects that might not need to be adapted at runtime, we are now developing a alternative static composition mechanism using the Java BCEL API. One of the goals of this extension is to be able to study these run-time overheads to check if they are really relevant to the application performance and how we are able to reduce the overhead in some situations. Extending the DAOP-ADL language to specify whether an aspect must be woven into components statically or dynamically, this tool manipulates component class files to weave static aspects at compile time. Additionally, it generates a new DAOP-ADL document which only contains the declaration of pointcuts for dynamic aspects. Therefore, dynamic aspects will be weaved at runtime as described previously.

Using this approach we sacrifice the independence of aspects and components at implementation level, in order to increase the application's performance. Independence is still maintained at design and architectural levels.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented the dynamic weaving mechanism of CAM/DAOP. The main contribution of this mechanism is that it performs the plugging of components and aspects using the information about the application architecture. This information is described during the architecture description phase using the DAOP-ADL language, and includes the definition of aspect's pointcuts. During the application instantiation, pointcuts are loaded in the internal structures of the DAOP platform, to be consulted at runtime. This have two important advantages: (1) aspects are more reusable in different contexts since they do not include the pointcut declaration, and (2) we bridge the gap between design and implementation, since the weaving of components and aspects is performed according to exactly the same information that was provided during the design and architecture phases.

Additionally, aspect's pointcuts can be added, modified or deleted at runtime without stopping or (re)compiling the application. The dynamic weaving mechanism offered by the platform allows that in order to automatically adapt the application behavior, only the information about the application architecture was modified, using for that the corresponding service of the DAOP platform.

7. ACKNOWLEDGMENTS

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

[1]see http://150.214.108.46/CoopTEL

8. REFERENCES

[1] The aspect oriented programming and jboss tutorial. http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html.

[2] Aspect-oriented software development web site. http://aosd.net.

[3] Aspectwerkz web page. http://aspectwerkz.codehaus.org/.

[4] Event-based aspect-oriented programming. http://www.emn.fr/x-info/eaop/tool.html.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of ECOOP'01*, pages 327–355, Budapest, Hungary, 18-22 June 2001. Springer-Verlag.

[6] G. Kniesel, P. Costanza, and M. Austermann. Jmangler - a powerful back-end for aspect-oriented programming. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-oriented Software Development*. Prentice Hall, 2004. To appear.

[7] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *Proc. of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, California, USA, 2003.

[8] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, Sept. 2003.

[9] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proc. of the Second International conference on AOSD*, pages 90–100, Boston, MA, 17-21 March 2003. ACM Press.

[10] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In M. Aksit, editor, *Software Architectures and Component Technology*. Kluwer Academic Publishers, 2001.

[11] R. Pawlack, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible and efficient framework for AOP in Java. In *Proc. of Reflection'01*, Kyoto, Japon, 25-28 September 2001. Springer-Verlag.

[12] M. Pinto, L. Fuentes, M. Fayad, and J. M. Troya. Separation of coordination in a dynamic aspect-oriented framework. In *Proc. of the First International Conference on AOSD*, pages 134–140, Enschede, The Netherlands, 22-26 April 2002. ACM Press.

[13] M. Pinto, L. Fuentes, and J. M. Troya. A dynamic component and aspect platform. *The Computer Journal*, Accepted for Publication.

[14] M. Pinto, L. Fuentes, and J. M. Troya. DAOP-ADL: An architecture description language for dynamic component and aspect-based development. In *Proc. of the Second International Conference on GPCE*, pages 118–137, Erfurt, Germany, 22-25 September 2003. Springer-Verlag.

[15] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proc. of the First International Conference on AOSD*, pages 141–147, Enschede, The Netherlands, 22-26 April 2002. ACM Press.

[16] W. Schult and P. Trger. Loom.net - an aspect weaving tool. In *Proc. of the ECOOP 2003 Workshop on Aspect-Oriented Programming*, Darmstadt, June 2003.

[17] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: An aspect- oriented approach tailored for component based software development. In *Proc. of the Second International conference on AOSD*, pages 21–29, Boston, MA, 17-21 March 2003. ACM Press.

[18] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Joergensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. of the 23rd International Conference on Software Engineering*, pages 233–242, Toronto, Canada, 12-19 May 2001. IEEE Computer Society.

A Source-level Kernel Profiler based on Dynamic Aspect-Orientation

Yoshisato YANAGISAWA
Tokyo Institute of Technology
2-12-1-W8-50 Ohkayama, Meguro-ku
Tokyo 152-8552, JAPAN
yanagisawa@csg.is.titech.ac.jp

Shigeru CHIBA
Tokyo Institute of Technology
2-12-1-W8-50 Ohkayama, Meguro-ku
Tokyo 152-8552, JAPAN
chiba@acm.org

Kenichi KOURAI
Tokyo Institute of Technology
2-12-1-W8-50 Ohkayama, Meguro-ku
Tokyo 152-8552, JAPAN
kourai@csg.is.titech.ac.jp

ABSTRACT

We present a source-level kernel profiler named KLAS. Since this profiler is based on dynamic aspect-orientation, it allows the users to describe any code fragment in the C language. That code fragment is automatically executed for collecting detailed performance data at execution points specified by the users. Enabling dynamic aspect-orientation is crucial since otherwise the users would have to reboot an operating system kernel whenever they change aspects. Although KLAS dynamically transforms the binary of a running operating system kernel for weaving an aspect at runtime, unlike other similar tools, the KLAS users can specify those execution points, that is, joinpoints through a source-level view. For example, the users can describe a pointcut that picks up accesses to a member of a structure; they do not have to explicitly specify the addresses of the machine instructions corresponding to the member accesses. We have implemented this feature by extending a C compiler to produce augmented symbol information. KLAS has been implemented for the FreeBSD operating system with the GNU C compiler.

1. INTRODUCTION

During the history of operating systems (OS), performance tuning of OS kernels has been an important topic for kernel developers. Even nowadays, the kernel developers are making serious efforts to run the OS kernel as fast as possible. They are still improving scheduling algorithms, implementation of network stack, lock mechanism, and so on. For example, both Linux and FreeBSD recently introduced new implementation of their process schedulers.

Investigating a performance bottleneck is the important first step for improving the performance of OS kernels. To do this, using a sophisticated performance profiling tool for OS kernels is mandatory. Here "sophisticated" means that the profiler reports not only the number of calls to each function constituting an OS kernel but also more detailed data specified by the users. In fact, we are studying a performance bottleneck of network processing of the FreeBSD operating system since we observed inappropriate behavior of the network module when multiple processes are simultaneously executing network I/O operations. We need a profiler that allows us to produce a log message including time stamp at any execution points (a.k.a joinpoints in AOP) that we specify.

This paper presents our kernel profiler called *KLAS*. It is a dynamic aspect-oriented system and allows the users to execute a code fragment as advice at specified execution points in an OS kernel. The advice is normally used to record time stamps but it can also executing any code written in the C language, for example, printing a log message. Since KLAS is a dynamic aspect-oriented system, the users can dynamically weave an aspect with a running OS kernel. Enabling dynamic weaving is crucial since the users can avoid rebooting a OS kernel whenever they change an aspect during investigation of kernel performance. For KLAS, we have developed a new implementation technique for dynamic aspect-oriented systems. KLAS replaces machine instructions in an OS kernel with breakpoint-trap instructions so that advice is woven at the address of those instructions. To enable the users to specify pointcuts with a source-level view, such as accesses to a member of a structure, we modified a C compiler so that it will produce extra symbol information. KLAS uses this extra information to identify the machine instructions that correspond to the specified pointcut. We implemented KLAS for the FreeBSD operating system with the GNU C compiler.

The rest of this paper is organized as follows. Section 2 describes requirements for kernel profilers. Section 3 presents our new implementation technique for dynamic aspect oriented programing (AOP). It also shows an overview of the current implementation of KLAS. Section 4 compares KLAS and other systems, including AOP systems and non-AOP systems. We conclude this paper in section 5.

2. REQUIREMENTS

To investigate a performance bottleneck, using a performance profiling tool is mandatory; in particular, a tool that can measure the elapsed time between interesting execution points in the OS kernel is useful. However, existing tools or techniques do not satisfy our requirements for investigating kernel performance. Since modern OS kernels are implemented with object orientation in the C language, a number of interesting execution points are calls to functions specified by function pointers. That profiler does not support such execution points; it only supports functions statically resolved. We below mention our requirements for such a kernel profiler.

First, the kernel profiler must enable the users to measure elapsed time between given two execution points. The users must be able to give those execution points in the kernel at runtime and change them, if necessary, without rebooting the kernel. The ability to change the execution points is crucial. The users would first measure the execution time of a large code section and then they would gradually narrow the range of that code section to find a performance bottleneck. Since rebooting the whole kernel is a time consuming task, frequent rebooting significantly decreases our productivity. Rebooting also clears the whole memory image and thus the internal data of the network module. After rebooting, the behavior that the users want to investigate might disappear. Furthermore, the code snippet for measuring the elapsed time must be given by the users since the users may want to measure the elapsed time between the execution points in which a certain variable holds a specific value. To do this, the measurement code must check the runtime value of that variable but only the users can give such code depending on a particular use case. Also, the users may want to print a log message, for example, to record the value of an interesting variable.

Second, the profiler should support the C language. The users must be able to specify execution points by indicating a point in a source file. This is mainly because the FreeBSD operating system, and other major operating systems like Linux, are written in C. Several features of the C language makes it difficult to develop a kernel profiler. For example, the macro processor makes it difficult to specify an execution point and the compiled binary includes only limited symbol information.

Third, the execution points that the users can specify for profiling must be fine grained. The possible execution points must include not only function calls but also member accesses, that is, accesses to members of structures. A number of execution points that we are interested in for performance profiling are function calls through function pointers. Modern OS kernels use function pointers for inter-module function calls since function pointers can be used for implementing a kind of polymorphism in the C language. If the read or write system call is issued, the OS kernel invokes a function pointed to by a function pointer associated with the accessed I/O device. The function pointer associated with each I/O device points to the read/write function dedicated for that device. The VFS (Virtual File System) uses the same technique for dispatching to a function appropriate to each type of file system. The network module of FreeBSD and NetBSD, which are descendants of 4.3BSD, uses this technique for deallocating a memory buffer (mbuf) in a means depending on a network device.

Finally, the prove effects due to the profiling should be minimized. If the overheads of measuring elapsed time is large, the obtained data would be obviously inaccurate. Once necessary data are obtained, the profiling code for the time measurement must be removed to avoid disturbance of the kernel behavior while the elapsed time of a different code section is being measured.

A naive approach for performance profiling of OS kernels is to manually insert profiling code into source files of the kernel, compile the source files, and reboot the kernel. However, this approach is error-prone and does not satisfy our requirements since it needs rebooting.

3. KLAS: KERNEL LEVEL ASPECT-ORIENTED SYSTEM

To fulfill all our requirements, we have developed a new dynamic aspect-oriented system called KLAS (Kernel-level Aspect-oriented System) for FreeBSD 5.2.1. AOP is the most promising approach for our requirements. KLAS receives the definition of an aspect from the users through a KLAS command running in the userland. Then it dynamically patches the running OS kernel to weave that aspect into the kernel at runtime. Since KLAS uses a modified version of gcc for augmenting the symbol information contained in the compiled binary of the OS kernel, it allows the users to pointcut member accesses at the source-code level.

3.1 Overview of the KLAS system

KLAS is a dynamic aspect-oriented system for the OS kernel of FreeBSD. The users can dynamically weave an aspect into the running kernel so that they can change the code section of which they measure the execution time. They do not have to reboot the kernel when they change a woven aspect. This feature improves the efficiency of the users' investigation since they do not have to wait until the kernel is rebooted and the behavior that they want to investigate appears again. They can start investigation as soon as they find the behavior that they are interested in.

KLAS allows the users to pick out member accesses (accesses to a member of a structure) by pointcut. As we have already mentioned, it is a crucial feature that the users can specify that an advice body is executed when a particular member of function pointer type is accessed. For example, this feature helps us investigate a performance bottleneck of network processing since we can easily measure the execution time of functions accessing the mbuf structure.

An aspect definition for KLAS is described in XML. Figure 1 shows an example of an aspect definition for KLAS. It pointcuts accesses to the member ext_free of the m_ext structure. Since the value of ext_free is a function pointer, this member access is a function call. The advice body associated with this pointcut prints the current time and the arguments to the function when that member access is performed. In KLAS, special variables $eip, $ebp and $esp are available within advice body. They represent eip, ebp and esp register.

3.2 Implementation

KLAS inserts the *hook* code into the OS kernel for executing advice body when the thread of control reaches there. The overhead due to the hook code is minimum since KLAS

```
<aspect name="log_mbuf_clean">
  <pointcut>
    <member-access name="ext_free" struct="m_ext" />
  <pointcut>
  <before-advice>
    void* resolve_arg(long eip, long ebp, int argn)
    {
      /* resolve the N-th argument of
         ext_free function. */
    }
    struct timespec ts;
    nanotime(&ts);

    printf("mbuf_clean@%d,%lld, arg:0x%x,0x%x\n",
        ts.tv_sec, ts.tv_nsec,
        resolve_argument($eip,$ebp,1),
        resolve_argument($eip,$ebp,2));
  </before-advice>
</aspect>
```

Figure 1: Aspect Definition in KLAS

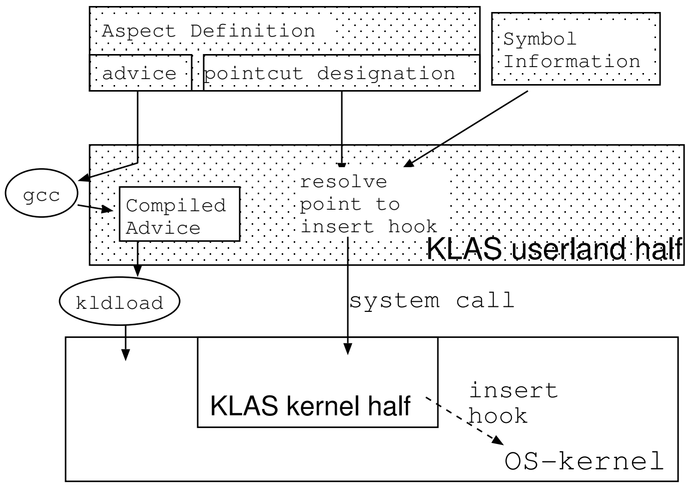


Figure 2: Implementation of KLAS

dynamically inserts the hook code only at the places corresponding to the joinpoint shadow picked out by given pointcuts. If the aspect is unwoven, the inserted hook code is also removed from the running OS kernel. Minimizing the overhead is important since the primary application of KLAS is to investigate a performance bottleneck in the OS kernel. If the overhead of using aspects is not negligible, the users may be confused by the disturbance by the prove effects and have a trouble to find a real performance bottleneck.

A unique feature of KLAS is that KLAS enables member accesses to be picked out by a pointcut. To do this, KLAS expands the symbol table contained in the compiled binary. This fine-grained pointcut helps the users to efficiently investigate a performance bottleneck in the OS kernel. The users can specify a pointcut to pick out interesting member accesses at the source-code level, and KLAS refers to the expanded symbol table so that it can insert the hook code at the machine instructions corresponding to those member accesses (Figure 2).

To use KLAS, the OS kernel must be compiled by our extended the GNU C compiler (gcc) with the -g debug option. During compilation, our compiler collects the names of structures and their members with the line numbers and the file names in which those members are accessed. The collected information is stored in an auxiliary file of the compiled kernel. Note that this information is not included in the normal symbol table of the compiled binary even if the -g option is given to the compiler. For example, the GNU C compiler discards this information after the parse tree is created; the structure names and the member names are converted from character strings to integer ID. numbers. The GNU C compiler uses not names but those ID. numbers for identifying structures and members after the parsing phase.

If KLAS is requested to dynamically weave a new aspect while the OS kernel is running, it refers the symbol information generated when the kernel was compiled. KLAS uses that information for identifying the addresses of the machine instructions corresponding to the joinpoints picked out by given pointcuts. To identify the address of a function, KLAS simply refers to the regular symbol table by invoking the nm command. To identify the address of a member access, KLAS performs the following three steps. First, KLAS refers to the auxiliary file generated by our extended compiler and obtains the file name and the line number at which that member access is executed. Then KLAS accesses the debug information, which is included in the regular symbol table. It uses the file name for identifying the name of compilation unit, which is an object file constituting the OS kernel, and it finally accesses the debug_line information (the DWARF2 format) of that compilation unit. The address of the line specified by the line number can be found in the debug_line information, which is also included in the regular symbol table. Since KLAS can obtain only the address of the first machine instruction of the line including the joinpoint, it cannot insert the hook code exactly at the instruction corresponding to that joinpoint. However, we believe that this limitation is not a serious problem for our application, which is investigating a performance bottleneck of the OS kernel. Moreover, this approach allows the users to use the same compiler that they are usually using for compiling the kernel because the information our modified GNU C Compiler generates is only a mapping between member accesses and line numbers.

KLAS uses the GNU C compiler (gcc) for compiling an advice body and the kldload command for loading the compiled advice body into the kernel land. After parsing an aspect definition written in XML, KLAS extracts an advice body and attaches the prologue and the epilogue to the advice body to make a source file of a loadable kernel module. This produced source file is compiled by gcc. The compiled binary is loaded by the kldload command. The advice body can be any code fragment if it is a valid C program in the kernel.

The loaded advice body is woven when a system call for dynamic weaving is issued. KLAS identifies the machine instruction corresponding to the joinpoint and replaces it with the breakpoint-trap instruction, which is the hook code of

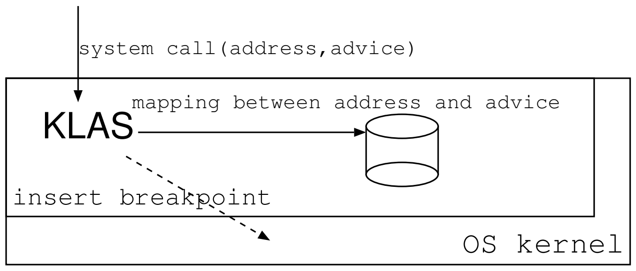

system call(address,advice)
mapping between address and advice
KLAS
insert breakpoint
OS kernel

Figure 3: KLAS in the kernel space

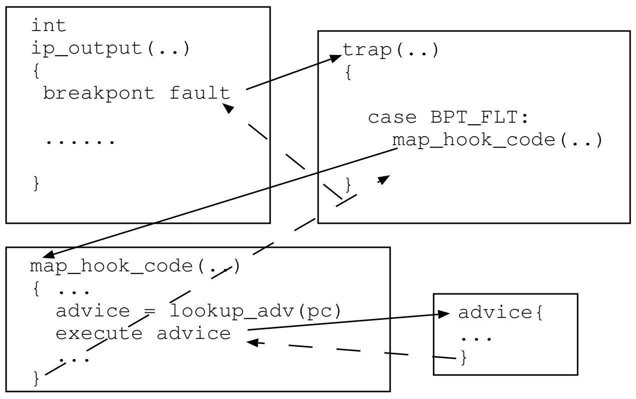


Figure 4: Execution of Advice

KLAS (Figure 3). This replacement is done while the OS kernel is running. Since the length of the breakpoint instruction of the x86 architecture is one byte, any machine instruction can be replaced with the breakpoint instruction. If the aspect is unwoven, the original machine instruction is substituted for the breakpoint-trap instruction. Note that the jmp instruction cannot be used as the hook code since the length of that instruction is three bytes. If an one-byte instruction located at the end of an basic block is replaced with the jmp instruction, the first instruction of the adjacent basic block is overwritten by the jmp instruction. This may cause system hang-up. However, according to our experiment, using the jmp instruction for the hook is about 25 times faster than using the breakpoint-trap instruction. We are planning extend KLAS to use the jmp instruction if the joinpoint is in the basic block.

When the thread of control reaches the breakpoint instruction substituted by KLAS, a breakpoint trap occurs (Figure 4). Then the trap handler executes the map_hook_code function, which we implemented. This function looks up the advice body corresponding to that breakpoint instruction, that is, the joinpoint and then executes that advice body. Finally, this function executes the original instruction replaced with the breakpoint instruction. KLAS execute this in the same way as DDB, the kernel debugger of FreeBSD, does.

4. RELATED WORK

4.1 Kernel Profilers

Ktr[8] is a kernel profiler included in BSD/OS and FreeBSD. This system is activated if a special compile option is given when the kernel is compiled. The users can manually insert logging code in the kernel source files before compilation. The logging code must be written by using CTRx macros. Since each logging code is individually activated during runtime according to the data structure called ktr_mask, the users can turn on and off the logging code by accessing ktr_mask. Unlike KLAS, Ktr does not allow the users at runtime to change the locations where log messages are printed. If the users insert logging code at a large number of locations, they can selectively activate only a few of them and change which ones are activated during runtime. However, this approach implies non-negligible overhead.

LKST[1, 2] is a kernel profiler for Linux. This system allows the users to record current time or execute any given code at a fixed set of locations in the kernel code. A problem of LKST is that the users cannot specify the locations where log messages are printed. They must select from the locations predetermined by LKST. This feature makes it difficult to investigate the behavior of the kernel in a fine-grained way.

KernInst[13] and GILK[9] can dynamically transform the binary code of the OS kernel. However, the users must specify which machine instructions are replaced with another code fragment. They cannot specify the replaced code with the source-level abstraction, such as a function call and a member access.

4.2 Aspect-oriented Solutions

AspectC++[12] is an aspect oriented system for the C++ language. Although it satisfies most of our requirements, it is a static aspect-oriented system. If the users change profiling code, that is, aspect code, the OS kernel must be recompiled and rebooted.

μDyner[11] is a dynamic aspect-oriented system for the C language, but the runtime overhead is not negligible. It inserts special *hook* code at the shadow of all the join-points at compile time. Some of the inserted hooks are activated by the pointcut description given at runtime and then they invoke an advice body when the thread of control reaches those hooks. That is, μDyner inserts the hook code at compile time at all the places in which the users may potentially want to measure the execution time. Since the number of the places in which the hook code must be inserted is usually large, the overhead due to the hook code is not negligible.

TinyC^2[7] is another dynamic aspect oriented system for the C language. Unlike μDyner, TinyC^2 can directly insert and remove the hook code in/from the compiled binary during runtime. This capability is provided by Dyninst[4], which is the backend system of TinyC^2. Since the hook code is inserted at only the places selected at runtime according to the given pointcut description, the overhead due to the hook code is minimized. However, TinyC^2 provides only a limited kind of execution points as joinpoints. For example, function calls are joinpoints but member accesses are not since the compiled binary of a C program does not include the information about which machine instruction corresponds to member accesses. The users must explicitly specify which machine instruction they want to pick out by pointcut description.

TOSKANA[3] is a dynamic aspect oriented system for OS-kernels. It runs on the NetBSD operating system. It uses the same approach as KLAS for loading advice. Since the hook is implemented with a branch instruction, the execution time of advice is faster than KLAS's. However, since it does not use a modified compiler, it can not pick out a member access as a joinpoint.

AspectC is an aspect oriented system for the C language. Research with AspectC showed that AOP is useful for implementing the cache mechanism in the kernel[6, 5] . Caching in the kernel is a concern crosscutting across a memory management module and a disk management module. Since these two modules are in different layers, implementing this concern without AOP is more difficult than other concerns cutting across multiple modules at the same layer. AspectC weaves an aspect by source-to-source translation at compile time. It does not support dynamic weaving. Also it does not provide member accesses as joinpoints.

PROSE[10] is an early dynamic AOP system for Java. It uses JVMDI (Java Virtual Machine Debugger Interface) to implement dynamic weaving of aspects. It sets breakpoints at the joinpoints specified by pointcuts. If the thread of control reaches one of those breakpoints, the JVM (Java Virtual Machine) transfers the control to the PROSE system so that PROSE will execute advice code associated with the joinpoint. This idea is the same as ours but KLAS is a dynamic AOP system for the OS kernel written in C. The cost of handling breakpoint traps in the OS kernel is relatively smaller than in the JVM.

5. CONCLUSION

Investigating a network bottleneck in the OS kernel needs a sophisticated kernel profiler that enables measuring execution time of a fine-grained code section. Since the OS kernel consists of a large number of layered modules, aspect orientation is a significant paradigm for designing such a kernel profiler. Furthermore, an aspect-oriented kernel profiler should be able to dynamically weave an aspect for avoiding kernel rebooting, which seriously decreases the efficiency of the investigation of performance bottleneck.

In this paper, we proposed an aspect oriented system named KLAS. This system provides fine-grained joinpoints, including member accesses to structures, so that the users can investigate details of the behavior of the OS kernel. KLAS extends the symbol information included in the compiled binary. KLAS collects symbol information, such as the file name and the line number of member-access expressions, at compile time and it makes the collected information available for the runtime weaver of KLAS. This is because normal C compilers produce a relatively smaller amount of symbol information than Java compilers. We are currently still implementing KLAS. The runtime weaver and the enhanced the GNU C compiler have been implemented but a language processor for compiling an aspect written in XML are still being implemented.

6. REFERENCES

[1] http://lkst.sourceforge.net/.

[2] http://oss.hitachi.co.jp/sdl/english/lkst.html.

[3] http://www.betriebssysteme.org/Aktivitaeten/Treffen/2004-Dresden/Programm/Folien/Engel/AOSTA-Slides.pdf.

[4] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[5] Y. Coady and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59. ACM Press, 2003.

[6] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98. ACM Press, 2001.

[7] G. T. Leavens and C. C. (eds.). Foal 9,003 proceedings - foundations of aspect-oriented langauges workshop at aosd 2003.

[8] G. Lehey. Improving the freebsd smp implementation. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 155–164. USENIX Association, 2001.

[9] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. GILK: A dynamic instrumentation tool for the linux kernel. In *Computer Performance Evaluation / TOOLS*, pages 220–226, 2002.

[10] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002.

[11] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. L. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119. ACM Press, 2003.

[12] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[13] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

Coupling Availability and Efficiency for Aspect Oriented Runtime Weaving Systems

Sufyan Almajali
Illinois Institute Of Technology
3300 S. Federal St.
Chicago, IL 60616, USA
almasuf@iit.edu

Tzilla Elrad
Illinois Institute Of Technology
3300 S. Federal St.
Chicago, IL 60616, USA
elrad@iit.edu

ABSTRACT

Performance and availability are two critical requirements of today’s systems. Current dynamic AOP approaches have addressed the performance issue from one specific dimension: the performance of code after the weaving process. Other performance factors may have a great impact on overall system performance. This includes performance of the weaving process itself and also system availability in single and multithread environment. In this paper, we present Dynamic Aspect C++ (DAC++), a new runtime aspect weaving system that addresses the performance and availability issues in an integrated approach. By addressing the following issues we increase system availability for runtime aspect weaving: We propose a finer-grained of atomic weaving, we introduce two new efficient aspect-weaving techniques and we support multithreading. The scope of performance impact has been extended to include a more general evaluation of the overall system performance. This includes the performance of both the woven product and the weaving/unweaving time. The synergy of these two contributions, increasing availability and overall performance consideration, led to runtime weaving systems of better throughput, response time, and accessibility to resources.

General Terms

Languages, Performance, Design

Keywords

Aspect Oriented Programming, Dynamic Weaving, Language Design.

1. INTRODUCTION

Many dynamic aspect oriented systems have been implemented. Examples of these are PROSE [13], Steamloom [3], Jboss [9], AspectWerkz [17], JAsCo [16], JAC [8], AspectS[7] and others. Some of the characteristics addressed by these systems include the technique used to introduce runtime weaving capability, the types of advices that can be used, the scope of the weaving process like per class or per instance, the ability to, partially or completely define a new aspect at runtime, and the ability to weave and unweave aspect at runtime.

None of the current AOP approaches have addressed the performance impact of runtime weaving at all different times of the program lifetime. Many factors contribute to the overall performance of runtime weaving system. The performance of systems after the weaving process is an important factor because it’s the most significant part of the overall system execution time, but other factors may degrade the overall system performance. One of these factors is the runtime weaving/unweaving process. This weaving /unweaving time can lead to performance degradations and poor system availability. Another factor is the weaving atomicity principles followed by these approaches. Not optimizing the weaving atomicity leads to extra degradation in system performance and availability. In addition, the weaving atomicity techniques followed by current systems make it difficult to support dynamic weaving for multithreaded and database transactional systems. A detailed analysis about the atomicity handling by current AOP systems is presented in the coming section.

In this paper, we show that addressing these two issues: (1) performance impact in all dimensions, and (2) weaving atomicity, can lead to a runtime weaving system that exhibits better performance and overall availability.

2. DAC++ Design Goals and Challenges

2.1 Design Goals

The main design goal for the DAC++ system is to support runtime aspect weaving system with the following characteristics: high system performance, high system availability and support of different service levels for the weaving process.

2.1.1 High System Performance

To come up with a high system performance, we need our system to provide efficient execution in the following cases: First, efficiency when the system does not require any aspects – this requires minimizing the overhead of introducing dynamic aspect capabilities so that when aspects are not raised, the execution time is as close as possible to the time of executing the code in a non AOP environment. Second, efficiency after we weave aspects at runtime - this requires runtime to be as close as possible to the runtime at a static woven environment. Third, we require an efficient weaving process and fourth an efficient unweaving process. Weaving atomicity will be discussed latter.

Current AOP approaches provide efficient execution for some of these four cases but not all. Our goal to make our system supports high performance in all of these four cases.

2.1.2 High System Availability

There are three different ways to view system availability: Throughput, Response Time and Access to Resources.

For throughput, the more jobs and transactions the system executes per interval of time, the higher the system throughput. For response time, the shorter time to respond to user requests, the faster the average response time. And finally, the smaller the service interruption time, the better the access to system resources.

Multithreading is a powerful tool for creating high performance applications. Threading is a key feature to maintain high system availability.

Supporting multithreading with the runtime weaving capability can lead to high performance applications that have the flexibility to change at runtime and adapt to different situations.

Some runtime weaving systems support weaving per thread capabilities. Examples are JBoss, AspectWerkz and Steamloom. To our knowledge, the majority of current dynamic AOP systems support coarse-grained atomic weaving. When a weaving request comes to the system, it makes sure that the entire aspect is woven for all targeted joinpoints before proceeding. For single thread applications, that would be easily applicable and there will be little or no effect on the system availability. For multithreaded applications, coarse-grained atomic weaving can cause perceptible degradation to system performance and availability especially if the weaving request is suppose to crosscut multiple threads. In case we have multiple threads running and one or more thread needs a longer time to finish execution, weaving an aspect at application, class or instance level for multiple threads cannot be done before all current running threads finish execution, and no new threads can start before the weaving process is complete. This would be the case when threads share some class and method invocations. Threads may need a long time to finish even if we have low CPU utilization because threads may contain asynchronous operation like I/O operations. Such cases can result in a system with smaller number of jobs served, longer response time and halting services for some time. This availability reduction time will last for the whole period of the weaving time. In addition, it will be always difficult to find the right time to do the weaving step. This could be a very complicated task.

In our system, we have relaxed the condition of atomicity from Coarse-grained atomic weaving to fine-grained atomic weaving. System consistency is still maintained under fine-grained atomic weaving. Using fine-grained atomic weaving simplifies the process of supporting multithreaded systems and minimizes availability degradation during the weaving process. Using fine-grained atomic weaving allows us to weave aspects at any time and at any level (Application, class, instance or thread) without the need to stop the running threads or wait until they finish their execution. Note that we still provide for system consistency (see section 3).

2.2 Design Challenges

To achieve our goals, we had to address the following design challenges.

2.2.1 Fine-Grained Atomic Aspect Weaving

First, let us show by example the difference between fine-grained and coarse-grained atomicity for the runtime weaving system.

Consider the example shown in Figure 1. Figure 1.a shows a program that has two transactions (T1 and T2) and each one of them invokes a number of methods. Assume that while the program is being executed in the middle of transaction T1, a request has been issued to weave aspect A. Assume that weaving aspect A affects methods m1 through m4 and the resulting methods after the weaving are m1A through m4A. If we are using Coarse-grained weaving, then system has to look like Figure 1.b. To keep the atomicity of both transactions, system has to wait until both transactions are finished and then weave aspect A to end up with program in Figure 1.b.

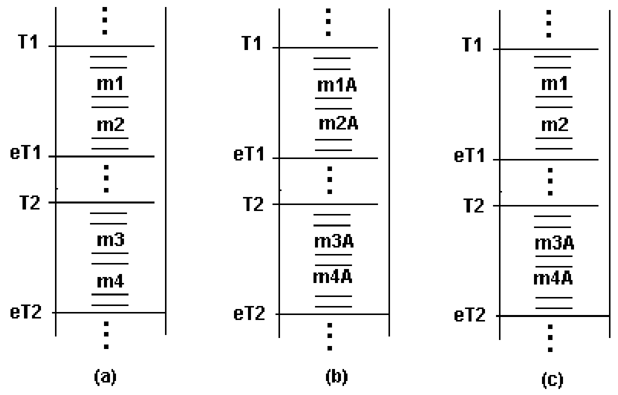


Figure 1. Coarse-Grained and Fine-Grained Atomic Weaving

With fine-grained atomicity, it would be possible to get a case like Figure 1.c where aspect A is woven for T2 but not T1 for a short period of time. This can improve weaving responsiveness and still maintain system consistency. The same thing applies for the multithreaded system in Figure 2. Figure 2.a shows a multithreaded program. Figure 2.b shows coarse-grained weaving and Figure 2.c shows a possible state that may occur sometimes during fine-grained weaving. Figure 2.d shows an invalid system state that should be avoided for single thread and multithreaded systems. Transaction T1 has aspect A partially woven. This state should be infeasible because it does not belong to either the coarse-grained weaving space or fine-grained weaving space.

Figure 3 shows a more complicated atomicity case. This case shows that atomicity needs to be verified at the method level and not only for transactions. Method m1 has in its body two invocations one for m2 and another for m3 (Figure 3.a). If we weave an aspect that targets methods m2 and m3, then the atomic weaving is shown in Figure 3.b. Figure 3.c shows an invalid case that should be avoided.

Achieving fine-grained atomicity is difficult. We need a system that is able to analyze the impact of each aspect on every method and then make this information available at runtime for weaving when needed. Also aspects interaction adds extra complexity to this needed analysis.

2.2.2 *Sustaining Both Efficiency and Availability*

To produce high system availability, we need a fast weaving approach. Also, we need to support a multithreading-aware runtime weaving system. On the other hand, to produce high performance execution time, we need a weaving approach that results in efficient woven code. Unfortunately, different weaving approaches are available and none of them can guarantee for us both fast weaving process and efficient woven code for all possible system cases.

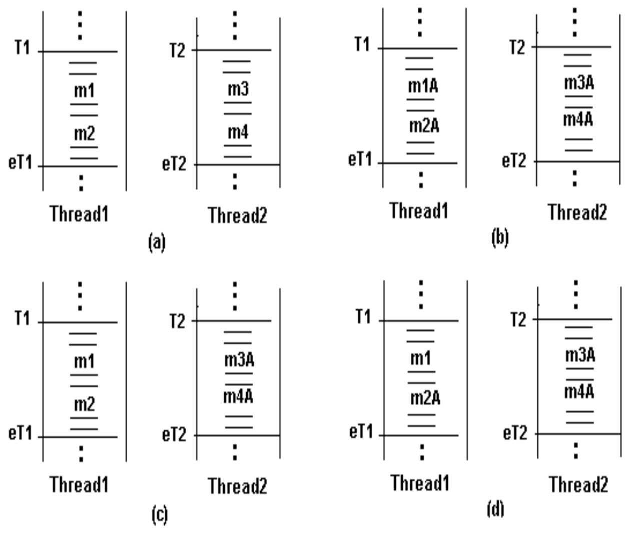


Figure 2. Coarse-Grained and Fine-Grained Atomic Weaving for Multithreading Systems

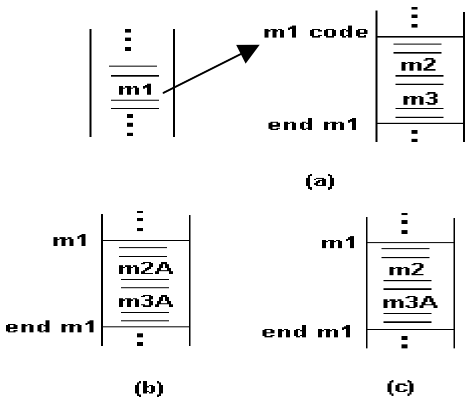


Figure 3. Atomicity at the method level

2.3 Paper Structure

The paper is structured as follows: In Section 3 we describe DAC++ architecture and the different components used and how they are integrated. In addition, Section 3 explains the implementation details of our system. The system performance is examined in Section 4. We conclude the paper with section 5.

3. DAC++ Architecture and Implementation Details

The long-term goal of DAC++ [1,2,4] compiler research is to support a complete feature set of AOP capabilities for networking services, protocols, devices and frameworks.

DAC++ compiler has been implemented to support efficient runtime weaving capabilities for the C++ language. The current version of DAC++ supports a subset of ANSI C++ syntax and extends the C++ language to support AOP capabilities. A customized version of the LCC project [6] has been used as the back end of DAC++ to give us more control over the code generation stage. Part of the back end has been modified to accommodate the needs of our DAC++ compiler.

DAC++ includes five main added elements: Atomicity Analysis for Runtime Weaving, Metadata, Two-Mode Runtime Weaving, Multithreading Support, and the Automatic Availability Analyzer.

3.1 Atomicity Analysis for Runtime Weaving

DAC++ performs Atomicity Analysis for Runtime Weaving at compilation time. Analysis determines the effect of weaving aspects on an application program. Atomicity analysis finds all weaving possibilities that can cause conflict with atomic execution. Analysis results in locating fine-grained atomic execution units. Those units could be methods or transactions. Guaranteeing atomic execution for these units eliminates the atomicity violations that can happen at runtime. Some aspect may cause atomicity conflict and others may not. We consider the aspect as a conflicting aspect only if it can cause side effect problems for method or transaction execution during weaving time. Logging, debugging and aspects of similar functionality are not going to be conflicting aspects in general.

To perform atomicity analysis, the DAC++ compiler performs four steps: control-flow analysis, inter-method data-flow analysis, method execution data-flow analysis and aspect weaving conflict analysis.

3.1.1 *Control-Flow Analysis*

Figure 4 shows how the analysis happens. In the first step, control flow analysis is used to compute the call graph of the program. Figure 4.b shows the call graph of the code given in Figure 4.a. Constructing call graph is a straightforward process for simple programming languages. In our case, there are three issues that make this process difficult, namely, separate compilation, function overload support and virtual function support.

Separate compilation can be bypassed as an issue by doing call graph construction only when an entire program is presented to a compiler at once, or it can be handled by saving, during each compilation, a representation of the part of the call graph seen during that compilation and building the graph incrementally. We have assumed the first solution although the second can be used in our system. This separate compilation problem applies in the same manner to the rest of steps.

The C++ support for function overloading allows the use of the same name to define different functions of different prototypes.

This problem is resolved using name mangling process. Finally, the problem of virtual function support is the most challenging one. This makes it impossible to create a call graph at compilation time. The reason for that is the need for runtime information to know which virtual function to call for different function call statements. As a result, we had to create an approximated call graph. For each virtual function call, we have created one arrow for each possible targeted virtual function.

3.1.2 *Data-Flow Analysis*

Next, the compiler performs data-flow analysis. This allows us to gain detailed information about all variables used by every method. C++ supports different type of variables. This includes global, member, static, local and global variables. In addition, our system allows the definition of aspect local variables and advice local variable. Our current system allows aspect to access program global variables, class static variables, and parameters of method invocation, aspect and advice local variables.

The goal of this analysis is to determine for each function call, a safe approximation of the side effects of each method invocation. We represented side effects using two functions: MOD and REF. We computed this for two types of variables supported by C++ global and static.

$\mathrm{MOD}(f,i)$ defines the variable that may be modified by executing the ith instruction of function f.

$\mathrm{REF}(f,i)$ defines the set of variables that may be referenced by executing the ith instruction of function f.

3.1.3 *Method Execution Data-Flow Analysis*

This step includes accumulating all variables that are used within the method body or its execution flow. The approximated call graph is used in this step along with information generated from previous stage. This allows us to approximate both the MOD and REF functions for each program function.

$\mathrm{MOD}(f)$ defines the set of variables that may be modified by executing function f. Also, the MOD function defines for each variable the number of modifications. $\mathrm{MOD}(f)$ takes the following form:

$$\mathrm{MOD}(f) = \{(\alpha 1, \sigma 1, \xi 1)\ ,\ (\alpha 2, \sigma 2, \xi 2)\ ,\ \ldots\ldots\ ,\ (\alpha n, \sigma n, \xi n)\}$$

where αi is the name of ith variable in the list, σi is the type of the variable (global or static), and ξi is the number of times variable αi can be modified inside this method or method execution flow. For each variable i, its name αi is fully qualified using the function or class mangled name.

Regarding the REF function, $\mathrm{REF}(f)$ defines the set of variables that may be referenced by executing function f and the number of references for each variable. $\mathrm{REF}(f)$ can be represented by the following form:

$$\mathrm{REF}(f) = \{(\alpha 1, \beta 1, \zeta 1)\ ,\ (\alpha 2, \beta 2, \zeta 2)\ ,\ \ldots\ldots\ ,\ (\alpha n, \beta n, \zeta n)\}$$

where αi is the name of ith variable in the list, βi is the type of the variable (global or static), and ζi is the number of times variable αi may be referenced inside this method or method execution flow.

MOD and REF functions are easy to compute for functions that do not contain other function calls. For nested function calls, the approximated call graph is used. Also, MOD and REF functions will be evaluated for the advice function for each aspect.

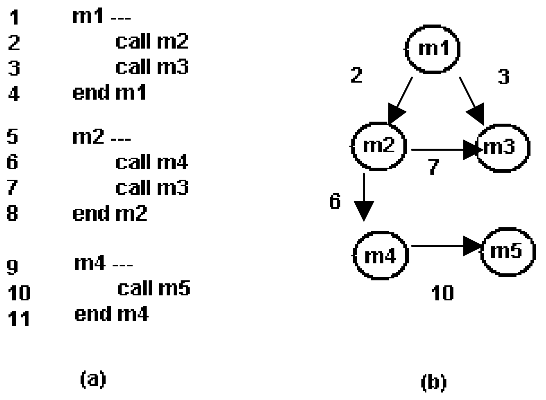
3.1.4 *Aspect Weaving Conflict Analysis*

The purpose of the final step is to study the side effect impact of each aspect in every method. Assume we want to weave aspect A with method M, then we have two cases:

Case 1: Aspect A does not cause side effects to the execution of the application.

Case 2: Aspect A does cause side effects to the execution of the application.

In case 1, we do not need to force atomic weaving because it is not required. In case 2, we need to force atomic weaving to avoid partial weaving of aspect A. To satisfy atomic weaving during the weaving of aspect A, the system should be in one of two states: 1) either run the effected application part without any aspect A side effects or 2) run it with same aspect A side effects.



```
1     m1 ---
2         call m2
3         call m3
4     end m1

5     m2 ---
6         call m4
7         call m3
8     end m2

9     m4 ---
10        call m5
11    end m4
```

Figure 4. Control-flow Analysis: Call-graph

The following algorithm is used to determine the aspects require atomic weaving and the program location for that.

Assume we have aspect "A" that crosscuts application methods represented by the set $S_A = \{M_1, M_2, M_3, \ldots M_n\}$.

(1) Assign Aspect "A" a unique version number V_A.

(2) For each method $M_i \in S_A$, use the approximated call-graph and find all execution paths that lead to M_i. Execution paths can be cached for later use and avoid redundant computation. The result of the second step will be a set of execution paths, each one of the following form: $(K_1 K_2 \ldots\ldots K_h M_i)$.

(3) Evaluate the following step for each execution path $(K_1 K_2 \ldots\ldots K_h M_i)$:

```
conflict = false;   count =1;
    while (count <=h)
{
$S_\alpha = REF(K_{count}) \cap MOD(A)$
(i) if $S_\alpha = \varnothing$ then
exit; // break while loop , no conflict

(ii) if ( $\sum \forall \beta_i$ for $\alpha_i \in S_\alpha$
= $\sum \forall \beta_i$ for $\alpha_i$ for all $K_{count}$ instructions before invoking $K_{count+1}$)
exit; // break while loop , no conflict  ($S_\alpha \neq \varnothing$ implicitly)

(iii) if ( $\sum \forall \beta_i$ for $\alpha_i \in S_\alpha$
= $\sum \forall \beta_i$ for $\alpha_i$ for all $K_{count}$ instructions after invoking $K_{count+1}$ )
exit; // break while loop , no conflict
(iv) if ( $\sum \forall \beta_i$ for $\alpha_i \in S_\alpha = \sum \forall \beta_i$ for $\alpha_i$ for REF($K_{count+1}$) )
then
     count=count+1
     else
    {  conflict=  true;
         record Aspect version number "$V_A$" as a conflict with current execution path. And force atomicity at method $K_{count}$
           exit; //  break while loop , conflict case}
  } // end of while statement
```

S_α defined in step 3 represents the set of variables used by method K_i and may get modified by aspect A. Condition (ii) of step 3 addresses the case when all of K_i instructions that reference the variable(s) modified by the aspect are before invoking method K_{i+1}. This leads to no conflict since the aspect A will be woven after all references happen. Condition (iii) addresses the opposite situation. Conflict will happen when some references are made before invoking method K_{i+1} and the others are made after invoking method K_{i+1}.

One important issue here is aspect interaction. When we weave an aspect A and follow it by weaving an aspect B, this needs more complicated analysis. It is possible that weaving aspect A or B alone does not cause conflict with method M , but weaving both aspects causes conflict with the method.

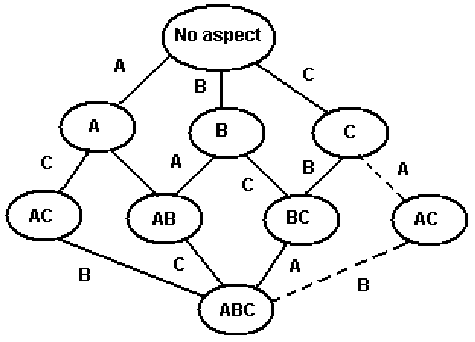


Figure 5. Aspect Interaction Analysis for Aspect A, B and C

To resolve this issue, DAC++ generates approximated call-graphs for all weaving possibilities of the aspect that have side effects (write statement(s)). Figure 5 shows aspect interaction for three aspects A, B and C. Each node represents a weaving case. Each node is given a version number to represent aspects being woven for the case.

For each node in the graph, DAC++ performs conflict analysis and records the conflicting version number with each method. Notice that this could be a space and time consuming process in case we have a large number of aspects, but it is needed only at compilation time (done one time) and the result graphs are not needed at runtime.

3.2 DAC++ Metadata

DAC++ metadata includes all information needed to support atomic runtime weaving/unweaving. It is the aspect runtime representation. DAC++ generates the initial metadata during program compilation and loading and linking time. Metadata information changes during program runtime to reflect the aspect weaving and unweaving status. Metadata includes three main categories: classes and methods metadata, aspect metadata and threading metadata.

Part of the metadata is method relocation entries. They are compiler-generated entries and used to track all locations that access the method. Many compilers generate relocation entries information [11]. This information is usually used by linkers to link multiple program modules. They are usually used at linking time, but we filter the unneeded relocation entries and load the rest in our system and utilize them at runtime.

3.3 Two-Mode Runtime Weaving

One of the main unique things used to support efficiency in our system is the inclusion of two approaches for runtime weaving. The two weaving approaches represent two different system modes for weaving aspects at runtime. DAC++ uses these two modes alternatively to meet the different system needs at different times for different weaving cases. Weaving modes include the following: Wrapping weaving mode and Splicing weaving mode.

Both modes share the first steps of the weaving process and differ in the rest of the steps. One exceptional characteristic of DAC++ weaving modes is their minimum service suspension time and the promptness of the weaving process without affecting program semantics or performance.

3.3.1 Wrapping Weaving Approach

Figure 6 shows how the wrapping mode works. When the AOP thread receives a weaving request, aspect metadata will be used to determine the targeted methods of this weaving process. For each one of these methods, DAC++ instantiates and customizes mini stubs of code that will be used for temporary fine-grained atomicity check and method call redirection. In addition, DAC++ creates a wrapper method that does not have any code but method invocations and two stack instructions. The wrapper method structure is formulated to reflect the type of advice: before or after. This whole process happens without the need to stop any program part to continue its execution in parallel. Also, DAC++ can control the percentage of CPU cycles given to the AOP weaving thread so we have minimum impact on system availability. After having the stub and the wrapper methods created simultaneously with program execution, the affected thread(s) will be suspended for very short period of time to do runtime relocation for the methods. This last step is actually runtime relinking and it is the only stage where we have to suspend the execution. Suspension can be done one thread at a time to have minimum impact on overall system availability. After this minimal service suspension time, thread(s) resume their execution from the state they were suspended at without losing any state information.

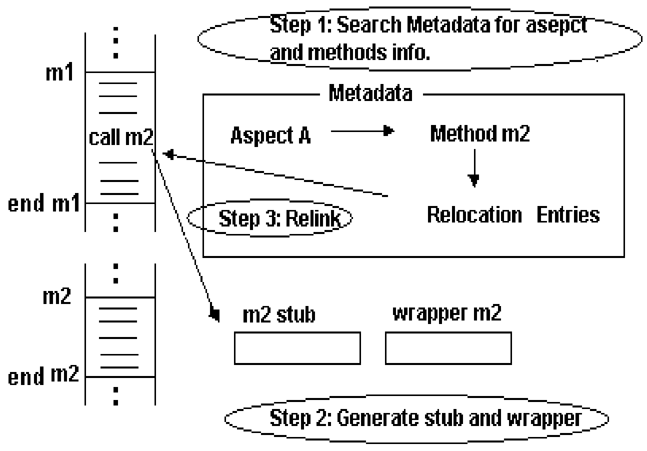


Figure 6. Wrapper Weaving Approach

The runtime relinking step allows method joinpoints to point to the stub method or wrapper method depending on atomicity conflict status. The stub takes care of atomicity checking. For each aspect weaving step, there is one Boolean variable safe_to_weave (default is false). Also, there are three types of stubs: first-step-weaving stub, atomicity-check-and-relink stub and atomicity-disable stub.

In case of weaving a non-conflicting aspect (an aspect that does not cause conflict in any case), the relocation entries are mapped to point to the wrapper method directly and stubs are not be used at all.

In case of weaving a conflicting aspect (aspect may cause atomicity conflict), the first-step-weaving stub, the atomicity-check-and-relink stub and the atomicity-disable stub are used as follows: The relocation entries are mapped to point to the first-step-weaving stub directly.

The first-step-weaving stub does the following tasks:

1) Check the stack to evaluate the current calling sequence and discover which methods are activated (discover current execution path).
2) If current aspect weaving version V_A conflicts with the current execution sequence, then set safe_to_weave to false otherwise, set safe_to_weave to true. Method causes conflict is predetermined at compilation time.
3) For that method stack frame, save its return address, and modify it to point at the atomicity-disable stub.
4) If safe_ to_weave then
    a. Relink caller to new wrapped_method
    b. Execute_wrapped_method
5) Else
    a. Execute_original_method
6) Replace first-step-weaving stub by atomicity-check-and-relink stub

Note that the first-step-weaving stub is executed one time only for each weaving request.

The atomicity-check-and-relink stub has only steps 4 and 5 of the first-step-weaving stub.

The atomicity-disable stub will be called only when system reaches the end of the method that has a conflict. The atomicity-disable stub will check again for atomicity conflict with the current stack and either reset the return address of the conflicting method as happened in the first step or, set the safe_to_weave variable to true. In case safe_to_weave is set to true, the next method calls will access the wrapper method directly and the stubs will not be accessed.

The relink step used by the DAC++ stubs allows next method calls to be direct method calls instead of redirecting each call through the stub. Also, to accomplish relinking, we do not need to have any information saved, the caller calling instruction address can be easily computed from the current stack frame of the current executing method.

The wrapper method contains the following assembly code:

1 **Wrapper_method**:

2 **Call** advice_function ; used if the advice is of type *before*

3 **Pop** some_memory_loc; this is used to pop the return

4 ; address of the wrapper method

5 **Call** original method ;

```
6    push some_memory_loc; replaceing the return address into its
7                                  ; correct location
8     Call advice_function ;  used if the advice is of type after
9    Leave
10  Ret
```

Either line 2 or 8 will be used but not both depending on the advice type. The reason for using the push and pop operations is to avoid pushing the method parameters again.

The wrapping weaving approach has the advantage of constant weaving time per method. The overall time complexity will be linear in the number of targeted methods affected by the weaving process. On the other hand, this approach adds little overhead during method execution because the wrapper method adds one indirect call step in addition to the two additional instructions, pop and push.

3.3.2 Splicing Weaving Approach

To reduce overhead completely, we have included another weaving approach that avoids indirect methods calls. We call this approach: splicing weaving approach. In this method, we follow the same start in the same manner as with the wrapping weaving approach by including a stub to check for atomicity conflicts. But, instead of using a wrapper method, DAC++ generates a new version of the method that contains the method code after the weaving process. Splicing the method code with the advice code on the fly does the process. Methods splicing does not need recompilation or method/class reloading. The original method code is already in memory. DAC++ reuses a copy of the method code and makes a small modification to accomplish the splicing with the aspect advice code. Splicing is done as the following:

1) Initially, the method object for X86 machines has the following format:

```
method_label:
        enter  X,0
   ; method code
          leave
          ret
```

Enter instruction is used to perform the method prologue by initializing needed registers and reserving Local_Bytes in the stack frame for local variables. Leave is used to perform the program epilogue.

2) The advice code has similar code:

```
advice_label:
        enter  Y,0
   ; advice code
          leave
          ret
```

3) The spliced code looks like the following:

```
splicied_method:
        enter  X+Y,0
   ; relocated advice code if before advice
   ; method code
   ; relocated advice code if after advice

          leave
          ret
```

A challenge issue for machine code splicing is its dependency on the memory location. Copying a method from one memory location to another is simply not going to work. To solve this problem, DAC++ offers two solutions. The first solution is to customize the back end of our compiler to generate method code that is location independent. Position Independent Code (PIC) is a technique used by some operating systems to load code without having to do load-time relocation and to share memory pages of code among processes even though they do not all have the same address space allocated [11]. PIC code does not help our situation because we need to relocate method code only and not the entire program. Consequently, we had to generate a kind of position independent method code (PIMC). Generating PIMC code is much easier as a process than PIC code. In PIMC code, all jump and branch instructions are made relative. But unlike PIC, we do not need to take care of changing instructions that have direct data access.

Although PIMC allows us to implement code splicing easily, PIMC has little overhead due to the relative referencing that it uses. For each jump and branch instruction, an extra memory reference is needed in order to access the PC register that saves the current execution address.

DAC++ offers programmers the option to generate PIMC code with version 2. The PIMC code of version 2 does not have relative address access instructions. Instead, it uses direct address access instructions. To make it a PIMC code, DAC++ attaches a list of memory addresses that need to be shifted during method copying and splicing time. This eliminates overhead completely but requires more space.

The same idea is applied to the advice code. The access to local variables need to be relocated by X amount because the frame pointer will be pointing at the original method local variables.

User can pick the optimization option based on the application needs and code size.

3.4 Multi-Threading Support

DAC++ supports runtime weaving for multi-threaded applications. Support includes POSIX threads, a well-known threading standard in the C++ and C languages. DAC++ allows weaving per thread. During program compilation, DAC++ identifies each thread type and performs the following steps: First, it uses the result of the call-graph generated during atomicity analysis to find all methods used by each thread and thread flow. After this, it compares all applications threads and sees the methods shared among the different threads. For each one of these methods, one copy or more of the method source code is created and method renaming is applied. The original method name is saved in the metadata part and renaming happens only for the mangled name. As a result, each thread will end up with its own code space. This allowed us to implement aspect weaving per thread without paying performance penalty. Figure 7 shows the way in which threads are handled. It is important to notice that the code space conflict has to be resolved for the inner method calls too. In case of a thread calling method m1, and method m2 is called inside m1, then DAC++ treats m2 in the same way it treats m1 with respect to threading handling. This repeats for all user-defined methods but not built-in functions like the mathematical sin function. We did not include support for per thread weaving with built in functions although it can be added to our system. It is interesting to know that cflow and weaving per instance are both supported using the same technique used for threads.

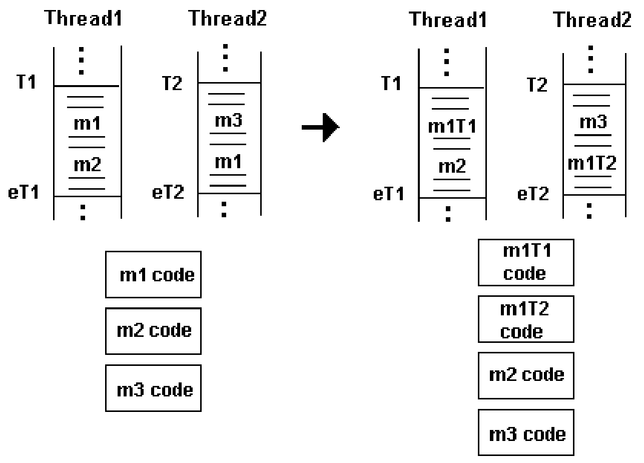


Figure 7. Threading Preprocessing

4. DAC++ Performance

We have measured the performance of DAC++ using different measurements. All evaluations were done using 3.0 GHz Pentium IV on Linux machine with I GB memory. First, we measured the overhead added by DAC++ when no aspect is woven. Table 1 shows the result of this evaluation. PIMC version 2 overhead is very minimum. The little overhead we have is due to the lack of support of all optimization levels by standard C++ compilers as we described in previous section. PIMC version 1 code has more overhead specially when it comes to loops and jumps instructors. This is expected because of the relative referencing used by this version of code generation.

Table 1. DAC++ Overhead When no Aspect is woven

Benchmark	DAC++ PIMC v1	DAC++ PIMC v2
Method call	10.02%	1.7%
Fibonacci	11.02%	2.03%
Nested Loop	15.02%	3.2%

In the second performance measurement, we evaluated the performance overhead of the two weaving approaches supported by DAC++ and compared them to statically woven aspect in AspectC++ system [15]. Table 2 shows the performance overhead after dynamically weaving aspect at run time.

Table 2. DAC++ Overhead of Weaving Aspect at Runtime

Benchmark	WWM / PIMC v1	WWM / PIMC v2	SWM / PIMC v1	SWM / PIMC v2
Logging Aspect	7.1%	2.11%	6.02%	1%
Fibonacci Aspect	9.4%	3.53%	8.02%	2.03%

In general, Splicing Weaving mode (SWM) with PIMC version 2 has the lowest performance impact. Notice that Wrapping Weaving mode (WWM) has more performance overhead and the reason is the indirection of method call and the additional parameter passing and handling necessary to accomplish the wrapping.

5. Conclusion and Future Work

DAC++ supports runtime weaving with new capabilities that are not offered by current approaches. It 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. It offers a minimum service suspension time with very efficient woven code. With its runtime relinking and fine-grained weaving capabilities, DAC++ makes the promptness of the weaving process faster with minimum program suspension.

The current DAC++ implementation has a set of limitations that need to be elaborated in future. First, We are planning to extend DAC++ capabilities in future to support full C++ syntax. Also, redefining the pointcut designator for an aspect at runtime is problematic for DAC++. We prevented it because it needs extra space at runtime and it slows down the weaving process due to the need for atomicity analysis.

Another issue that we need to address is the compiler optimization techniques that fit with DAC++. Aggressive optimization levels supported by current C++ compilers have note been addressed yet. The optimization level of spliced code generated on the fly is not high compared to the code generated during compilation. Run time code optimization is needed to achieve better performance with woven code.

One of the biggest challenges to runtime weaving in C++ is to support inter-type aspect runtime weaving. We needs to modify the C++ object model at runtime. This needs a major change in compiler technique.

Finally the effect of the program execution on aspect semantics using fine-grained atomic weaving needs more elaboration.

6. ACKNOWLEDGMENTS

This work is supported in part by CISE NSF grant No. –0137743.

7. REFERENCES

[1] Almajali, S. and Elrad, T. A Dynamic Aspect Oriented C++ using MOP with Minimal Hook Weaving Approach*. In Dynamic Aspect Workshop*, Lancaster, England. March 2004.

[2] Almajali, S. and Elrad, T. Dynamic Aspect Oriented C++ for Upgrading without Restarting. *In proceeding of Conference on Advances in Internet Technologies and Applications with Special Emphasis on E-Education, E-Enterprise, E-Manufacturing, E-Mobility, and Related Issues*, Purdue, USA, July 8-11, 2004.

[3] Bockisch, C., Haupt, M., Mezini, M. and Ostermann, K. Virtual Machine Support for Dynamic Join Points. In AOSD 2004 Proceeding. ACM Press, 2004

[4] DAC++ Home Page. http://www.iit.edu/~almasuf/dacpp.html

[5] Douence, R. and Sudholt, M. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[6] Fraser, C. and Hanson, D. A Retargetable C Compiler: Design and Implementation. . Addison-Wesley, 1995.

[7] Hirschfeld, R. Aspect-Oriented Programming with AspectS. http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/Docs/AspectS_NODe02_Erfurt2_rev.pdf

[8] JAC Home Page. http://jac.aopsys.com

[9] JBoss AOP Home P. http://www.jboss.org

[10] Levine, J. Linkers and Loaders. Morgan-Kauffman, October 1999.

[11] Lippman, S. Inside the C++ Object Model. Addison-Wesley, 1996.

[12] Popovici, A., Gross, T. and Alonso, G. Dynamic Weaving for Aspect Oriented Programming. In AOSD 2002 Proceedings. ACM press, 2002.

[13] Popovici, A., Gross, T. and Alonso, G. Just-in-Time Aspects. In AOSD 2003 Proceeding. ACM Press, 2003

[14] Schult, W. and Polze, A. Dynamic Aspect Weaving with .NET. http://www.dcl.hpi.uni-potsdam.de/dcl/papers/GI2002.ps

[15] Spinczyk, O. ,Gal, A. and Schröder-Preikschat, W. AspectC++: An aspect-oriented extension to the C++ programming language. *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of Conferences in Research and Practice in Information Technology. ACS, 2002.

[16] Vaderperren, W. and Suvee, D. Optimizing JAsCo dynamic AOP through HotSawp and Jutta. *In Dynamic Aspect Workshop*, Lancaster, England. March 2004.

[17] Vasseur, A. Dynamic AOP and Runtime Weaving for Java-How does AspectWerkz address it*? In Dynamic Aspect Workshop*, Lancaster, England. March 2004.

SONAR: System Optimization and Navigation with Aspects at Runtime

Chunjian Robin Liu
University of Victoria
cliu@cs.uvic.ca

Celina Gibbs
University of Victoria
celinag@cs.uvic.ca

Yvonne Coady
University of Victoria
ycoady@cs.uvic.ca

ABSTRACT

Traditional system optimization and navigation techniques, based on static system structure and static instrumentation, are not suitable for understanding and handling today's complex, distributed and dynamic systems at runtime. This paper introduces an approach we call *SONAR* (System Optimization and Navigation with Aspects at Runtime). Through a combination of Extensible Markup Language (XML), dynamic Aspect-Oriented Programming (AOP) and Java Management Extensions (JMX), SONAR provides a fluid and unified framework for runtime system optimization and navigation. In particular, dynamic aspects are used to integrate scattered code level artifacts across predefined abstraction boundaries – application, framework/middleware/virtual machine, operating system – together as a meaningful entity with respect to users' interests. Such aspects can be defined, enabled and disabled by a stakeholder at runtime. A preliminary evaluation of SONAR shows only minimum impact on performance and a relatively small memory requirement for large systems.

1. INTRODUCTION

Today's systems are increasingly challenging to holistically analyze due to both their size and complexity. Though layering, componentization, and virtualization can be leveraged to provide meaningful abstractions at various levels, these abstractions also make system-wide navigation and analysis, which must cross these predefined boundaries, more difficult than in simple flat systems. For example, to perform a root-cause fault-analysis, exceptions and states at various levels of the software stack need to be collected and analyzed collectively. This is difficult because application level exceptions are usually handled by the middleware and hidden from both the lower layers and the users. Likewise, certain exceptions at lower levels can be hidden or transformed to a different representation for higher levels to digest [2,3]. To be able to understand the root-cause of a problem, or some other systematic behaviour, information from entities across layers and/or distributed at different locations needs to be easily collected and correlated.

Looking at this problem from another angle, today's complex system architectures are designed and documented with multiple views (structural - class, deployment; behavioral - use case, sequence, collaboration; control flow, data flow) for multiple stakeholders. At an implementation stage, these design views are mapped into code level artifacts (module/component, file, package, class, method). For structural views, the mapping is usually direct and explicit, since current object-oriented programming methodology provides a hierarchical/structural decomposition and code is developed based on such decomposition. Other views, such as behaviour views, are typically difficult to map or realize from hierarchical/structural decomposed code level artifacts. Furthermore, users with different roles, i.e. designer, developer, administrator and maintainer, might have different interests and therefore require different views of a single system at runtime. These views may be directly derived from design views, or may represent new perspectives based on more specific stakeholder interests.

In addition to these increasing demands, increasing heterogeneity of large systems, dynamic features of programming languages, frameworks/middleware/virtual machines and operating systems, and other advanced techniques (such as configuration, adaptation and autonomic computing) make the overall system more dynamic and as a result, static instrumentation techniques are not suitable for navigation and analysis such systems. For example, by using Java reflection, a level of indirection is added and therefore a program can achieve higher degree of flexibility. However, it also makes realizing which object is actually invoked more difficult than a static call. For systems such as pervasive systems, which feature a high degree of dynamism in terms of device/service states and availability, static based techniques are equally unsuitable for analyzing runtime behavior.

In order to understand system-wide runtime behavior, certain entities in the system (or even from the outside environment) might need to be correlated together and represented as a single entity – coarser/finer grained or even crosscutting – for navigation and other tasks. Moreover, such views should be able to be reconstructed when users' interests change. Runtime analysis of a problem has been shown to typically be an iterative process, with an analyst's focus constantly changing during the process of analyzing [4]. Thus, views need to be easily defined and constructed dynamically across the predefined boundaries of class, component, subsystem, layer and so on. Ideally, views should be able to be easily removed once users no longer need them, and further incur little to no performance penalty. Simply put, systems need to be abstracted and viewed differently and dynamically.

Based on this premise, SONAR was motivated by this need for a fluid mechanism to dynamically integrate all scattered artifacts (entities and data) cross predefined vertical and/or horizontal abstraction boundaries together to represent a meaningful entity with respect to a user's interest. As a result, SONAR was designed with three key goals in mind:

1. **Language/framework agnostic definition:** As there are now an increasing number of AOP frameworks for different programming languages. SONAR can be configured to leverage this variety in order to diagnose more systems.

2. **Dynamic instrumentation:** Instrumentation code should be able to be inserted dynamically. Furthermore, such code should be able to be removed with zero impact.

3. **Simple visualization and management:** Collected data should be easily visualized and managed. Furthermore, management should also be standard-compliant.

The rest of this paper is organized as follows: Section 2 covers some background information. Section 3 explains SONAR's architecture and describes relevant implementation details showing how some of the most challenging goals above are accomplished. In Section 4, we evaluate our implementation based on performance, memory footprint, and other factors. Finally, we discuss related work and future research directions in Section 5 and conclude in Section 6.

2. Background

This section briefly introduces the three key technologies used by SONAR – XML, AOP, and JMX – and the specific ways in which SONAR uses each.

2.1 Why XML?

Extensible Markup Language (XML) was originally designed to improve the functionality of the Web by providing more flexible and adaptable information identification[1]. Given that XML is not a fixed format like HTML (a single, predefined markup language), it can be generally used to customize markup languages.

SONAR uses XML's ability to encapsulate information in order to pass it between different computing systems which would otherwise be unable to communicate.

2.2 Why AOP?

Aspect-oriented programming (AOP) modularizes crosscutting concerns – concerns that are present in more than one module, and cannot be better modularized through traditional means [7]. Looking at an aspect, a developer can see both the internal structure of a crosscutting concern, and its interaction with the rest of the program during execution. Three key elements of AOP are *joinpoints, pointcuts* and *advice.* Joinpoints are well-defined points in the program flow such as method call/execution, and field accesses, pointcuts select certain execution points in the program flow, and advice provides code to run at these selected points. Dynamic AOP allows aspects to be introduced/removed to/from a system at runtime.

SONAR leverages dynamic AOP's ability to structure system-wide crosscutting concerns for dynamic analysis, and introduce/remove them at runtime.

2.3 Why JMX?

Originally known as Sun's *JMAPI*, Java Management Extensions (JMX) [5] is gaining momentum as an underlying architecture for J2EE servers. It defines the architecture, design patterns, interfaces, and services for application and network management and monitoring. Managed beans (MBeans) act as wrappers, providing localized management for applications, components, or resources in a distributed setting. MBean servers are a registry for MBeans, exposing interfaces for local/remote management. An MBean server is lightweight, and parts of a server infrastructure are implemented as MBeans.

SONAR leverages JMX's support for highly modular and customizable server architectures, and standard support for management features.

3. SONAR Design and Implementation

The subsections that follow provide an overview of SONAR's current design and implementation. Further features and enhancements are discussed in Section 6.

3.1 SONAR Architecture

In order to address the goals described in Section 1, the overall architecture of SONAR is organized as shown in Figure 1. Aspects are generated from XML-based definition files, deployed to instrumented applications, application framework/middleware, and virtual machine. They are then managed through JMX compatible tools through domain independent API such as those related to deployment/undeployment of aspects and domain specific API such as system navigation. SONAR is thus designed and implemented in three main parts: AOP integration, XML transformation/code generation and JMX management. Each corresponds to one goal in Section 1, and is further explained in detail in the following subsections.

[1] http://www.ucc.ie/xml/#acro

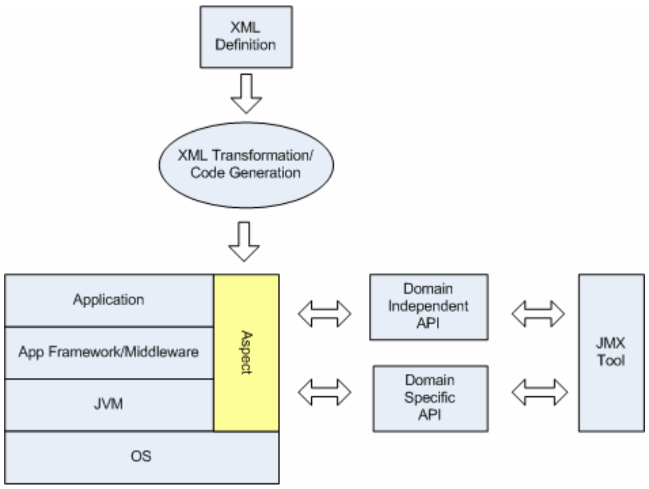


Figure 1. SONAR overall architecture.

3.2 AOP Integration

To achieve dynamic instrumentation, we chose dynamic AOP since it provides a language level (code centric) support for augmenting existing systems for various purposes. AOP's rich joinpoint model provides a solid foundation for implementing instrumentation. The joinpoint model covers almost all execution points in a system written in certain languages. These points include method invocation/execution, field access and so on. This enables fine-grained instrumentation – almost all significant events in the source code can potentially be instrumented.

Dynamic AOP further provides a powerful mechanism for runtime aspect manipulation such as runtime deployment/undeployment. In other words, advice can be dynamically woven into targets and dynamically removed from targets. Our implementation is based on a Dynamic AOP implementation, AspectWerkz[1].

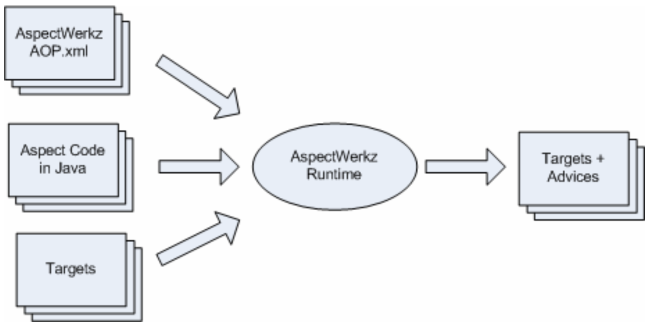
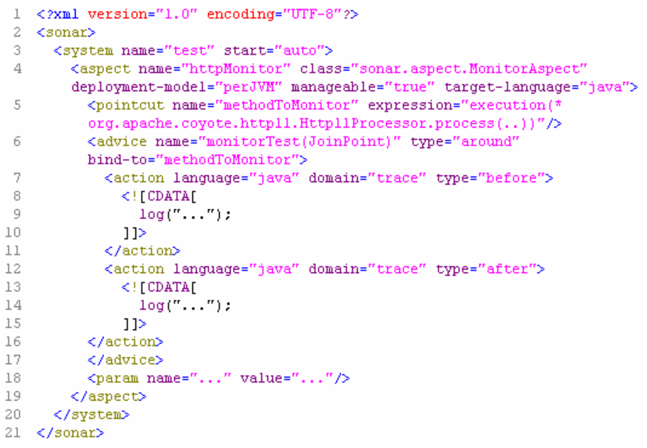


Figure 2. Illustrates how configuration file, aspects and targets are used in AspectWerkz.

3.3 XML Transformation/Code generation

3.3.1 XML Definition

Aspects are defined in AOP framework independent XML files. Therefore, they can be implemented using different AOP frameworks or even in different programming languages such as Java, C++.



```
<?xml version="1.0" encoding="UTF-8"?>
<sonar>
  <system name="test" start="auto">
    <aspect name="httpMonitor" class="sonar.aspect.MonitorAspect"
    deployment-model="perJVM" manageable="true" target-language="java">
      <pointcut name="methodToMonitor" expression="execution(*
      org.apache.coyote.http11.Http11Processor.process(..))"/>
      <advice name="monitorTest(JoinPoint)" type="around"
      bind-to="methodToMonitor">
        <action language="java" domain="trace" type="before">
          <![CDATA[
            log("...");
          ]]>
        </action>
        <action language="java" domain="trace" type="after">
          <![CDATA[
            log("...");
          ]]>
      </action>
      </advice>
      <param name="..." value="..."/>
    </aspect>
  </system>
</sonar>
```

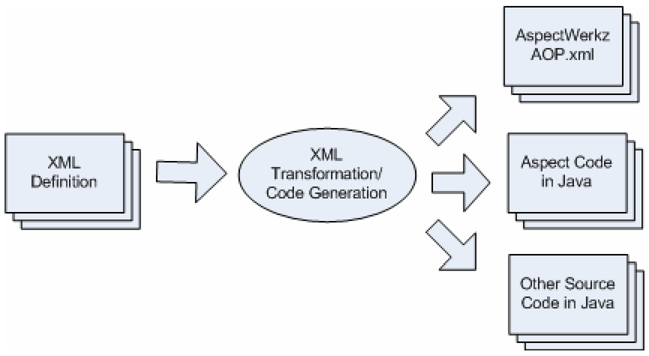
Figure 3. Sample XML definition file.

The core content of an aspect includes: variable/method declaration, pointcut expression, advice with actions, parameter definition. Variable/method declarations and actions are discussed in details in the next 2 sections. The current schema is mainly based on AspectWerkz's aspect definition schema, which is very similar to many other existing AOP frameworks, with additions required for transformation and code generation. Aspects can be declared to be started automatically or manually. Automatically started aspects are enabled when the target systems are loaded, while manually started aspects have to be explicitly manually enabled at the runtime.

3.3.2 Domain Specific API or Language

The target domain in SONAR prototype implementation is system optimization and navigation. Variable/method declarations and actions contain code targeting this specific domain. The reason for using a domain specific API or language is to separate aspect code from domain specific implementation. For example, the log method used in the Figure 3 is defined outside of the aspect code. It can be implemented as printing to screen, writing to a log or sending to some management console. As a result, the implementation choice of such method can be made independently from aspect code and therefore, can be adjusted based on the target system. Other reasons are to ease the development and limit the impact of aspects. In the current implementation, declarations and actions are written only in Java.

3.3.3 XSLT Transformation & Code Generation



XML Definition
XML Transformation/ Code Generation
AspectWerkz AOP.xml
Aspect Code in Java
Other Source Code in Java

Figure 4. XML definition files are transformed into other XML files and source code in target language using XSLT and domain specific compiler/code generator.

As shown in Figure 4, XSL Transformation (XSLT) is used to transform XML definition files into other XML files, such as the aspect definition file for AspectWerkz (Figure 5), aspect code (Figure 6) and other necessary source code such as interfaces and helper classes for management purposes. If variable/method declarations and actions in the aspect definition are written in domain specific languages, domain specific compiler might be used to compile the code into the language used in the target system.

```
<?xml version="1.0" encoding="UTF-8"?>

  <!DOCTYPE aspectwerkz PUBLIC "-//AspectWerkz//DTD//EN"
  "http://aspectwerkz.codehaus.org/dtd/aspectwerkz2.dtd">

  <aspectwerkz>
    <system id="test">
      <aspect name="httpMonitor" class="sonar.aspect.MonitorAspect"
      deployment-model="perJVM">
        <pointcut name="methodToMonitor" expression="execution(*
        org.apache.coyote.http11.Http11Processor.process(..))"/>
        <advice name="monitorTest(JoinPoint)" type="around"
        bind-to="methodToMonitor"/>
        <param name="..." value="..."></param>
    </aspect>
  </system>
</aspectwerkz>
```

Figure 5. AspectWerkz's aop.xml generated by transforming the definition file in Figure 3.

```
package sonar.aspect;

import org.codehaus.aspectwerkz.*;
import org.codehaus.aspectwerkz.definition.*;
import org.codehaus.aspectwerkz.joinpoint.*;
import org.codehaus.aspectwerkz.transform.inlining.deployer.*;

import sonar.util.*;

import java.lang.management.*;
import javax.management.*;
import javax.management.openmbean.*;

public class MonitorAspect implements MonitorAspectMBean {
  private final AspectContext aspectContext;

  public MonitorAspect(final AspectContext aspectContext) {
    this.aspectContext = aspectContext;
  }


  public Object monitorTest(final JoinPoint joinPoint) throws Throwable {
    log("...");

    final Object result = joinPoint.proceed();

    log("...");

    return result;
  }
}
```

Figure 6. Java source code containing AspectWerkz specific code generated from the definition file in Figure 3.

3.4 JMX Management

JMX is used as a means to manage the aspects. This includes retrieving data from aspects, invoking operations and receiving event notification. Through JMX, the aspects can be managed by JMX-compatible tools remotely and/or locally. The tool we used is called JConsole which is a JMX-compliant graphical tool for monitoring and management and is built in Sun's JDK distribution. Figure 7 and Figure 8 show how JConsole can be used to manage aspects, more specifically MonitorAspect as implemented in SONAR. MonitorAspect monitors HTTP request, database access and JSP service. The Figure 7 illustrates how the invocation statistics collected by MonitorAspect from those three different instrumentation points are visualized as line charts in JConsole. These data and therefore charts are automatically updated. Figure 8 shows the operations supported by MonitorAspect. The deploy()/undeploy() are used to deploy/undeploy MonitorAspect. After being undeployed, advices defined in MonitorAspect are removed from their targets. MonitorAspect still can be accessed by JConsole. Therefore, it can be redeployed by invoking deploy() from JConsole.

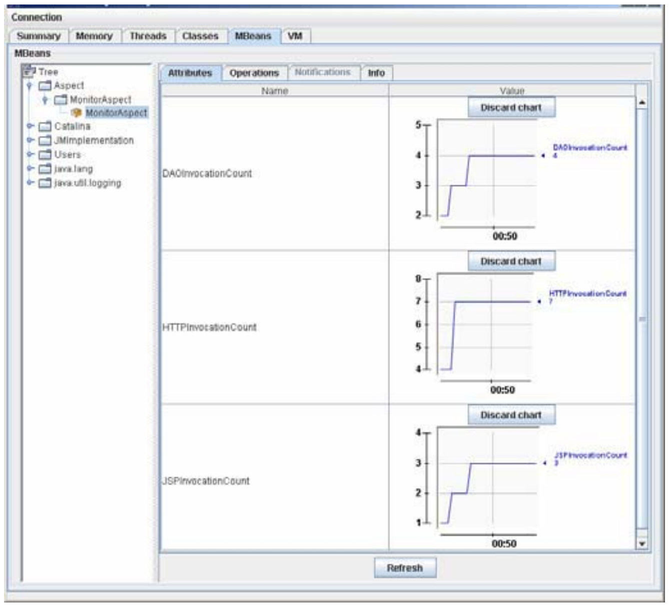


Figure 7. Data from aspects is visualized and can be updated in JConsole.

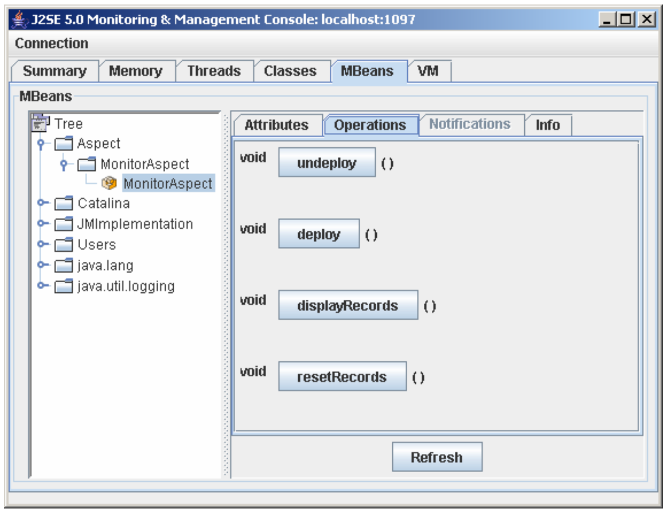


Figure 8. Operations of aspects are listed and can be invoked in JConsole.

4. ANALYSIS/EVALUATION

Currently, there is no JVM that supports schema redefinition of any loaded classes. In other words, changes (such as add, remove or rename fields or methods, change the signatures of methods, or change inheritance) are not allowed at runtime[2][1]. However, dynamic deployment/undeployment of aspects in AspectWerkz requires schema changes to target classes.

To get around with the above restriction, AspectWerkz uses a preparation mechanism to prepare target classes for later deployment/undepolyment at runtime. The special construct, named deployment scope, is used to specify the joinpoints to be prepared – by adding a call to a public static final method that redirects to the target join point. The added indirection will surely introduce overhead. However, such indirection can be inlined by most modern JVMs [1].

As a result, it is not possible to achieve absolute zero impact on the performance and memory footprint on target classes. Table 1 shows the impact of using AspectWerkz, with regarding to class file size and runtime performance. The added size to target class file is around 1000 bytes.

Table 1. Impact on file size and performance.

	Size (bytes)	Runtime Performance (ms)
Original	1,064	0.281
After preparation (no advice weaven)	2,447	0.282
After weaven	1,839 + the size of the aspect(s)	0.328

As indicated in the Table 2, Tomcat server's startup time is significantly increased because it is running under AspectWerkz' online mode – aspects are weaved into target classes when they are loaded into JVM.

Table 2. Startup time of Tomcat. Sampled by running Jakarta Tomcat 5.5.4. Spring Framework 1.1.3, Spring JPetstore and AspectWerkz 2.0 (RC2) with online mode on JDK 1.5.0.

Configuration	Start up time (ms)
Tomcat	5,953
Tomcat with JMX enabled	6,122
Tomcate with AspectWerkz and JMX enabled	27,883

One limitation of the current joinpoint model used in defining aspects in SONAR's XML files is that it is based on the joinpoint model from AspectJ and AspectWerkz. Thus, it is closely targeted to object-oriented programming language model. Many advanced features are not available to systems developed in non-object oriented languages. This makes integration with legacy systems harder. Moreover, AOP provides a low-level code centric approach. It is focused on static structure of code such as control flow. It lacks support for data flow and other dynamic behavior. For example, based on current AOP technology, it is easy to track when a field is accessed but there is no straightforward way to track where the value of the field is passed to and how the value is used in a program.

SONAR does not explicitly support distribution since, currently, no AOP framework provides built-in support for distribution – aspects cross distributed applications/systems. However, by using JMX, aspects deployed in a distributed fashion could ultimately be accessed and managed through JMX Remote API.

[2] As mentioned in Java API's instrumentation section, such restriction might be lifted in the future.

5. RELATED WORK

5.1 Pinpoint

Pinpoint [2,3] is a dynamic analysis methodology that automates problem determination in complex systems by coupling coarse-grained tagging of client requests with data mining techniques. Data mining correlates the believed failures and successes of these client requests as they pass through the system. This combined approach is used to determine which component(s) are most likely to be at fault. Pinpoint has been implemented and used as a framework for root-cause analysis on the J2EE platform that requires no knowledge of the application components.

Pinpoint consists of a communications layer that traces client requests, a failure detector that uses traffic-sniffing and middleware instrumentation, and a data analysis engine. It would be possible to merge Pinpoint and SONAR in such a way that would eliminate the manual instrumentation currently required for Pinpoint's middleware instrumentation.

5.2 Magpie

Magpie's [8] goal is to provide synthesis of runtime data into concise models of system performance [8]. In Magpie, online performance modeling is an operating system service. Magpie's modeling service collates detailed traces from multiple machines, extracts request-specific audit trails, and constructs probabilistic models of request behaviour. . It would be possible to adopt some of the strategies used by Magpie that apply to distribution and performance debugging to extend SONAR to further environments.

5.3 DTrace

DTrace [4] is a unified tracing toolkit for both system and application levels. DTrace can be used to observe, debug and tune system using the D programming language which is designed specifically for tracing. As a result, it is a comprehensive dynamic tracing framework, though only applicable within the Solaris Operating Environment. DTrace attains many of the goals shared by SONAR, but within this proprietary environment.

5.4 PEM/K42

Vertical Profiling[3] is an approach to correlating performance and behavior information across the layers of modern systems (hardware, operating system, virtual machine, application server, and application) to identify causes of performance problems. The Performance and Environment Monitoring (PEM) group and K42 group [6, 9] at IBM Research are getting promising results using this among other approaches to develop effective system diagnosis tools. We believe SONAR to be an early prototype of a tool that would fit with this family.

6. FUTURE WORK

As mentioned above, both DTrace and PEM/K42 provide unified dynamic tracing across operating system, virtual machine and application. The current SONAR implementation lacks an easy way to instrument the operating system and certain aspects of virtual machine. As for a future improvement, we plan to investigate how to unify SONAR with such tool kits in order to provide an efficient means for tasks requiring flexible and customizable tracing crossing all layers in the software stack, but still focusing more on higher level such as application and middleware. Additionally, we plan to further investigate advances in AOP, especially dynamic AOP. We also hope to eventually integrate a high level language specifically for optimization and navigation into SONAR.

7. CONCLUSION

SONAR uses XML, AOP, and JMX in order to achieve a language/framework agnostic, dynamic, manageable, unified framework for system-wide analysis at runtime. We believe the results we have gathered from the prototype to date to be promising in terms of both functionality and performance.

8. ACKNOWLEDGMENTS

We would like to thank Jonathan Appavoo for his inspiration and insights.

9. REFERENCES

[1] AspectWerkz, http://aspectwerkz.codehaus.org/index.html.

[2] Chen, M., Kiciman, E., Accardi, A., Fox, A., Brewer, E. Using Runtime Paths for Macroanalysis. In HotOS IV (2003).

[3] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. Proc. International Conference on Dependable Systems and Networks (IPDS Track), pages 595-604, June 2002.

[4] DTrace, http://www.sun.com/bigadmin/content/dtrace/

[5] JMX, http://java.sun.com/products/JavaManagement.

[6] K42, http://www.research.ibm.com/K42/

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP), 1997.

[8] Magpie, http://research.microsoft.com/projects/magpie/.

[9] PEM, http://www.research.ibm.com/vee04/Duesterwald.pdf

[3] http://www-plan.cs.colorado.edu/~hauswirt/Research/

Dependency Issues in Aspect Weaving

Aditya Varma, T.V. Prabhakar
Department of Computer Science
IIT Kanpur
INDIA
tvp@iitk.ac.in

ABSTRACT

Aspect Oriented Programming is helpful in writing less tangled and easier to maintain code with a greater reuse. Weaving of aspects could be done at compile time, static weaving or at run-time called dynamic weaving. When multiple aspects are woven, the dependencies amongst them might push the system into an inconsistent state. In this paper, we describe the problems that could arise when there are dependencies amongst the aspects being woven. We also present an analysis on whether "immediate or deferred weaving" should be done at runtime.

Keywords

Advice, aspect, dependency preserving aspect weaving, dynamic weaving, emergent properties, inconsistency, weaving.

1. INTRODUCTION

Aspect orientation is a framework for separation of concerns, which crosscut the base functionality of the system. By "crosscutting concerns", we mean those program fragments that cannot be encapsulated in a single module and are scattered over various modules in the system. Different methodologies have evolved to address issues raised by aspects such as AOP [10], subject oriented programming [17], Meta object protocols [2], composition filters [5] etc. In aspect-oriented programming, an "aspect" is a basic unit implementing a concern. The aspects such formed are then woven into the system comprising the base functionality using an aspect weaver. The advantage is improved modularity, efficient code, aspect reuse and a more easily maintainable code. Initially, weaving of aspects started with static weaving - weaving done at compile time. AspectJ [9], Hyper/J [14] are examples of static weaving of aspects. AspectJ is an extended language of Java facilitating features and constructs for writing and inserting aspects at compile time. An aspect is defined in terms of "Join points", "Pointcuts" and "advices". A join point represents a well-defined point in the execution of a program. Point cut represents a collection of join points. An advice represents the behavior to be executed at the join point. Aspect dependence can be defined using the "declare precedence" keyword. Weaving of an aspect into base code requires access to the source code. The base code is transformed by embedding calls to the advice methods. For every dynamic join point in the aspect there is a corresponding static shadow in the source code. Each piece of advice is matched with each static window. If there is a match, a call to the advice method is embedded into the source file.

The main drawback of static aspect weavers is that they do not facilitate the insertion or deletion of aspects during the execution of system. The decision to which aspect is to be woven has to be done beforehand, leaving no chance to reconfigure the system at runtime without aborting and restarting it. Once an aspect is statically woven, identifying aspect specific statements that are added during weaving is difficult. This has led to the advent of dynamic weaving of aspects.

There are numerous examples, which illustrate the necessity for weaving and unweaving aspects at runtime.

We consider the two well-known examples of debugging and load balancing aspects. Let us suppose that an application is being under execution and we want to trace the execution stack or obtain some values of the variables. Assuming that a debugging aspect is written for the purpose, it needs to be woven into the application at runtime. Unfortunately, the static weavers cannot implement this.

Another well-known example is that of load balancing. Let us suppose that a system is designed consisting of multiple servers, multiple clients etc. The configuration of the system needs to be modified depending on the number of the incoming clients and the number of presently available servers. This requirement necessitates that load balancing functionality be written as an aspect and added at runtime.

The two most widely used approaches for dynamic weaving are: "byte code manipulation" and the "interpreter approach [16]". There is also another approach in use called "proxy approach [6]" though not widely used as compared to the former ones. Currently there are tools basing on the above-mentioned approaches like Prose [12], JAC [11], Wool [19], AspectWerkz [15], CLAW, AOP/ST, Jasco etc.

Most of the weaving tools do not consider inter-dependencies that could exist between aspects. They just follow a fixed order (for example, Prose [13] executes aspects in the decreasing order of their insertion time). This could result in a violation of dependencies leading to inconsistencies in the system.

In this paper, we present the problems that arise when dependencies are not considered during weaving of aspects. We start with static weaving and then move on to dynamic weaving. Weaving at runtime also involves an additional parameter, the point in execution at which the aspect enters the code base.

The rest of the paper is structured as follows: In section 2, we present the dependencies and the problems that arise in case of static weaving. We also illustrate an analysis, to be done before the weaving of an aspect. We do the same for dynamic weaving in section 3. Finally, section 5 presents the conclusions.

2. PROBLEMS WITH WEAVING

Dynamic aspect weavers provide a solution to the problem of switching off any aspects, modifying already added ones or inserting any new ones at runtime. The drawback with currently available static and dynamic weavers is that there is no support for specifying the dependency of the aspects. Though some support is available in AspectJ, it is limited in its capabilities. Prose, a dynamic aspect-weaving tool, executes aspects in the reverse order of their insertion. Even in the case of a single aspect, the advices concerning a single join point are executed in the reverse order of their specification.

According to Huang [8], aspects not only interact with the primary abstraction but also with other aspects.

Since weave order cannot be specified, weaving multiple aspects at a time or even a single aspect, deviating from the dependency order would lead to problems in the system. Failure to preserve dependency order of aspects would lead to inconsistencies in various states of the system. These undesirable behavior is termed "emergent properties".

2.1 Dependencies

According to Kienzle [7], aspects are categorized into three kinds: orthogonal, uni-directional and circular.

Orthogonal aspect provides functionality, which is independent of the functionalities of the other aspects or the base application. Ex.- Timing aspect implemented by a timing counter.

Uni-directional aspect provides functionality that is dependent on the functionalities of other aspects in the system. It can be further classified into uni-directional preserving and uni-directional modifying aspects

Uni-directional preserving aspect does not affect the functionality of any other aspect. Ex. – Billing aspect depending on the timing aspect.

Uni-directional modifying aspect modifies the functionality of some part of an application, but transparently without the latter's awareness.

Circular dependency exists when several aspects form a chain of dependencies between them.

2.2 Consistency problems

Essentially, by dependency we mean a write operation followed by a read or a write followed by a write. We can see that orthogonal aspects do not pose any concern with regard to the dependency order issue, but others do. The weaving and execution of aspects can be in a single thread of execution, i.e. no concurrent executions. It can also be done through multiple threads of execution. However, the consistency problems that we will be discussing will be same in both the cases.

We classify the inconsistencies, which occur due to dependency-failing-aspect-executions in a fashion similar to the levels of consistency in the database terminology like "Repeatable read", "Read committed", and "Read uncommitted".

Repeatable read allows only committed records to be read. It also requires that, between two reads of a record by a transaction, no other transaction is allowed to update the record.

Read committed is more relaxed compared to repeatable read in the sense that between two reads of a record by a transaction, the records can be updated by any other committed transaction.

Read uncommitted is the lowest level of consistency. It allows even uncommitted records to be read.

When we consider the problem of inconsistency caused due to the failure in preserving aspect dependencies during execution, the notion of "problem" depends on the level of inconsistency acceptable by the user and the domain under consideration. For Ex. – If we consider an application, where the user queries for search results on the web during the execution of the application, the user may well be comfortable with the partial results or he may as well be less bothered even the results change for two consecutive searches. In that case, we can say that the user allows for weaker levels of consistency.

The upcoming discussion deals with the before and after advice declarations. Advice defines the additional code that should be executed at the join points. 'Before advice' is executed when a join point (say field modification join point) is reached. When the control returns from the join point, 'after advice' is executed. The problems illustrate only generic instances, and do not deal with any real time domain problems.

The discussion assumes that the weaving follows the ordering as in tools like Prose, JAC etc. That is if there are m aspects woven, the execution order goes as:

$$\mathbf{B_1\ B_2\ B_3 \ldots B_m\ \ F\ \ A_1\ A_2\ A_3 \ldots A_m.}$$

B – before advice; **F** – base functionality; **A** – after advice.

Some tools like AspectJ weave in this fashion:

$$B_1\ B_2\ B_3 \ldots B_m\ \ F\ \ A_m \ldots A_3\ A_2\ A_1.$$

We base our discussion on the former. The analysis for the latter can be done on similar lines.

If we are weaving at runtime, the program counter could be at any point along this sequence at the time of weaving. This gives rise to different kinds of problems that we will discuss further. We assume that the developer of an aspect has prior knowledge of the aspects woven into the system. Therefore, the developer while providing the aspect for weaving at runtime specifies its dependencies with the existing aspects.

2.2.1 Static weaving

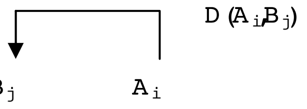
We begin our analysis with the static case, observe the dependency problems and move on to dynamic weaving. As said before, AspectJ facilitates for specifying aspect priorities. However, it is limited in its capabilities. Giving a fixed order to the weaving of aspects (like Prose), would lead to inconsistent states in the system.

Now, we will try to analyze the various dependencies that need to be considered in the weaving of aspects.

We use the following notation to specify the dependencies:

$D(A_i, B_j)$: There is a dependency of A_i on B_j

A_i reads/writes a value written by B_j



Whenever we say 'not solvable' in the future discussion, we mean that an ordering, satisfying the constraints cannot be obtained.

2.2.2 Problems with Static Weaving

There are three cases to consider:

Case 1: If there are 'm' aspects and no dependencies are specified between them, then the aspects can be woven in any order.

Case 2: If a single aspect is to be woven, the order of weaving the aspect is not concerned by whether it's after advice has a dependency on the before advice or not.

The weave order would be:

$$B_1\ \ F\ \ A_1$$

Case 3: If two or more than two aspects are to be woven, and if dependencies exist between them, then those dependencies have to be analysed as to whether a valid ordering satisfying these constraints can be obtained or not.

(a) If the aspects are orthogonal, they can be woven in any order.

(b) If the aspects are non-orthogonal and some of them are uni-directional, then we need to consider the dependencies specified between them to obtain a valid ordering.

We assume that by default, the weaving order is $B_2\ B_1\ F\ A_2\ A_1$.

The dependencies between two aspects asp1 and asp2 can be any of the below specified ones:

$D(A_1, B_1)$

$D(A_2, B_2)$

$D(B_1, B_2)$

$D(B_2, B_1)$

$D(A_1, B_2)$

$D(A_2, B_1)$

$D(A_1, A_2)$

$D(A_2, A_1)$

We left out $D(B_i, A_j)$, as we assume that before advices do not depend on the after advices.

If $D(A_2, A_1)$ or $D(B_2, B_1)$ or both is/are specified, then the weave order would be in contrast with the required. Violating the dependency order would lead to **read committed and read uncommitted** inconsistencies. The corresponding advices have to be swapped to maintain the dependency.

Similarly, we can easily state that if a cyclic dependency occurs between the specified dependencies, then the aspects cannot be woven. Swapping would not solve the problem.

Ex:- $D(A_1, A_2)$ and $D(A_2, A_1)$; $D(B_1, B_2)$ and $D(B_2, B_1)$.

No other combinations of the above-mentioned dependencies would lead to problems.

The reason is that of the above eight dependencies, four are of an after advice depending on a before advice. These dependencies whether present or not would not be of any problem. The reason is that after advices always come later in execution relative to the before advices.

Of the remaining four dependencies, the dependencies involving only before advices would not lead to any inconsistency in the after advices. Therefore, only a cyclic ordering between the dependencies as specified before, would lead to an unsolvable problem.

In case of a possible weave ordering based on the dependencies, the order of the advice execution would be any of the below:

$B_2\ B_1\ F\ A_2\ A_1$

$B_1\ B_2\ F\ A_1\ A_2$

$B_1\ B_2\ F\ A_2\ A_1$

$B_2\ B_1\ F\ A_1\ A_2$

In the case of more than two aspects to be woven into the system, the analysis would be the same with the same problems occurring, as specified above.

Algorithm:

```
Static-weave (i,j) //Default weaving order: $B_j B_i F A_j A_i$
{
    if (i & j) are orthogonal
        Weaving can be done. Return true.
    else
    if ( (D ($B_i$,$B_j$) & D ($B_j$,$B_i$)) || (D ($A_i$,$A_j$) & D ($A_j$,$A_i$)) )
        Weaving cannot be done. Return false.
    else
    if (D ($B_j$,$B_i$) || D ($A_j$,$A_i$) )
        Weaving can be done
        by swapping the corresponding advices.
        Return true.
}
```

3. DYNAMIC WEAVING

Before going into the discussion, we assume that a valid ordering is obtained for the statically woven aspects. Now coming to dynamic weaving, we strive to obtain an order equivalent to that of the static case i.e.

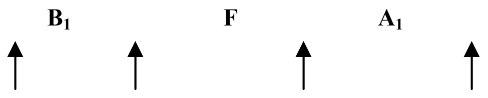
If D (X , Y) is required: Weave Y before X.

If we consider an aspect that is already woven and at runtime, a new aspect or a modified version of the same aspect is to be woven, there are different weaving points depending on when the aspect arrives:

Since the weaving command could come at any arbitrary time, the points at which weaving can be done, are as shown below:

Ex - **B_1 F A_1**

If an aspect is to be woven at runtime, the points to be considered are:



The arrows represent the possible points for weaving.

Therefore, in dynamic weaving, two issues come into consideration. One is, whether a valid weave order is possible for the given dependencies. If so, can the weaving be done immediately? There will be instances where weaving can be done, but not at the very point of arrival of the aspect. Weaving of the aspect may need to be deferred to a later stage in execution. All this we will see in our coming subsections.

We have considered three types of scenarios to present the dependency order problem. They will be dealt in succession.

3.1 No pre-existing aspects

In our first scenario, we assume that no aspects are woven statically. An aspect asp1 is to be woven at runtime into the system.

In this situation, the aspect can or cannot be woven directly at the very instance of arrival. It depends on whether the after advice has a dependency on the before advice or not. It also depends on the point of execution at which the aspect arrived.

If the aspect arrives before the execution of the join point functionality, then the aspect can be immediately woven.

If the aspect arrives after the execution of the join point functionality, then the dependency of the after advice on the before advice has to be considered. If dependency exists, then weaving should not be performed at that instant. It should be deferred to a later stage in execution. Otherwise, weaving can be performed.

Therefore, the final advice execution order would be:

B_1 F A_1 (if aspect arrives before 'F')

or

F A_1 (if aspect arrives after 'F' and weaving is allowed)

or

F (if aspect arrives after 'F' and weaving is deferred)

Algorithm: //Aspect i has to be inserted

```
Dynamic-weave (i, weave-point)
{
    if (weave-point before F ) do weave.
    if (weave-point after F )
        if (D ($A_i$,$B_i$) ) defer weaving
        else
        do weave.
}
```

3.2 One aspect already exists

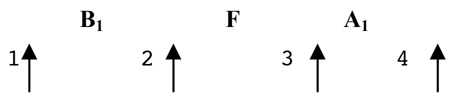
In this scenario, we assume that we already have an aspect asp1 woven into the system at static or runtime. A new aspect asp2 is to be woven. Here two cases need to be considered.

If asp2 is orthogonal, it can be woven into the system. However, decision should be taken as to whether it can be woven immediately. The analysis required is the same as that considered in the previous section. It is not restated to avoid redundancy.

If asp2 depends on the services of the asp1, then the dependencies specified should be considered before making the decision.

Before weaving: **B_1 F A_1**

The new aspect may arrive at any of the four weaving points shown below:



In the below analysis, each case refers to a weaving point.

Case 1:

If asp2 is to be woven before the execution of B_1, then it can be directly woven, if a valid ordering satisfying the inter-aspect dependencies is obtained. For this, the analysis to be done is the same as that done in the case of static weaving, where two or more aspects are to be woven.

The final advice execution order can be any of the following:

B_2 B_1 F A_2 A_1

B_1 B_2 F A_1 A_2

B_1 B_2 F A_2 A_1

B_2 B_1 F A_1 A_2

Case 2:

In this case, initially it should be checked whether **D(B_1 , B_2)** is required.

If **D(B_1 , B_2)** does not exist, then the analysis would be the same as that of Case 3 in static weaving. If **B_2** comes before **B_1** in advice execution order, **B_2** would not be executed.

If **D(B_1 , B_2)** exists, then weaving is attempted without the execution of **B_2**.

In both the above situations, the problem of weaving the aspect without **B_2**'s execution depends on whether **A_2** has dependency on **B_2** and the inter-aspect dependencies.

If **D(A_2 , B_2)** does not exist and the inter-aspect dependencies provide a weave order (analysis is the same as that of Case 3 in static weaving), then weaving can be performed.

If **D(A_2 , B_2)** exists, we should check whether a valid weave ordering can be obtained satisfying the inter-aspect dependencies (analysis is the same as that of Case 3 in static weaving). If yes, then the weaving should be postponed to a later point in execution. Weaving the aspect immediately would lead to **read uncommitted inconsistency** as **A_2** may access data written by **B_2** (which does not execute).

If a valid ordering cannot be obtained with the inter-aspect dependencies, weaving should not be done.

The final advice execution order can be any of the following:

B_1 B_2 F A_2 A_1

or

B_1 F A_2 A_1

or

B_1 F A_1 A_2

or

B_1 F A_1

Case 3 & 4:

In this case, **B_2** is not executed. Therefore, the analysis should be done by considering the dependence of **A_2** on **B_2**.

If **D(A_2 , B_2)** exists and a valid ordering can be obtained with the inter-aspect dependencies, weaving should not be done immediately. It is deferred to a later stage. Weaving the aspect immediately would lead to **read uncommitted inconsistency** as A_2 may access data written by B_2 (which does not execute).

If **D(A_2 , B_2)** does not exist and the inter-aspect dependencies provide a valid weave order (analysis same as that of case 3 in static weaving), then weaving can be performed.

If a valid ordering cannot be obtained with the inter-aspect dependencies, weaving should not be done.

The final advice execution order can be any of the following:

B_1 F A_2 A_1

or

B_1 F A_1 A_2

or

B_1 F A_1

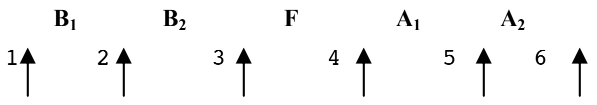
Issues concerning the weaving of multiple aspects at a time involve considerations of multiple issues specified above. What can be stated is that concerns involving multiple aspects can be resolved into issues involving two aspects, one aspect at a time.

```
Algorithm: //Aspect i already exists & aspect j has to be woven
          //Before weaving: $B_i$ F $A_i$
     Dynamic-weave (j, weave-point)
     {
        if (weave-point before $B_i$)
               if (Static-weave (i, j)) do weave.

        if (weave-point after $B_i$ & weave-point before F)
        {
           if (!D ($B_i$, $B_j$))
                  if (Static-weave (i, j))
                      if (D ($B_j$, $B_i$)) goto $L_2$.
                      else
                      do weave.
           else
                 goto $L_2$.
        }
$L_2$: { if ( !D ($A_j$ ,$B_j$) )
          if ( Static-weave (i,j) ) do weave.
      else
          if ( Static-weave (i,j) ) defer weaving.
      }
 if ( weave-point after F ) goto $L_2$.

 if ( weave-point after $A_i$ )
        if ( Static-weave (i,j) ) do weave.
}
```

3.3 Two aspects

In this scenario, we assume that two aspects asp1 & asp2 are already woven into the system and are in execution. We need to examine the situation when one of the aspects changes. That is, it is modified and sent to be re-woven.

The weaving points to be considered are:



In the following analysis, each case refers to a weaving point.

Modified asp1 **to be rewoven**

Case 1 :

If weaving is at the point 1, then the default weaving order of the advices would be:

B_1' B_2 F A_1' A_2

The analysis to be done to determine the possibility of weaving is the same as that done in the case of static weaving, where two or more aspects are to be woven.

The final advice execution order can be any of the following:

$B_2\ B_1'\ F\ A_2\ A_1'$

$B_1'\ B_2\ F\ A_1'\ A_2$

$B_1'\ B_2\ F\ A_2\ A_1'$

$B_2\ B_1'\ F\ A_1'\ A_2$

Case 2 - 4 :

In this scenario, older version B_1 would be executed.

To weave the newly modified aspect asp1, dependence of A_1' on B_1' should be checked.

If $D(A_1', B_1')$ exists, weaving of the aspect should not be done immediately. It must be deferred to a later stage in execution.

If $D(A_1', B_1')$ doesn't exists, weaving of the aspect is attempted considering the inter-aspect dependencies. The analysis would be the same as done in case of static weaving involving two or more aspects.

In case, weaving cannot be done or deferred to a later execution point, executions with the older versions will continue.

The final advice execution order can be any of the following:

$B_1\ B_2\ F\ A_2\ A_1'$

or

$B_1\ B_2\ F\ A_1'\ A_2$

or

$B_1\ B_2\ F\ A_1\ A_2$

Case 5 & 6:

Weaving can be done if a valid order can be obtained from the aspect dependencies specified.

The final advice execution order would be:

$B_1\ B_2\ F\ A_1\ A_2$

The analysis is similar in case of weaving the modified version of asp2.

Similarly, the scenario of a single orthogonal modified aspect to be rewoven is a special case of this section.

There may be some domain specific situations, where the version difference between advice executions may not be of problem. In such a case, in Case 2 – 4, modified aspect can be woven though $D(A_1', B_1')$ exists and B_1' is not executed.

Algorithm:

//Aspects i & j were already inserted. Aspect i′ (modified i) has to be woven.

//Before weaving: B_i B_j F A_i A_j

```
Dynamic-weave (i′, weave-point)
{
   if ( weave-point before $B_i$ )
          if ( Static-weave (i′,j) ) do weave.

   if ( weave-point after $B_i$ & weave-point before F )
          goto $L_2$.

   if ( weave-point after F ) goto $L_2$.

   $L_2$:{ if ( !D ($A_{j'}$ ,$B_{j'}$) )
           if ( Static-weave (i′,j) ) do weave.
```

```
        else
            if ( Static-weave (i', j) ) defer weaving.
        }
    if (weave-point after $A_i$)
            if ( Static-weave (i', j) ) do weave.
}
```

4. IMPLEMENTATION

A dynamic aspect weaver requires the dependencies between the aspects. The class constructs written for the tool should facilitate for the specification of dependencies (as in the case of aspectJ that provides "declare precedence" keyword). Then the problem can be solved. In a sense, the dependencies should be specified along with aspects for insertion.

Once the dependencies are specified, the analyses discussed in sections two & three, should be done based on the run-time information for dynamic weaving. For this JVMDI can be used to obtain information of the stack frame and method executions etc. during run-time. Only then, the decision to weave the aspect or not, and if so when in the execution phase, is to be made.

When an aspect cannot be woven, feedback is provided to the aspect developer specifying the dependent aspects. The system will continue its operation with the woven aspects.

We are in the process of implementing a dynamic aspect-weaving tool with the feature of dependency preservation. The interpreter approach wherein hooks are inserted as breakpoints is being applied for dynamic weaving.

5. CONCLUSION

In this paper, we discussed the issues that arise when there are dependencies between aspects. We initially considered the static case of weaving. We then presented an analysis that is necessary before an aspect can be woven at runtime. Essentially one needs to check whether a weaving is permissible at all, and if yes, at what point of execution can the weaving happen. An implementation incorporating these ideas is in progress.

6. REFERENCES

[1] Alonso, G., A. Popovici, T. Gross. Dynamic Weaving for Aspect Oriented Programming. In 1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands, Apr. 2002.

[2] Bobrow, D.G., G. Kiczales and J. des Rivieres., The Art of Meta-Object Protocol, MIT Press 1991.

[3] Böllert, K.: "On Weaving Aspects". European Conference on Object-Oriented Programming (ECOOP), Workshop on Aspect Oriented Programming, June 1999.

[4] Bradley, J.T., An Examination of Aspect-Oriented Programming in Industry. Honors Thesis, Advisor **Roger T. Alexander**, Technical Report CS-03-108, Department of Computer Science, Colorado State University, Fort Collins, Colorado 2003

http://www.cs.colostate.edu/~rta/publications/CS-03-108.pdf

[5] Composition Filters homepage http://trese.cs.utwente.nl/oldhtml/composition_filters

[6] Constantinides, C.A., T. Elrad, M.E. Fayad, Netinant P., Designing an aspect-oriented framework in object-oriented environment, ACM Computing surveys, March 2000.

[7] Jörg Kienzle, Yang Yu, Jie Xiong, On Composition and Reuse of Aspects.

[8] Huang, J., Experience using AspectJ to implement Cord, OOPSLA 2000.

[9] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold: "Getting Started with AspectJ", Communications of the ACM, October 2001

[10] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. In 1997, European Conf. on Object-Oriented Programming (ECOOP, 97), pages 220-242. Springer Verlag, 1997.

[11] Pawlak, R., L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java," in Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001), LNCS 2192, pp. 1–24, Springer, 2001.

[12] Popovici, A., T. Gross, and G. Alonso., Just-In-Time Aspects: Efficient Dynamic Weaving for Java, AOSD 2003, Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, Massachusetts.

[13] Popovici, A., T. Gross, and G. Alonso, " PROSE website http://prose.ethz.ch/Wiki.jsp?page=AboutProse" 2003.

[14] Tarr, P., H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-dimensional Separation of Concerns. In 1999 Intl. Conf. on Software Engineering, pages107-119, Los Angeles, CA, USA, 1999.

[15] Vasseur, A., and J Boner, "AspectWerkz Web Site", http://aspectwerkz.codehaus.org" 2004.

[16] Wasif Gilani and Olaf Spinczyk. A Family of Aspect Dynamic Weavers, AOSD 2004 – Dynamic Aspects Workshop, Lanchester, UK, Mar. 2004

[17] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In Proceedings of OOPSLA '93.

[18] Xerox Corporation. The AspectJ Programming Guide. Online Documentation, 2002. http://www.aspectj.org/.

[19] Yoshiki Sato, Shigeru Chiba, Michiaki Tatsubori, A Selective, Just-In-Time Aspect Weaver, Proceedings of the second international conference on Generative programming and component engineering, Erfurt, Germany, Year of Publication: 2003

Quantifying over Dynamic Properties of Program Execution

Christoph Bockisch Mira Mezini Klaus Ostermann
Software Technology Group
Darmstadt University of Technology, Germany
{bockisch,mezini,ostermann}@informatik.tu-darmstadt.de

ABSTRACT

In a pointcut we want to fully specify the points in the execution of a program at which an advice is to be executed. The pointcut languages of current aspect-oriented programming languages only provide limited support for specifying points in the execution that do not directly map to points in the program code. As a result, an aspect programmer has to implement logic to keep track of certain runtime properties manually. This logic is detached from the advice's pointcut. In this paper, we identify two common patterns of dynamic properties on which advice rely. We present pointcut designators that allow to declaratively specify the join points based on runtime properties in a pointcut and outline a possible implementation.

1. INTRODUCTION

Pointcut-and-advice units are used in AspectJ-like languages [9] to modularize crosscutting concerns. The advice part is a piece of code and a pointcut is a special expression specifying a set of join points at which the advice must be executed. Join points are points in the execution of a program, for example reading a field or calling a method. Each join point has two parts of context. First, there is the static context, which can be retrieved by analyzing a static represenation of the program, such as the source code or bytecode. For a "field-read" join point the static context is, among others, the field's name and the class in which the field-read expression stands. Second, there is the dynamic context, which is made up of the situation at runtime when the join point is executed. Again, using "field-read" exemplarily, the dynamic context can be the value of the field, the methods on the current call stack, or even the complete history of the program execution.

The intuition behind using aspect-oriented programming (A-OP) is that a concern's implementation is well localized and the knowledge of when an advice must be executed is bundled to the advice. This implies two goals we want to achieve when writing aspect-oriented code: first we want to execute code in different contexts *implicitly*, second we want to express the knowledge of these contexts *declaratively.*

The pointcut language of AspectJ has extensive support for selecting join points based on the static context. We call this pointcuts based on *static properties.* In addition, it also provides pointcut designators (PCD) for selecting join points by their dynamic context - pointcuts based on *dynamic properties.* Using the `cflow` or `cflowbelow` PCDs one can specify the methods that must be on the call stack when the join point is executed. With the PCDs `target`, `this` and `args` one can specify the dynamic type of the receiver object, active object or argument objects. However, there are more dynamic properties that can be relevant for the execution of an advice at runtime.

In this paper we give examples of aspects that show why a better support for pointcuts based on dynamic properties is needed. They show that it is not possible to specify the *precise* set of join points *declaratively* with current pointcut languages. Additionally, the relevant parts of the join points' dynamic context must be accessed programma-tically. For the given examples we have implemented aspects in Alpha [1, 11], an experimental aspect-oriented language that allows pointcuts to reason over dynamic properties of a program. It is possible in Alpha to define new pointcut designators in a declarative way. We present the resulting PCDs in a pseudo AspectJ notation, so that it is not necessary to introduce Alpha in detail in this paper.

The remainder of this paper is organized as follows. In the next section we show two examples of aspects which heavily rely on dynamic properties. We discuss how these aspects can be implemented in current aspect-oriented languages. Section 3 defines a notation for extended dynamic pointcut designators in an AspectJ-like pointcut language. A conclusion is given in section 4. Section 5 presents our ongoing work as well as some related work.

2. CURRENT STATE

In this section, we present two examples of aspects for which a pointcut must quantify over complex dynamic properties. We discuss an implementation for those examples in a conventional aspect oriented language where aspects are active globally and during the whole program execution, and in alternative languages where aspects can be deployed dynamically. This discussion shows that neither of them satisfies our goals.

The first example is a text editor application (Fig. 1). At any time, the editor has at most one document opened. Documents can be created, edited and saved. For this application, we want to write an aspect that prevents the quitting of the application and the creation of a new document when the current document is in a dirty state, i.e., there are unsaved changes.

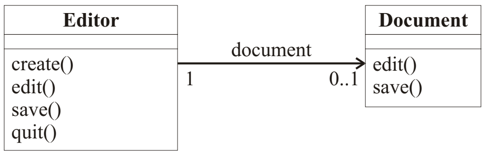


Figure 1: Base code for the editor example.

A natural description of the pointcut for this problem is: "calls to the methods `create()` or `quit()`, when there has been a previous call to the method `edit()` but no call to `save()` or `create()` since then". Fig. 2 illustrates this.

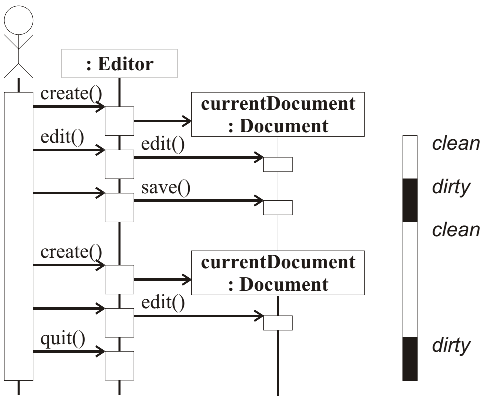


Figure 2: A possible sequence of method calls in the editor example. The last call to `quit()` should be prevented because the document is dirty at this time.

The second example is a graphical editor (Fig. 3). The program has a hierarchy of `Figure` classes that can be displayed by an instance of the class `Display`. To reflect the current state of the `Figure` objects on a `Display`, we want to define an advice calling the `Display`'s method `draw()` with the pointcut: "calls to a setter method of a `Figure` to which a `Display` points" (see Fig. 4).

2.1 AspectJ

For the discussion of a conventional aspect-oriented language we exemplarily use the AspectJ language [9]. This uses static weaving, i.e., it weaves the aspects into the program at pre-runtime. We call this "static deployment" as opposed to "dynamic deployment" discussed in the next subsection.

A possible AspectJ implementation for the editor example is given in Listing 1. The intended pointcut has a static part, namely call-instructions to the methods `quit()` or `create()`, and a dynamic part. For the dynamic part the history of the program execution must be accessed to decide

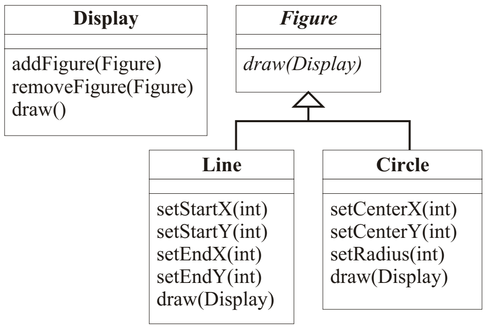


Figure 3: Base code for the display updating example.

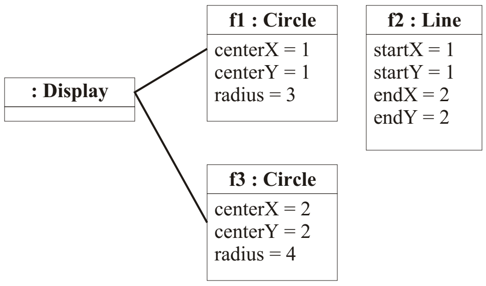


Figure 4: A possible configuration of objects in the display updating example. Changes to `f1` and `f3` should trigger a call to the `Display`'s `draw()` method.

whether there have been calls to the method `edit()` since the last call to `save()`.

AspectJ's pointcut language only allows to involve that part of the history which is still on the call stack, i.e., by the pointcut designators `cflow` and `cflowbelow`. However, in our example the relevant calls to `edit()` or `save()` are not, in general, on the stack when the method `quit()` or `create()` is called. Hence, we store this part of the history in a field of the aspect (`documentDirty`) which is maintained by two separate advice. The one advice setting the state to "dirty" after the `edit()` method has been called. And the other one setting the state to "clean" after one of the methods `save()` or `create()` have been called.

The pointcut for the advice of preventing the disposal of dirty documents is defined to match each call to `quit()` and `create()`, which is the static part. As the dynamic part an `if` pointcut designator is added. This one checks the state of the field `documentDirty` so that the pointcut only matches when the document is indeed dirty.

```
public aspect PreventDirtyDispose {
  private static boolean documentDirty;
  pointcut makeDocumentDirty():
    call(void Editor.edit());
  pointcut makeDocumentClean():
    call(void Editor.save()) ||
    call(void Editor.create());
  after(): makeDocumentDirty() {
    documentDirty = true;
  }
  after(): makeDocumentClean() {
    documentDirty = false;
  }
  pointcut disposeDocument():
    (call(void Editor.quit()) ||
      call(void Editor.create())) &&
    if(documentDirty);
  void around(): disposeDocument() {
    // prevent quitting or creating a new document
  }
}
```

Listing 1: An AspectJ aspect that prevents the disposal of dirty documents.

By using advice to keep book of the execution history, the aspect of preventing the disposal of a dirty document is still localized. However, the knowledge of when the document is dirty is spread over several advice. As a result, it is not specified in a declarative manner but uses conditional logic.

For the second example, the intended pointcut also has a static and a dynamic part. The static part says that calls to setter methods on `Figure` objects are selected, and the dynamic part says that the `Figure` objects must be reachable from a `Display` object. The only dynamic property of the target object we can specify in AspectJ is its dynamic type. This is done by using the `target` pointcut designator (similarly there are the designators `this` refering to the active object and `args` refering to arguments of the join point). But we can not decide via the object type whether the target is reachable from a `Display` object, or not.

The display update example is basically an instance of the observer pattern [4] for which [6] presents an abstract implementation in AspectJ. Listing 2 shows a concrete adaptation that implements the presented example. The code for implementing the updating behavior is localized in the aspect.

A pointcut is specified that matches each potential join point, i.e., each call to a method whose name starts with `set` on a `Figure`. But actually the call to a setter yields a join point only if the target is an instance of `Figure` that can be reached from a `Display` object. Therefore, an `if` pointcut designator is added to specify the intended dynamic property. To make the dynamic property of "reachability" accessible to the pointcut we maintain a mapping between `Figure` and `Display` objects by separate advice.

```
public aspect DisplayUpdate {
  static private Hashtable perFigureDisplays
    = new Hashtable();
  static private Set getDisplays(Figure figure) {
    Set result = (Set) perFigureDisplays.
      get(figure);
    if(result == null) {
      result = new HashSet();
      perFigureDisplays.put(figure, result);
    }
    return result;
  }
  pointcut addFigure(Figure f, Display d) :
    call(void Display.addFigure(Figure)) &&
    args(f) && target(d);
  after(Figure f, Display d): addFigure(f, d) {
    getDisplays(f).add(d);
  }
  pointcut removeFigure(Figure f, Display d) :
    call(void Display.removeFigure(Figure)) &&
    args(f) && target(d);
  after(Figure f, Display d):
    removeFigure(f, d) {
    getDisplays(f).remove(d);
  }
  pointcut change(Figure f) :
    call(* Figure.set*(..)) &&
    target(f) && if(!getDisplays(f).isEmpty());
  after(Figure f): change(f) {
    Iterator iterator =
      getDisplays(f).iterator();
    while(iterator.hasNext()) {
      ((Display) iterator.next()).draw();
    }
  }
}
```

Listing 2: An AspectJ aspect that implements the display updating concern.

As in the editor example, the aspect of display updating is localized, but it is not possible to specify the intended pointcut declaratively. The pointcut is even harder to understand because the intended dynamic property is only accessed indirectly by the maintenance logic. The question "is an object `a` reachable from another object `b`?" is not answered by investigating the object heap. It is rather approximated by investigating method calls, namely calls to `addFigure()` and `removeFigure()`.

2.2 Other Approaches

In AspectJ the pointcuts are evaluated at weave-time. For pointcuts relying on static properties the evaluation results in points in the program code that directly correspond to the specified points in the execution. As a result the advice can be woven there. For pointcuts that specify dynamic properties, the evaluation only results in potential join points. AspectJ generates code that checks if the dynamic properties are satisfied there. The advice is only executed if the check succeeds [8]. This is also the case for the `if` pointcut designator used in the last subsection.

There are other approaches of aspect-oriented languages that allow aspects to be deployed at runtime [10, 2] or even to deploy aspects for single objects [3, 12] (we will refer to this as instance-local deployment). In such languages, it is possible to define pointcuts that only depend on static properties and deploy/undeploy an appropriate advice when the dynamic property becomes satisfied respectively unsatisfied. In the editor example, we would write the pointcut `call(* Editor.create()) || call(void Editor.quit())`. The advice would be deployed after the method `edit()` has been called and undeployed after a call to `save()` or `create()`. A more general description of this approach is given in Morphing Aspects [5].

With instance-local deployment, we can write the pointcut for the display update example by only specifying the static part of it, such as `call(void Figure.set*(..))`. An advice with this pointcut is deployed for `Figure` objects that are added to a `Display` and undpeloyed for each `Figure` that is removed from a `Display`.

Similar to the presented examples, the deployment can also be realized as an advice. However, when using dynamic deployment, the knowledge of when the advice really must be executed is also not expressed declaratively as it was postulated as a goal of AOP ealier in this paper. The lack of runtime checks before executing the advice leads to improved performance [7], but does not help writing aspect-oriented code more clearly, at least for the two presented problems.

3. EXTENDED POINTCUT DESIGNATORS FOR DYNAMIC PROPERTIES

In this section, we show how to implement the examples from section 2 in Alpha [1, 11], a toy aspect-oriented language, where pointcuts can be specified based on a rich set of dynamic properties. To accomplish this, facts are generated in a Prolog [13] database at relevant join points[1]. This database contains facts about (1) the abstract syntax tree of the program, (2) the type hierarchy, (3) a complete history of the execution trace up to the current point in execution, and (4) the current content of the heap. Pointcuts are written as Prolog queries which provides us with functional abstraction for pointcut designators. Everytime a fact is added to the database, all pointcuts are evaluated. When the join point matches a pointcut, the associated advice is executed. All this combined allows us to define high-level pointcut designators that rely on a rich set of static and dynamic properties of join points.

AspectJ is widely known and thus we present the resulting high-level pointcut designators as suggestions for extensions to AspectJ's pointcut language. By doing so we can omit an introduction to Alpha which is not necessary for argumenting the need of better pointcut designators. These extensions are tailor-made for the presented examples and are not useful in general. The general pointcut designators implemented as extensions to Alpha can be downloaded from [1].

To specify the control flow in which join points can occur it must be possible to specify when the control flow is entered and when it is exited. To stay conform with AspectJ, one pointcut is specified that describes the join points at which the control flow is entered and one pointcut can be specified at which the control flow is left. As opposed to the PCDs `cflow` and `cflowbelow` both pointcuts generally are distinct. Further, the points in the execution to be specified can be just before a join point or just after a join point (similar to the difference of a `before` and an `after` advice), see Fig. 5. Thus, we suggest to add the following pointcut designators to AspectJ.

```
afterstart(<pointcut>)
afterend(<pointcut>)
beforestart(<pointcut>)
beforeend(<pointcut>)
```

With these extensions, the pointcut for the editor example can be specified as in Listing 3. The first part of the pointcut specifies method calls that lead to the disposal of the current document, as in Listing 1. The `afterend` and `beforeend` PCDs declare that join points can only match

[1]Alpha uses static optimization techniques to find the smallest possible set of join point shadows. This process can be compared to the evaluation of pointcuts by the AspectJ weaver [8].

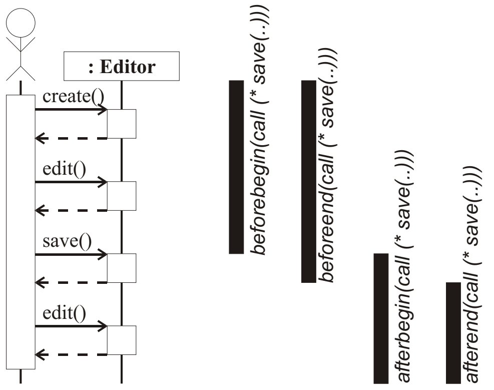

: Editor
create()
edit()
save()
edit()
beforebegin(call (* save(..)))
beforeend(call (* save(..)))
afterbegin(call (* save(..)))
afterend(call (* save(..)))

Figure 5: Join points specified by the new control flow based pointcut designators.

after the current document has been put into a dirty state (i.e., the method `edit()` has been called), and before the document has been made clean. Of course, we can not look into the future to see whether the `beforeend` pointcut will eventually match. Thus the latter PCD is implemented in a way that the complete pointcut only matches when the `afterend` pointcut has already been matched during the execution, but the `beforeend` pointcut has not matched since then or never at all. If any of these pointcuts has been matched several times in the execution, only the last time is taken into account. The right ruler in Fig. 2 is black for the time spans at which both conditions are satisfied. Join points can only occur during those time spans.

```
pointcut disposeDirtyDocument():
  (call(void Editor.quit()) ||
    call(void Editor.create())) &&
  afterend(call(void Editor.edit()) &&
  beforeend(call(void Editor.create()) ||
            call(void Editor.save()));
```

Listing 3: A pointcut for the editor example using the extended pointcut language.

For object-graph based pointcuts we have identified the need for a pointcut designator based on the "reachability" property for the target in the context of the join point. We suggest the pointcut designator `targetreachable(<type>)` to select join points where the target is reachable from an object of type `<type>`. With this designator we can define the pointcut for the display update example declaratively. As is possible with the `target` PCD in AspectJ, values from the join point's context can be bound with the `targetreachable` pointcut designator. Unlike in AspectJ it is, however, possible that several `Display` object have the property that the `Figure` is reachable from them. In this case the associated advice is called once for each value in the context.

```
pointcut changeDisplayedFigure():
  call(void Figure.set*(..)) &&
  targetreachable(Display d);
```

A similar pointcut designator can be added to specify the reachability property for the active object, i.e., `thisreachable(<type>)`. In our prototype the reachability property can also be specified for argument objects. But we will not present this here because this PCD requires a different syntax than `targetreachable` and `thisreachable`.

4. CONCLUSION

With the possibility of defining *precise* pointcuts in a *declarative* way as presented in the last section, the meaning of an advice becomes more clear. In the presented AspectJ implementation of the aspects in section 2 we used the `if` PCD to refer to dynamic properties in the pointcuts. Thus, you have to understand the helper-advice for maintaining the state accessed in the `if` PCD before you can understand which join points are selected.

What's more the presented examples each represent a pattern that occurs repeatedly. For example the specification of a control flow as in the editor example is also used by other concerns. Let's assume we want to add an aspect for recording macros to the editor. The user can start and stop the recording by pressing a button which results in a call to the method `startMacro()` and `stopMacro()` accordingly. Between two such method calls, each call to `edit()` must be recorded. The problem is very similar to the prevention of disposing a dirty document, but we have to implement the complete logic for keeping track of the execution history, over again.

We used AspectJ's pointcut language to describe our extensions to the pointcut language because AspectJ is widely known. However, we suggest to exetend the pointcut languages not only of conventional AO lanuages with globally deployed aspects. The suggested extensions are complementary to the deployment mechanism. Steamloom [3], for example, uses an AspectJ-like pointcut language, too, including the PCDs based on dynamic properties that already exist in AspectJ, such as `cflow`.

5. ONGOING AND RELATED WORK

We are currently classifying properties on which pointcuts can be founded. The properties are rated based on the complexity needed for an efficient implementation. We are also exploring the possibility to implement pointcuts based on dynamic properties in an execution environment with dynamic and instance-local deployment that is being developed at our group (Steamloom [3]).

[14] presents an extension to AspectJ where temporal relations between events can be specified in a pointcut that must be true to select a join point at runtime. The relations are defined using context free grammars. The authors of [14] do not, however, analyze the need for expressing dynamic properties in pointcuts, in general. We see this extension as a possible notation for pointcut designators based on the execution history as well as a possible implementation of such PCDs.

6. REFERENCES

[1] Alpha project. http://www.st.informatik.tu-darmstadt.de/pages/projects/alpha/.

[2] AspectS homepage, 2003. http://www.prakinf.tuilmenau.de/ hirsch/Projects/Squeak/AspectS/.

[3] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD 2004 Proceedings*. ACM Press, 2004.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[5] S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD*, pages 46–55, 2004.

[6] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings OOPSLA '02. ACM SIGPLAN Notices 37(11)*, pages 161–173. ACM, 2002.

[7] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In M. Weske and P. Liggesmeyer, editors, *Proc. Net.ObjectDays 2004*, volume 3263 of *LNCS*. Springer, 2004.

[8] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proc. of AOSD'04*. ACM Press, 2004.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[10] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings Conference on Aspect-Oriented Software Development (AOSD) '03*, pages 90–99. ACM, 2003.

[11] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings ECOOP '05. to appear*. Springer.

[12] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proc. of AOSD'04*. ACM Press, 2004.

[13] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[14] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, 2004.

Jumping Aspects Revisited

Bruno De Fraine, Wim Vanderperren, Davy Suvée
System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
bdefrain,wvdperre,dsuvee@vub.ac.be

Johan Brichau
Programming Technology Lab (PROG)
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
jbrichau@vub.ac.be

ABSTRACT

In this paper, we propose an extension of the JAsCo aspect-oriented programming language for declaratively specifying a protocol fragment pointcut. The proposed pointcut language is equivalent to a finite state machine. Advices are attached to every transition specified in the pointcut protocol. We claim that stateful aspects benefit from run-time weaving and therefore introduce the JAsCo run-time weaver. When employing this run-time weaver, a stateful aspect is only woven at the join points it is currently interested in. When a state-change occurs, it is rewoven to the new target join points. Hence, a real *jumping aspect* is realized, that literally jumps from join point(s) to join point(s).

Keywords

Aspect-Oriented Software Development, Run-Time Weaving, Stateful Aspects, JAsCo

1. INTRODUCTION

An aspect definition generally consists of two separate parts: the aspect applicability condition (pointcut specification) and the aspect functionality (advice). The aspect applicability condition determines when the aspect's functionality should be invoked. In early aspect-oriented languages, this condition was often expressed in terms of static locations in the base program. However, it was argued early on that conditions in terms of run-time events were more appropriate (e.g. jumping aspects [3], AspectJ's dynamic join point model [19], event-based AOP [10],...).

The importance of dynamic applicability conditions is very well illustrated by a relatively new kind of aspects: *stateful aspects* [8]. These aspects describe their applicability in terms of a sequence of run-time events. The true dynamic nature of their applicability condition even implies that these aspects can benefit from a *run-time weaver* that enables aspects to 'jump' in a very literal sense. This concept is illustrated in the context of JAsCo [28], which is a dynamic aspect language with support for the definition of stateful aspects, and which provides a run-time weaver for dynamically adding and removing aspect behavior.

The following section describes how stateful aspects benefit from a run-time weaver. Section 3 describes stateful aspects in JAsCo, together with an analysis of the drawbacks of static weaving. Section 4 describes run-time weaving in JAsCo and weaving of stateful aspects using run-time weaving. Section 5 describes related work and section 6 concludes the paper and discusses future work.

2. STATEFUL ASPECTS BENEFIT FROM RUN-TIME WEAVING

Stateful aspects [8] are aspects that are triggered by the occurrence of a consecutive sequence of events. They are considered stateful because their applicability condition needs to consider the notion of *state* to keep track of the (past) sequence of events. In contrast, the applicability condition of traditional aspects is stateless because the aspect applies on *all* the events that it matches. For example, consider a stateful logging aspect that only requires to write data to a log if a user is logged in. The applicability condition of this stateful logging aspect can be described in terms of a sequence of events. The first event is the execution of the `login()` method. Subsequent events trigger the aspect's advice that writes data to a log until the execution of the `logout()` method, which terminates the applicability of the stateful logging aspect. Another example, which is used in this paper, is a simple publish/subscribe collaboration, implemented using an aspect. In such a stateful publish/subscribe aspect, the aspect only publishes to a subscriber *after* it has actually subscribed, i.e. the publish advice is only applicable after the execution of the `subscribe()` method.

Some AOP languages, such as JAsCo [28], provide linguistic support for the description of stateful aspects. In JAsCo, the aspect applicability condition of stateful aspects is described by means of a declaratively specified *protocol* that describes the desired sequence of events. An implementation of such a protocol in JAsCo is shown later on, in section 3.2. Although stateful aspects could be simulated using traditional aspects by keeping an explicit state variable in the aspect code, such a manual implementation is a cumbersome and error-prone task. More importantly, it involves tangling the aspect-applicability mechanism with the aspect's functionality inside of the advice, which is undesirable. Providing linguistic support for the implementation of stateful aspects frees the programmer from the burden of writing the state bookkeeping code, which allows for a trivial and clean implementation of stateful aspects.

However, the weaving of stateful aspects using a static weaver results in a significant performance overhead, as code needs to be woven at any possible join point that is considered by the stateful aspect. Although a stateful aspect might be interested in many different join points in its lifetime, only a limited set of join points are applicable at a particular point in time. Hence, the stateful aspect is only interested in the occurrence of a particular subset of events, i.e. those events that are applicable given the aspect's state. Consequently, the presence of woven code at those join points that are not applicable only induces a performance overhead. Static weaving of stateful aspects also results in a severe limitation in expressiveness because the possible join points need to be known at compile-time. Some stateful aspects however require to compute the applicable join points at run-time. For instance, reconsider the case of the simple publish/subscribe protocol. In some cases, a decision about the events that should be published can only be taken upon subscribing (i.e. at run-time). This means that the aspect can only be woven at the required join points after the subscription event.

To resolve these problems, we propose to weave stateful aspects at run-time. Instead of statically weaving code at any possible join point that might be required for the execution of the stateful aspect, the aspect is only woven at the applicable join points at run-time. In essence, when a stateful aspect changes state (because an event expressed in the protocol has occurred), the run-time weaver unweaves the code at those join points in which the aspect is no longer interested and reweaves it at the appropriate join points. This realizes a real *jumping aspect* that literally jumps from join point(s) to join point(s). As a result, no unnecessary woven code is left at inapplicable join points. Furthermore, a run-time weaver provides the opportunity to determine the join points at run-time. Also notice that when the aspect is no longer applicable, it is completely unwoven and thus does not cause a performance overhead any longer. Previous work by Costanza [6] also motivates that aspects should be able to vanish.

Run-time weaving thus provides a natural and efficient technique for the realization of stateful aspects. Of course, run-time weaving itself also causes a significant performance overhead. Nevertheless, for stateful aspects that do not require frequent reweaving (i.e. do not change state very often), the performance gain from the absence of woven code at unnecessary join points can be far greater than the performance loss induced by run-time weaving.

3. JASCO LANGUAGE

3.1 Introduction

The JAsCo [28] AOP approach is an aspect-oriented extension for Java that allows for a clean modularization of crosscutting concerns. The JAsCo language tries to stay as close as possible to the original Java syntax and concepts, and introduces only two new entities, namely *Aspect Beans* and *Connectors*. An aspect bean is an extended version of

```
1  class PublishManager {
2
3    // Bookkeeping/notification code
4    void addListener(MethodListener ml) { ... }
5    void notifyListeners(String methodname, Object[] args) { ... }
6
7    hook Publish {
8      Publish(topublish(..args)) {
9        execute(topublish);
10     }
11
12     after() {
13       notifyListeners(thisJoinPoint.getMethodName(), args);
14     }
15   }
16 }
```

Figure 1: A JAsCo aspect bean for simple publish behavior

a regular Java Bean that allows describing crosscutting concerns independently of concrete component types and APIs. JAsCo connectors on the other hand are used for deploying one or more reusable aspect beans within a concrete component context and provides support for describing their mutual interactions.

The JAsCo language is illustrated by implementing the simple publication behavior that was mentioned in the previous section. Suppose the execution of certain methods of a component should be published to interested listeners. It should however be avoided to tangle the components logic with code that manages this publication system. JAsCo allows to specify this behavior as a reusable aspect bean that is illustrated in figure 1. Note that an aspect bean looks very similar to a regular Java Bean, and likewise implements a number of ordinary Java class members. In addition, one or more hook definitions that implement the crosscutting behavior can be specified. For example, the `PublishManager` aspect bean contains a number of standard methods to manage and notify the listeners (lines 3-5), and a `Publish` hook (lines 7-15) that is responsible for invoking the notification after the execution of a relevant method. A hook has one or more constructors that specify in an abstract way *when* the behavior should be triggered, and one or more advice methods (`before`, `around`, `after` ...) that specify *what* this behavior consists of. In this case, the constructor (lines 8-10) outlines that the hook behavior applies to the execution of the abstract method `topublish`. The `after()` advice method (lines 12-14) then specifies that, after this event, the listeners should be notified of the method name and the arguments.

JAsCo's abstract and reusable aspect beans are deployed onto a concrete component context by making use of connectors. Each connector allows to explicitly instantiate and

```
static connector PublishUpdates {

  PublishManager.Publish publish =
    new PublishManager.Publish(void ComponentX.update*(*));

  publish.after();

}
```

Figure 2: A JAsCo connector for the publishing of updates

initialize one or more logically related hooks. Figure 2 illustrates a connector that instantiates the `Publish` hook of figure 1 onto the update methods of the `ComponentX` component. This is realized by passing these methods as a wildcard to the hook constructor (lines 3-4). A connector additionally allows to select and order the behavior methods of the instantiated hooks. In this case, it is specified that the `after()` advice method of the `Publish` hook should be executed whenever a join point of this hook is encountered (line 6). As a result of the declarations in the connector, the `Publish` hook is applied to the update methods of `ComponentX`, and as such, registered listeners will be notified after the execution of these methods.

Now consider an extension of the `PublishManager` aspect bean where this publication only occurs *after* a certain subscribe method has been executed. To capture this behavior using traditional (stateless) aspect facilities, we have to manually implement code that maintains a state regarding this condition. This approach is illustrated in figure 3, where the extended `ConditionalPublishManager` aspect bean is presented. It contains two hooks: the new `Subscribe` hook is responsible for setting the `subscribed` instance variable to true after the execution of an abstract subscribe method, and the `ConditionalPublish` hook extends the `Publish` hook to make the publishing behavior apply only when the state variable `subscribed` is true. In the specification of these hooks, the usage of the `isApplicable()` hook method is crucial. This JAsCo language construct allows to describe a run-time condition for a hook as the advices are only executed when the body of this method evaluates to true (similar to the `if` pointcut construct in AspectJ). For the first hook, the `isApplicable()` method in line 10 specifies that the hook should not execute when the `subscribed` variable already has a true value. For the second hook, the method in line 18 specifies that the publishing behavior should only apply when the `subscribed` variable reflects a subscribed state. As no other elements of the `Publish` hook are modified, they can be inherited as such.

Although it is possible to implement the desired functionality using only stateless aspect facilities, this is quite a cumbersome and error-prone task, since it requires to capture each state in a separate hook and involves adding code to maintain variables regarding this state. In the next section, a stateful extension to the JAsCo language is presented which solves these problems by allowing the developer to declaratively specify a protocol of expected pointcuts.

```
1  class ConditionalPublishManager extends PublishManager {
2
3    boolean subscribed = false;
4
5    hook Subscribe {
6      Subscribe(subscribe(..args)) {
7        execute(subscribe);
8      }
9
10     isApplicable() { return !subscribed; }
11
12     after() {
13       subscribed = true;
14     }
15   }
16
17   hook ConditionalPublish extends Publish {
18     isApplicable() { return subscribed; }
19   }
20 }
```

Figure 3: An extended JAsCo aspect bean for conditional publish behavior

3.2 Stateful Aspects Language

Mainstream aspect-oriented approaches rarely support protocol history conditions. In many cases, it is only possible to refer to previous join points when they still have an activation record on the stack (i.e. using the `cflow()` keyword in AspectJ). In order to solve this limitation, Douence et al. [8] propose a formal model for aspects with general protocol based triggering conditions, named *stateful aspects*. In this section, we illustrate how the JAsCo language is extended with stateful pointcut expressions, based on this formal model.

```
1  class StatefulPublishManager extends PublishManager {
2
3    hook PublishSubscribe {
4
5      PublishSubscribe(subscribe(..args),
6                         topublish(..args)) {
7
8            Waiting: execute(subscribe) > Publish;
9            Publish: execute(topublish) > Publish;
10     }
11
12     after Publish() {
13       notifyListeners(thisJoinPoint.getMethodName(), args);
14     }
15   }
16 }
```

Figure 4: A JAsCo stateful aspect bean for the simple subscribe/publish protocol

To illustrate the JAsCo stateful aspects syntax, reconsider the simple publish-subscribe protocol from the previous sections. Only when a subscription event occurred, the aspect should start publishing. Figure 4 illustrates how this protocol can be declaratively described by making use of the JAsCo stateful aspect language. The constructor of the stateful hook `PublishSubscribe` (line 5-10) describes a protocol-based pointcut expression. Every line in the constructor defines a new transition within the protocol. Each transition is labeled with a name (e.g. `Waiting`), defines a JAsCo pointcut expression (e.g. `execute(subscribe)`) and specifies one or more destination transitions that are matched after the current transition is fired. A transition fires when its pointcut expression evaluates to true. For example, the `Waiting` transition only fires whenever the concrete method(s) bound to the abstract method parameter `subscribe` are executed. In that case, transition `Publish` is activated and will be evaluated for the subsequent join points encountered during the application's execution.

A stateful aspect always starts by evaluating the first defined transition. As a result, a protocol `subscribe-topublish` is described. In between the fired transitions, other join points can also be encountered. As such, a sequence of events `methodY-subscribe-methodX-topublish` is also a valid instance for the defined protocol and will trigger the associated transitions.

On every transition defined in the stateful constructor, advices can be attached which are executed whenever the transition is fired. For example, the `after Publish` advice (line 12-14) is only triggered whenever the transition `Publish` is fired. In other words, the advice is executed whenever the concrete method(s) bound to the abstract method parameter `topublish` are executed in that state of the stateful aspect. To sum up, the stateful `PublishSubscribe` hook will only start notifying interested subscribers when a subscription event occurred.

Figure 5 illustrates how the stateful aspect of figure 4 is instantiated and deployed using a JAsCo connector. This example is similar to the connector of figure 2 as it bounds the abstract method `topublish` to the update methods of `ComponentX`. Additionally, the `subscribe` abstract method parameter is bound to a concrete subscription method of a certain `PSComponent`. Consequently, as soon as this subscription method has been executed, the aspect will start intercepting update methods on `ComponentX` and will start notifying its registered listener(s).

```
static connector PSConnector {
  StatefulPublishManager.PublishSubscribe ps =
    new StatefulPublishManager.PublishSubscribe(
      boolean PSComponent.subscribe(),
      void ComponentX.update*(*)
    );
}
```

Figure 5: The JAsCo connector for deploying the stateful `PublishSubscribe` hook.

3.3 Advanced Language Features

In addition to attaching advices on each transition separately, it is also possible to describe global advices that are triggered for all fired transitions. In this case, the advice is specified as usual, but the transition label is omitted. It is also possible to attach a specific `isApplicable` method to a particular transition in the protocol. Hence, the transition will only fire when both the pointcut expression and the `isApplicable` condition evaluate to true. Likewise to advices, a global `isApplicable` condition can be specified which is applied to all transitions. In that case, transitions are only fired when they satisfy their pointcut expression and both the global and local `isApplicable` conditions. The following code fragment shows both a global and local `isApplicable` condition.

```
isApplicable() {
 // global condition for all transitions
}
isApplicable XTrans() {
  // local condition only relevant for the transition XTrans
}
```

The JAsCo stateful aspects constructor can also specify multiple destination transitions for a given transition. The syntax is illustrated in the code fragment below. After firing the `XTrans` transition, both the `YTrans` and `QTrans` transitions are evaluated for subsequent encountered join points (line 4). Note that the destination transitions are evaluated in the sequence defined in the destination expression. As such, when both the `YTrans` and `QTrans` transitions are applicable for a given join point, only the `YTrans` transition will be fired and only the `YTrans` destination transitions will be evaluated for subsequent encountered join points. This allows to keep the protocol deterministic and efficient to execute. It is also possible to omit a destination transition for a certain transition. In that case, when the transition fires, no more transitions need to be evaluated and the aspect *vanishes*. This concept is illustrated by the `QTrans` transition (line 6). Also notice that this transition describes a more involved pointcut designator using the `cflow` keyword.

In case the stateful aspect requires to start by evaluating more than one transition, the `start` keyword can be employed. This keyword is followed by a list of starting transitions for matching join points when the aspect is deployed. Multiple start transitions are specified similarly to multiple destination transitions, by using `||` as delimiters. When no start transition is specified, the first defined transition is used as the starting one.

```
1  //starting with two transitions:
2  start > XTrans || QTrans;
3  //two destination transitions:
4  XTrans: execute(methodA) > YTrans || QTrans;
5  //no destination transition:
6  QTrans: execute(methodB) && !cflow(methodC);
7  YTrans: execute(methodC) > YTrans;
```

The syntax proposed in the previous paragraphs provides a way for specifying powerful protocols but might be too tedious in case of simple protocols. Therefore JAsCo also supports a simpler syntax for protocols that do not require multiple destination transitions for a given transition. The following code fragment illustrates a constructor that is equivalent to the constructor of figure 4. Labeling transitions is still possible in order to be able to attach local advices to specific transitions.

```
PublishSubscribe(subscribe(..args), topublish(..args)) {
  execute(subscribe) > Publish: execute(topublish) > Publish;
}
```

The JAsCo stateful aspect language also supports triggering aspects on the opposite (complement) of a protocol. Furthermore, JAsCo stateful aspects are non-strict per default, i.e. they allow non-specified intermediate transitions. Specifying strict protocols is also supported. The discussion of these features is however outside of the scope of this paper. The interested reader is referred to [30, 15] for more information.

3.4 Implementing Stateful Aspects

A naive approach to realize a stateful aspect would be to weave it at all possible join points defined within its protocol. This induces a performance overhead at all these join points, while the stateful aspect is only interested in a limited set of join points corresponding to the subsequent transitions that are to be evaluated. In order to implement stateful aspects more efficiently, a genuine run-time weaver is required which is able to reweave the stateful aspect each time a transition is fired.

Another major problem with statically weaving stateful aspects is that the pointcuts have to be defined in advance.

As argued in section 2, it would be interesting to dynamically decide the concrete join points that have to be used for triggering the subsequent transitions. In that case, a static binding of the abstract method parameters of the hook constructor is not possible and a run-time weaver is necessary for reweaving the stateful aspect after a transition has fired.

To address these shortcomings, we propose to employ a run-time weaver. The following sections introduce the JAsCo run-time weaver and explain how stateful aspects are implemented employing this weaver.

4. TOWARDS RUN-TIME WEAVING

The JAsCo technology was originally *trap*-based. At every join point a trap is inserted that defers execution to the JAsCo run-time infrastructure. This infrastructure will trigger any aspects that apply to the join point or, when no aspects are applicable, it will return to the normal execution. As such, dynamic aspect addition and removal becomes possible because the aspects behavior itself is not statically woven. The JAsCo distribution contains a preprocessor tool that inserts traps at all possible join points before run-time by transforming the necessary classes. These transformations are performed through the byte-code adaptation library Javassist [5]. The main problem of this preprocessing approach is performance. Applications equipped with JAsCo traps execute often more than ten to twenty times slower than the original, which is unacceptable. This overhead stems in part from the naive interception system, namely inserting traps at all possible join points.

In order to improve the performance of JAsCo, the JAsCo HotSwap framework [29] is introduced. HotSwap is a custom-made byte-code instrumentation framework that allows altering the byte-code of a class, even if it is already loaded into the virtual machine. As such, it is possible to install traps just-in-time when a new aspect is added to the system. Likewise, the original method byte-code is re-installed when the aspect is removed and no other aspect are applicable. JAsCo HotSwap has two different implementations, depending on the virtual machine version. For Java 1.4, HotSwap employs the Java Debugging Interface (JDI) to dynamically replace classes. When a 1.5 compatible virtual machine is detected, HotSwap employs the novel Java Programming Language Instrumentation Services (JPLIS) API, which avoids running the virtual machine in debugging mode. Both libraries are standard libraries available in most standalone (i.e. not embedded) virtual machine implementations, making JAsCo perfectly portable over a wide range of platforms. They also make sure that the application is left in a consistent state after byte-code of a class has been replaced. The byte-code manipulations themselves are also performed through the Javassist library.

In principle, JAsCo HotSwap already suffices to efficiently implement stateful aspects as it allows to only insert traps to join points where the stateful aspect is currently interested in. However, by inserting traps that refer to the JAsCo run-time infrastructure, the performance of JAsCo is still not optimal. In several benchmark experiments, the JAsCo advice execution performance is measured [29, 15] to be five to ten times slower than the statically woven language AspectJ. This overhead is mainly caused by the additional indirection these traps impose. In addition, the traps have a fixed implementation for every possible advice attached, so they have to capture all possible relevant run-time information. However, capturing the actual arguments in an array for instance, is a very expensive operation. When the actual arguments are not required in the advices, a substantial performance gain can be realized by avoiding capturing this run-time information.

4.1 The JAsCo Run-Time Weaver

In order to improve the run-time performance of JAsCo AOP, a run-time weaver is proposed. Instead of inserting traps, a highly optimized code fragment is inserted into the target join points. This code fragment directly invokes all applicable advices in the correct sequence and thus avoids the indirection through the JAsCo run-time infrastructure. The JAsCo approach is however a dynamic AOP approach. As such, the woven join point behavior might become invalid. This event occurs when a connector is added that instantiates a hook that is applicable on a join point where aspects are already attached or when a connector is removed that contains an applicable hook for such a join point. In addition, it is possible to change some properties of a connector dynamically so that the applicable context of the instantiated hooks is altered. The JAsCo run-time weaver is able to cope with these issues. When no advices are applicable any longer, the original byte-code of the method is reinstalled.

Generating optimized code for a target join point is not always achievable because some pointcut expressions have to be re-evaluated for every execution of the join point (precisely because of the dynamic join point model). For example, when a hook defines a `cflow` condition in its constructor, this constructor has to be re-evaluated for every execution of a join point. The entire constructor body does not have to be re-evaluated however. In this case, only the result of the `cflow` condition is able to change for different executions of the join point. As such, partial evaluation techniques are used to cache a partially evaluated constructor. In addition, for the particular `cflow` construct, it is sometimes possible to statically analyze whether the condition might ever be true or not by examining the call graph of an application. This technique is elucidated in [26].

Another major optimization of the JAsCo run-time weaver consists of detecting which static and dynamic reflective join point information the aspects might require. Suppose for instance that an aspect only requires the method name of the current join point. In that case, most AOP implementations still capture all actual arguments in an array, which is a very expensive operation, even though they are not required. The JAsCo aspect bean compiler analyzes in detail which contextual join point information is required and stores this information in the compiled aspect so that the run-time weaver can exploit this. If the applicable aspects at a join point only require the method name for instance, only this information is captured. Obviously, because this detection happens at compile-time, it has to be conservative and thus might still capture too much. For example, if a logging advice contains a dynamic test for selecting whether it logs only the method name or also the arguments, the advice is analyzed to require the actual arguments and the method name, while in some cases it only requires the method name. Nevertheless, this analysis allows for a significant optimization in a large number of cases.

The main drawback of the run-time weaver is the increased run-time overhead for adding and removing aspects. In the trapped approach, when a trap is already placed, adding a new aspect does not require any HotSwap overhead whatsoever. Also, even if a new trap has to be inserted, this is a lot less costly than weaving because the code for the trap itself remains constant whereas with run-time weaving, a new code fragment has to be computed for each individual join point. In order to address this overhead, JAsCo is still able to combine the regular preprocessing approach with the run-time weaver and even with the trapped HotSwap approach. Classes that are preprocessed to include traps are never subject to run-time weaving. In addition, it is possible to define a global function that dynamically decides whether a trap is inserted or whether the run-time weaver is employed. This function has the following signature:

```
boolean inlineCompile(JoinPoint jp, Vector hooks)
```

When the method returns true, the run-time weaver is employed, otherwise a trap is inserted. Both reflective information about the join point and the list of applicable hooks are available for deciding whether run-time weaving is appropriate. As such, a heuristic function can be implemented that for example only activates the run-time weaver for join points that are executed more than twenty times in the past second. JAsCo thus effectively combines and integrates three alternative aspect weavers.

4.2 Performance Evaluation

In order to evaluate the performance of the novel JAsCo run-time weaver, we employ the AWBench [17] benchmark[1]. This benchmark is a project of the AspectWerkz team and is especially created to compare the performance of dynamic AOP systems. AWBench is a micro benchmark and consists of 12 tests, all advising a single method in a different way. Every test is executed two million times and the average execution time of the method is recorded. When a certain test is not directly supported by the AOP approach, it is simulated using the best available alternative (e.g. when no *after throwing* advice is available, it is simulated using around advice). We compared the performance of JAsCo with the following AOP approaches: AspectJ 1.2 [19], JBoss/AOP 1.0 [16], AspectWerkz 2.0 [18], Spring/AOP 1.1.1 [27], dynaop 1.0 Beta [11] and cglib 2.0.2 [4]. The next paragraph shortly introduces the technologies employed in each of these approaches. Notice that this selection is not meant as a comprehensive overview of dynamic AOP approaches. Nevertheless it includes a significant portion of the practically used dynamic AOP systems.

AspectJ and AspectWerkz both use a traditional weaver that invasively weaves the aspects into the target classes at run-time. Similar to JAsCo, AspectWerkz also features a genuine run-time weaver while AspectJ is limited to compile-time weaving. JBoss/AOP uses an approach similar to the original JAsCo technology, namely inserting traps to all advised join points. In contrary to JAsCo, the traps are installed at load-time and can never be removed. As such, at join points where no traps are attached, dynamic aspect interference is impossible. Spring/AOP and dynaop are two proxy-based approaches that employ the Java Dynamic Proxies feature to dynamically attach advices to objects. Dynamic Proxies are instance-based, so it is easily possible to only advice one object of a certain class. The main drawback however is that class-based aspect application is more difficult to realize. In addition, Dynamic Proxies induce a relatively high performance overhead. cglib (Code Generation Library) is not an AOP framework per se, but a byte-code adaptation library with extensive AOP features.

Figure 6 on page 7 illustrates the results of running the AWBench with the introduced approaches. The performance of JAsCo using the trapped approach to attach aspects is also measured. Notice the logarithmic scale of the results to fit all results in one clear chart. In all benchmarks the three approaches that use weaving (JAsCo, AspectJ, AspectWerkz) perform significantly better than the others. In the most simple before advice for example, JAsCo executes more than a hundred times faster than Spring/AOP. The trapped approaches (JBoss/AOP and JAsCo no-RTW) perform worse than weaving but still execute considerably faster than the proxy-based approaches (Spring/AOP and dynaop).

For the before advices where the run-time context is fetched declaratively, the three woven approaches perform equally well. All three optimize the join point interception to only fetch that data that is requested. When reflection is used however, JAsCo is able to improve on both AspectWerkz and AspectJ. This is because JAsCo has a fine-grained required context detection, also when it is reflectively queried. `thisJoinPoint` vs. `thisJoinPointStaticPart` is the only difference accounted for by AspectJ and AspectWerkz. When `thisJoinPoint` is employed, all possible run-time context information (target object and type, caller object and type, actual arguments and types, etc...) is stored, whereas only a fraction of this dynamic information might be effectively required.

In addition, when several advices are combined or when an around advice is employed JAsCo seems to improve more significantly on AspectJ and certainly on AspectWerkz. In all the other tests JAsCo, AspectJ and AspectWerkz are very close. As such, it seems that the performance of compile-time and run-time weaving approaches converges and probably a boundary of traditional weaving has been reached. In any case, the run-time performance of JAsCo has been improved quite considerably when comparing it with the trapped approach and thus the goal of the run-time weaver has been accomplished.

4.3 Implementing Stateful Aspects using the Run-Time Weaver

The previous paragraphs motivate and explain the JAsCo run-time weaver. As explained before, stateful aspects also particularly benefit from run-time weaving. A naive approach for integrating a stateful aspect would be weaving it at all possible join points defined within the protocol. This induces a performance overhead at all these join points, while the stateful aspect is only interested in a limited set of

[1]The AWBench distribution including JAsCo can be downloaded here: http://ssel.vub.ac.be/jasco/awbench

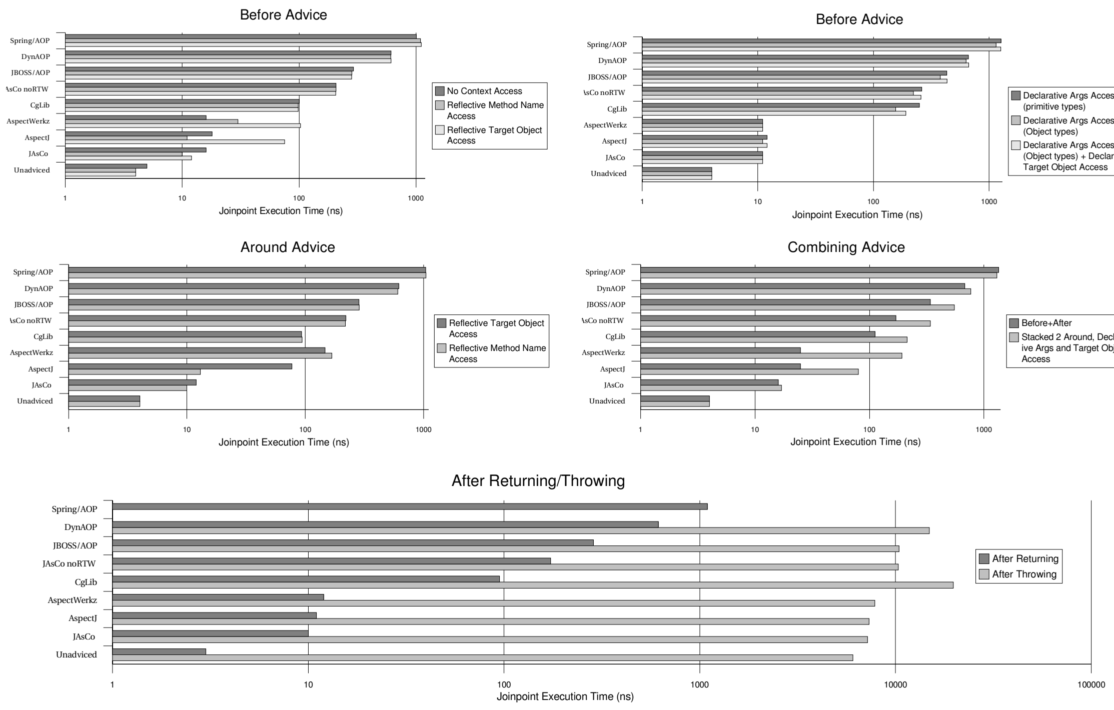


Figure 6: AWBench benchmark results run on a PENTIUM4, 2GHZ, 256 RAM with Ubuntu Linux 4.10, Java 1.5.0 update 1. Notice the logarithmic scale of the timings in order to keep every result readable.

join points corresponding to the subsequent transitions that are to be evaluated. By employing the run-time weaver, it is possible to only weave the stateful aspect at those join points where the aspect is currently interested in. When a transition is fired, the weaver unweaves the aspect at the join points associated with the current transition and weaves it back in at the join points relevant for the subsequent transitions. As such, a real *jumping aspect* is realized. Notice that when the aspect *vanishes* because no subsequent transitions are defined, it is completely unwoven. As a result, no performance overhead for the aspect is endured any longer.

The weaving process itself does however also require a significant overhead. Therefore, when a given protocol is encountered many times in a short time interval, it might be more efficient to weave the aspect at all possible join points of the protocol instead of weaving and unweaving it on-the-fly. This can be configured in JAsCo by using the novel Java 1.5 annotations (meta-data). When the `@WeaveAll` annotation is supplied to the hook, as illustrated by the code fragment below, the run-time weaver weaves the aspect at all join points and never unweaves it unless the aspect itself is manually removed or vanishes.

```
@jasco.runtime.aspect.WeaveAll
hook StatefulHook { ...
```

In order to implement the stateful pointcut itself, the pointcut is translated to a Deterministic Final Automaton (DFA) [13]. The JAsCo stateful aspects language is equivalent to a DFA because every expression defines one DFA transition, two DFA states and possibly several connection DFA transitions for the destinations. Therefore, the JAsCo compiler compiles a stateful aspect constructor to a DFA that is interpreted at run-time. Every transition of a DFA contains a representation of the pointcut definition and possibly an `isApplicable` condition. When a join point is encountered, the outgoing transitions of the current state are evaluated with the given join point and when a match is encountered, the state machine moves to the destination state. When this event occurs, all associated advices are executed and the aspect is rewoven to the new join points corresponding to the outgoing transitions of the destination state. Because of this implementation strategy, a stateful aspect can be executed very efficiently. It suffices to check only the transitions of the current state, as JAsCo stateful aspect protocols are regular and can be interpreted by a regular DFA. When non-regular protocols are allowed, a history of all relevant encountered events should be maintained, which is very expensive.

5. RELATED WORK

The emerging stateful aspect research is still quite young and at the moment not many AOP approaches support its ideas and concepts. Douence et al. are the first ones to propose an extension of their formal aspect model [7], to support stateful aspects [8]. The advantage of having a formal model is that it allows to automatically deduce possible malicious interactions among aspects. Furthermore, the model supports the composition of stateful aspects using well-defined composition operators. A proof of concept implementation of this model, based on static program transformations, is available [9]. JAsCo improves upon this implementation, as only a subset of join points needs to be woven, whereas a static approach requires to weave all possible join points defined within the protocol.

Walker et al. introduce *declarative event patterns* (DEPs) [31] as a means to specify protocols as patterns of multiple events. Here, AspectJ aspects are augmented with special DEP constructs that can be advised. Their approach is based on context-free grammars, and involves a transformation of the DEP constructs into regular AspectJ aspects that contain an event parser. While DEPs can recognize properly nested events and thus possess an even higher degree of declarative expressibility than the JAsCo approach, they only provide the ability to attach advice code to the entire protocol. Separate transitions of the protocol can as such not be advised, and several overlapping protocols are required to mimic JAsCo stateful aspect behavior. Furthermore, the fact that DEPs lose their identity in a preprocessing step that reduces them to standard aspects, rules out the possibility for optimizations by a weaver that analyzes the feasible transitions of the protocol.

Finally, Masuhara et al. [20] propose an extension of the AspectJ pointcut language to identify join points which are based on the data flow of values within an application. Their novel *dflow* pointcut designator allows to declaratively specify that a particular join point can only match if its arguments are originating from the arguments/return value of a previously encountered join point. By explicitly declaring this preferred data flow, this mechanism allows specifying a more precise pointcut then possible using the current AspectJ pointcut designator language. Although a data flow aspect is not completely similar to a stateful aspect, this research illustrates the need for a mechanism that allows the specification of aspect behavior defined in terms of the history of previously encountered join points. It should however be mentioned that JAsCo stateful aspect are also able to capture data flow pointcuts. This however requires a programmatic approach, which is not as declarative as the approach proposed by Masuhara et al.

Apart from the dynamic AOP technologies employed during our performance evaluation, several other AOP approaches have been introduced for enabling dynamic AOP. Many of these approaches make use of traps and a corresponding registry infrastructure for dynamically (un)weaving aspects. Event-based aspect oriented programming (EAOP) for instance, allows specifying crosscutting concerns by employing event patterns which are described using a formal language [10]. On the implementation level, EAOP inserts traps that query a central execution monitor that has a global view of the executing application and which contains all active EAOP artifacts. In contrast to JAsCo however, EAOP inserts these traps by employing source-code transformations, which obstruct performance optimizations. JAC [21] also make use of traps. Here, these traps are automatically inserted at load-time of the application by making use of byte-code transformations. The Dynamic Aspect-Oriented Platform (DAOP) [22] is an approach that targets legacy component-based systems and allows flexible application of aspects at run-time. DAOP introduces a distributed platform, where a middleware layer is employed for storing the aspect composition information. DAOP does not require any component adaptation and allows aspects to remain first-class entities at run-time.

PROSE [24] and Wool [25] both employ the Java Virtual Machine Debugging Interface (JVMDI) for intercepting the program's execution. A dedicated execution monitor is deployed on top of the JVMDI, which allows capturing the relevant execution events. Whenever an event is encountered where an aspect is applied upon, the corresponding aspect behavior is triggered. Wool improves upon PROSE, as it also allows to invasively insert join points. In addition, aspects are able to implement their own heuristics for deciding whether they should be invasively inserted or not. The Wool heuristics improve on JAsCo as they can be customized on a per-aspect basis whereas in JAsCo only one global heuristics function can be specified.

PROSE2 [23] and Steamloom [2] both aim at achieving an aspect-aware Java Virtual Machine in order to boost the run-time performance of AOP. PROSE2 proposes a next-generation implementation for the original PROSE approach, this by incorporating the execution monitor for joint points into the virtual machine itself. This execution monitor is then responsible for notifying the AOP engine which executes the corresponding advices. Steamloom is implemented as an extension of IBM's Jikes Research Virtual Machine [14] and employs its adaptive optimization system for decorating the base application with aspects. Similar to JAsCo, this mechanism allows for the structure-preserving compilation of reusable aspects which are explicitly deployed at run-time. The main advantage over a run-time weaver is that it avoids the weaving overhead, which makes it very suitable when aspects are deployed and removed frequently.

Filman [12] finally proposes dynamic injectors in order to introduce aspects within an application. These dynamic injectors are incorporated into the OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. Dynamic injectors are first-class objects that can be added and adapted at run-time. At the implementation level, a wrapping approach is employed for injecting the logic of an aspect within a component communication channel.

6. CONCLUSIONS AND FUTURE WORK

This paper presents an extension of the JAsCo language that allows to declaratively specify a regular protocol fragment pointcut. Advices can be attached to each transition in the protocol. Furthermore, we present the JAsCo run-time weaver, of which the run-time performance is able to compete with current state-of-the-art AOP. In this paper we claim that stateful aspects benefit from run-time weaving because 1) they can be executed faster and 2) dynamic pointcut introduction becomes possible. By implementing stateful aspects using the JAsCo run-time weaver, genuine jumping aspects are realized that jump from join point(s) to join point(s) depending on the state of the aspect. It is even possible that the aspect vanishes when no new join points are defined for a certain state.

A limitation of the current stateful aspects language is that it only supports regular protocols. Protocols that require a non-regular language (like for example `n times A; B; n times C`, where `n` can be a different number in every occurance of the protocol), cannot be represented. For instance, in order to enhance the example of figure 4 with an unsubscribe transition so that the aspect is unwoven when no subscribers are present, a non-regular protocol has to be used because the aspect has to wait for an equal amount of unsubscriptions as subscriptions before it can be unwoven. The advantage of keeping the protocols regular is that they can be efficiently evaluated using a DFA. A naive implementation of a non-regular protocol would require to keep the complete history of all encountered join points in memory, which is not very practical. In literature, several domain-specific optimization techniques for interpreting non-regular languages have been proposed [1]. Extending the JAsCo stateful aspects language to non-regular protocols while still allowing an efficient implementation is subject for future work.

Another intesting area for future work consists of developing heuristics for deciding whether the run-time weaver has to be used or whether the trapped approach is desired. As such, the performance of JAsCo-enabled applications can be automatically tweaked. For long running applications, it could be even possible to exploit learning strategies in order to learn the most optimal heuristic.

7. ACKNOWLEDGEMENTS

Bruno De Fraine and Davy Suvée are supported by a doctoral scholarship from the Institute for the promotion of Innovation by Science and Technology in Flanders in the Industry (IWT).

8. REFERENCES

[1] J. Aycock and N. Horspool. Schrodinger's token. *Software Practice and Experience*, 31(8), 2001.

[2] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proceedings of AOSD*, Lancaster, UK, Mar. 2004.

[3] J. Brichau, W. D. Meuter, and K. D. Volder. Jumping aspects. In *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

[4] cglib. *cglib Project.* http://cglib.sourceforge.net/.

[5] S. Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP, LNCS*, Cannes, France, July 2000.

[6] P. Costanza. Vanishing aspects. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[7] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of GPCE*, Pittsburgh, USA, Oct. 2002.

[8] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of AOSD*, Lancaster, UK, Mar. 2004.

[9] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. *Aspect-Oriented Software Development*, Sept. 2004.

[10] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of REFLECTION*, Kyoto, Japan, Sept. 2001.

[11] dynaop. *dynaop Project.* http://dynaop.dev.java.net/.

[12] R. Filman. Applying aspect-oriented programming to intelligent systems. In *Position paper at the ECOOP 2000 workshop on Aspects and Dimensions of Concerns*, Cannes, France, June 2000.

[13] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory.* Addison Wesley, 2st edition, 2001.

[14] IBM. *The Jikes Research Virtual Machine.* http://www-124.ibm.com/developerworks/oss/jikesrvm.

[15] JAsCo. *JAsCo Distribution Website.* http://ssel.vub.ac.be/jasco.

[16] JBoss Inc. *JBoss/AOP Project.* http://www.jboss.org/developers/projects/jboss/aop.

[17] Jonas Bonér and Alexandre Vasseur. *AspectWerkz AWBench Project.* http://docs.codehaus.org/display/AW/AOP+Benchmark.

[18] Jonas Bonér and Alexandre Vasseur. *AspectWerkz Project.* http://aspectwerkz.codehaus.org/.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP*, Budapest, Hungary, June 2001.

[20] H. Masuhara and K. Kawauchi. Dataflow pointcuts in aspect-oriented programming. In *Proceedings of APLAS*, Bejing, China, Nov. 2003.

[21] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of REFLECTION*, Kyoto, Japan, Sept. 2001.

[22] M. Pinto, L. Fuentes, M. Fayad, and J. Troya. Separation of coordination in a dynamic aspect oriented framework. In *Proceedings of AOSD*, Enschede, The Netherlands, Apr. 2002.

[23] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *Proceedings of AOSD*, Boston, USA, Mar. 2003.

[24] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of AOSD*, Enschede, The Netherlands, Apr. 2002.

[25] Y. Sato, S. Chiba, and T. Michiaki. A selective, just-in-time aspect weaver. In *Proceedings of GPCE*, Erfurt, Germany, Sept. 2003.

[26] D. Serini and O. D. Moor. Static analysis of aspects. In *Proceedings of AOSD*, Boston, USA, Mar. 2003.

[27] Spring. *Spring/AOP Project.* http://www.springframework.org/.

[28] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component-based software development. In *Proceedings of AOSD*, Boston, USA, Mar. 2003.

[29] W. Vanderperren and D. Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *Proceedings of Dynamic Aspects Workshop*, Lancaster, UK, Mar. 2004.

[30] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful aspects in JAsCo. In *Submitted to SC 2005, LNCS*, Edinburgh, Scotland, Apr. 2005.

[31] R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, USA, Nov. 2004.

Contextual Pointcut Expressions for Dynamic Service Customization

Thomas Cottenier
Illinois Institute of Technology
cottho@iit.edu

Tzilla Elrad
Illinois Institute of Technology
elrad@iit.edu

ABSTRACT

In service-oriented environments, components are discovered and integrated at runtime. The type of the client entities that expose potentially interesting contexts can therefore not be anticipated, and can not be subject to type-based pointcut expressions.

This paper proposes a contextual pointcut construct that attempts to address context passing mechanisms needed for concurrent customization of services.

Contextual pointcut expressions generalize the semantics of 'cflow' to enable advices to retrieve a richer set of context information along the call path to a target joinpoint. A context visitor collects information about the particular circumstances in which a target joinpoint is executed, including references to the joinpoints encountered along the path.

Contextual advices have the power to alter the control flow of the aspect execution, and return control to the joinpoints that are still alive in the target joinpoint's call path.

1. INTRODUCTION

The thoughts expressed in this position paper draw from the authors' experiences in applying AOP concepts and languages constructs to the development of a distributed aspect platform for dynamic and distributed service composition and customization

The particularities of distributed environments, and service-oriented environments in particular, force distributed aspect platforms to take a different perspective on context passing pointcut expressions and joinpoint composition.

Remote pointcuts [1] allow us to locally specify events of interest occurring on remote hosts. When the pointcut is triggered, the metadata of the joinpoint is passed to the local host and the corresponding advice is executed locally. Context passing from the joinpoint to the remote advice is part of the remote pointcut semantics.

However, more advanced context passing scenarios are not straightforward to support in a distributed setting.

In AspectJ, context passing from a client down through the calls that lead to a target service can be implemented as shown in Figure 1.

This type of context passing addresses the concurrent customization of services according to the context of a client entity. Concurrent service customization [2] means that a same service instance can be customized in one execution context without impacting clients interacting with the service instance in other contexts.

However, in service-oriented environments, components are discovered and integrated at runtime. The type of the client entities that expose interesting contexts can therefore not be anticipated, and can not be subject to type-based pointcut expressions. Tough, the contexts that are of potential use for service customization are matched by:

```
cflow( call(* *(..)) && target(obj))
                && execution(* Service.f(..))
```

This pointcut expression is triggered at each method call encountered from the root to the call to the Service.f() method. However, only the last client context encountered is exposed by the joinpoint. Each client context is overridden by the next one.

To tackle context-dependent dynamic service composition and customization, we propose to generalize this construction, so that the context of all encountered entities on the path from the entity that initiates the interaction to the final caller is potentially available at the service side. We therefore attach a visitor to the cflow pointcut that accumulates the encountered context references instead of overriding them.

The paper is structured as follow. Section 2 briefly introduces Contextual Aspect-Sensitive Services and its context passing construct. Section 3 illustrates the need for a generalization of 'cflow' through a distributed advice chaining example. Section 4 introduces contextual pointcut expressions and section 5 discusses dynamic service customization. Finally, section 6 concludes this paper.

```
pointcut myServiceCall(Client client): cflow( call(* *(..)) && target(client) );
pointcut myServiceExec(Service service): execution(* Service.f(..)) && this(service);
pointcut myServiceContext(Client client, Service service): myMethodCall(client) &&
                                                                myMethodExec(service);
```

Fig.1. Caller context passing for a remote pointcut

2. SERVICE-ORIENTED POINTCUTS

The Contextual Aspect-Sensitive Service (CASS) [3,4,5] platform proposes a new technique to dynamically compose Web Services in a decentralized manner, by deploying SOAP message interceptors at the boundaries of Web Services. Remote pointcuts are declared with respect of the port types defined in the service WSDL definitions, to maintain platform independence.

CASS enables crosscutting and context-dependent behavior to be factored out of the Service implementations and modularized into separate units of encapsulation that are exposed as Web Services. Service orchestrations can then be defined in a much more flexible way, as services can be dynamically customized to address changing business rules or context-dependent requirements. Because CASS weaves the coordination logic directly at the level of web service interfaces, support for a ‘perCflow’ type of aspect construct is fundamental in order to provide the capability to specify the context under which a given orchestration should be activated.

CASS provides joinpoint and advice callback interfaces to deal with asynchronous message based interactions. These callback interfaces allows multiple advices bound to a shared joinpoint to be executed concurrently and also enable synchronization primitives to be embedded into pointcut expressions, using the ‘join’ pointcut composition operator. A pointcut that is composed of ‘joined’ pointcut expressions is triggered only once each one of the expression has been triggered in a specific context. Asynchronous joinpoints can handle not being returned control to, when their callback methods are never invoked.

In CASS, messages are intercepted at the joinpoints specified by a ‘cflow’ expression and wrapped into an envelope which contains the cflow expression, as well as the joinpoint instance reference. The system ensures the propagation of this information from caller to callee, and within a same service instance execution context.

The following specification shows how ‘cflow’ is implemented CASS. The effect of the ‘cflow’ aspect is to wrap calls to the ‘MathService’ web service into an envelope that contains the current execution context of the call.

The ‘inContext’ pointcut will only match the execution of an operation on ‘MathService’ if it is called in the context of the ‘cflow.context’ pointcut.

The smooth the progress of the discussion, a notation derived from AspectJ/AspectWerkz will be used in the following sections, instead of the Cass specification.

Next sections motivate the need for a generalization of ‘cflow’ that enables advices to retrieve a richer set of context information along the call path to a target joinpoint.

3. DISTRIBUTED ADVICE SEQUENCES

When defining a chain of advice in a distributed environment, the advices of the chain might very well execute on different hosts. In order to minimize unnecessary use of network resources, it is desirable to support direct chaining from one advice host to another, instead of letting the remote joinpoint dispatch the advice calls.

The example depicted in Figure 3 represents a simple B2B choreography. A billing service and a credit service perform some processing before and after an order is processed on a supplier service. Both the billing service and the credit service can force the transaction to rollback.

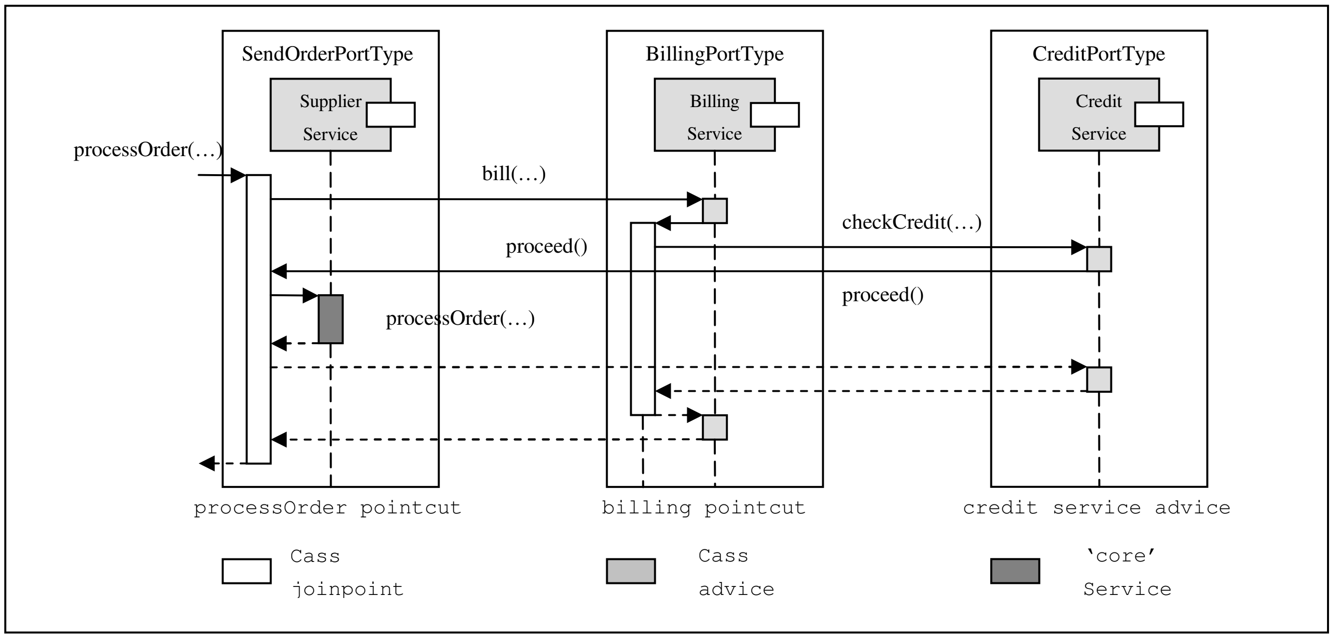
A chain of advices around the execution of the ‘processOrder’ method on the supplier service can be decomposed by declaring a pointcut expression on the call to the ‘proceed’ method of the joinpoint the ‘BillingService’ advice is bound to.

Instead of returning control to the billing joinpoint, the credit service advice returns control to the ‘orderProcessing’ joinpoint, that’s higher on the call stack. This example illustrates how advices can return control to other joinpoints then the one that initially triggers them.

While this construct allows to effectively implementing chain of advices in a distributed setting, it forces the developer to pass along the root joinpoint reference from joinpoint to joinpoint, until the last advice.

```
<cass name="cflow">
   <pointcut name="context"
            service="edu/iit/concur/cass/testservices/MathService"
            operation="add"
            type="client"/>
   <advice name="contextPropagationAdvice"
            type="around"
            bind-to="context"
            host="http://localhost:8081"
            service="edu/iit/concur/cass/testservices/MathService"
            operation="add"/>
</cass>
<cass>
   <pointcut name="inContext"
            host="http://localhost:8081"
            service="edu/iit/concur/cass/testservices/MathService"
            operation="*"
            context="cflow.context"/>
</cass>
```

Fig.2. Cass specification of context pointcut



```
pointcut joinpoint_proceed(JoinPoint jp): call(* proceed()) && target(jp);
pointcut billing(JoinPoint jp): within(BillingService) && joinpoint_proceed(jp):
void around (JoinPoint jp) : billing(jp){
        checkCredit();
        jp.proceed();
        logTransaction();
}
```

Fig.3. Distributed sequence of advice on a shared joinpoint

This implementation is therefore very brittle and not reusable as the 'CreditService' advice implementation should be independent on whether it is used in an advice chain or not.

In order to pass the root joinpoint context to the advice chain, we propose to generalize the semantics of cflow to enable advices to retrieve a richer set of context information along the call path to a target joinpoint. Indeed, the joinpoints encountered in the chain are successively in the control flow of each other.

4. CONTEXTUAL POINTCUT EXPRESSIONS

Contextual pointcuts follow the same triggering rules as cflow pointcuts, but their joinpoints can accumulate context information during their life cycle. The context lives from the point the corresponding cflow expression evaluated to true until the system exits from the control flow.

Contextual pointcuts are defined by programmatically specifying a joinpoint context visitor. The joinpoint visitor is part of the pointcut definition rather than the advice definition.

We here adopt a notation that is closer to that of AspectWerkz. The arguments and return value of an advice are exposed by accessing the 'JoinPoint' parameter.

A visitor implements the 'Context' interface, which defines a 'visit' method and a 'proceed' method.

The 'visit' method is called by the runtime system each time a joinpoint is triggered in the control flow of the contextual pointcut the visitor is bound to.

The 'proceed' method allows an advice to return control, not only to the joinpoint that triggered it, but also to the other joinpoints encountered along that path.

The code sample of figure 4 defines a visitor for sequential chaining of advices in a distributed setting. On proceed, control is returned to the root joinpoint.

The advice sequence can now be defined as depicted in figure 5.

The visitor of figure 6 corresponds to the regular cflow behavior: each context is successively overridden by the next one.

```
pointcut processOrder = "execution(* SupplierService.processOrder(..))";
pointcut chain = "call(* Context.proceed()) && chainVisitor(processOrder)";

class chainVisitor implements Context{
        JoinPoint cjp;
        int i=0;
        public void visit(JoinPoint jp){
                if(i==0)
                        cjp=jp;
        }
        public Object proceed(){
                return cjp.proceed();
        }
}
```

Fig.4. Contextual Visitor for Distributed sequence of advice on a shared joinpoint

```
@Around("processOrder && chain")
public void bill(Context context){
        checkBillingInfo(…);
        context.proceed();
        doBilling(…);
}

@Around("within(BillingService) && chain")
public void checkCredit(Context context){
        checkCredit();
        context.proceed();
        logTransaction();
}
```

Fig.5. Distributed sequence of advice with contextual pointcut.

```
class cflowVisitor implements Context{
        JoinPoint cjp;
        public void visit(JoinPoint jp){
                cjp = jp;
        }
        public Object proceed(){
                return cjp.proceed();
        }
}
```

Fig.6. Contextual Visitor implementation of cflow

5. DYNAMIC SERVICE CUSTOMIZATION

In many cases, nothing is known about the type of the client entities of a service. There is then not other alternative than to expose the entire interaction context:

```
pointcut exposecallpath: context(call(* *.*(..)));
pointcut computation: exposecallpath &&
                    execution(* S.do_computation(…));
```

This kind of context propagation is however not scalable, and requires the context consolidation logic to be woven into all entities that are potentially in the path from the client to the service. There is therefore a need for mechanisms to discriminate entities on the base of the context they may expose. As this decision needs to be taken before the logic is woven into the entities, it needs to be expressed at the pointcut level.

In service-oriented environments, services can expose additional information about themselves in the form of Service Data. Service Data is set of structured data that describes static and dynamic properties of web service instance.

Figure 7 presents an example of a contextual visitor that uses service data to discriminate service contexts.

In the example, the visitor logic needs to be woven only at the interface of services that expose the 'ReliabilityServiceDataElement'.

Properties of this nature should therefore somehow be expressed at the pointcut level. Semantic Web techniques might be a way to achieve this goal.

5. CONCLUSIONS

This position paper proposes to generalize the semantics of 'cflow' to enable advices to retrieve a richer set of context information along the call path to a target joinpoint.

Contextual visitors collect information about the particular execution circumstances of a target joinpoint, including references to the joinpoints encountered along the path.

```
class ComputationContext extends Context{
  List reliabilityData = new ArrayList();
  List joinPoints = new ArrayList();
  JoinPoint lastjp;
  public void visit(JoinPoint jp){
    ServiceData sd = jp.getThis().
                      getServiceData();
    if(sd.contains("ReliabilitySDE")){
      reliabilityData.
       add(sd.get("ReliabilitySDE"));
      joinPoints.add(jp);
      lastjp = jp;
    }
  }
  public Object proceed(){
    if(computeReliability() > 0.95)
       return lastjp.proceed();
       ((JoinPoint)jps.get(0)).signal(
              new ReliabilityException()
       );
    }
    private float computeReliability(){ …}
}
```

Fig.7. Discrimination of context based on service data

When control is returned to the target joinpoint, contextual advices have the power to alter the control flow of the aspect execution, and return control to other joinpoints that are still alive in the target joinpoint's call path.

REFERENCES

[1] M. Nishizawa, S. Chiba, M. Tatsubori, Remote Pointcuts – A language Construct for Distributed AOP, In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development,* Lancaster, UK, March 2004

[2] Eddy Truyen, Dynamic and Context-Sensitive Composition in Distributed Systems, PhD Thesis, October 2004.

[3] T. Cottenier, T. Elrad. Validation of Aspect-Oriented Adaptations to Components, *Ninth International Workshop on Component-Oriented Programming as part of ECOOP'04*, Oslo, Norway, June 2004.

[4] T. Cottenier, T.Elrad. Layers of Collaboration Aspects for Pervasive Computing, in proceedings of the *5th Argentine Symposium in Software Engineering* (ASSE'2004), Cordoba, Argentine, September 2004

[5] T. Cottenier, T. Elrad. Contextual Aspect Sensitive Services www.iit.edu/~concur/asc

Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation

Stephan Herrmann Christine Hundt* Katharina Mehner* Jan Wloka*
Technical University of Berlin Fraunhofer FIRST
{stephan,resix,mehner}@cs.tu-berlin.de jan.wloka@first.fhg.de

ABSTRACT

Many aspect-oriented programming languages employ static transformations in order to produce the executable system. Some aspects, however, should only be effective if certain conditions are fulfilled that can only be evaluated at runtime. The naïve approach of using conditionals within the advice code easily leads to scattering and tangling regarding these conditionals, suggesting that they should be separated from the advice code. In this paper we analyze how aspects can be made conditional, i.e. how their effect can be controlled based on runtime values.

We present an extension to the ObjectTeams/Java programming language that provides a flexible means for controlling the instantiation and activation of an aspect implemented by a "team" and its roles by means of *guard predicates*. We discuss different points in a program for which we consider a guard predicate suitable. We argue that this approach is more general than the ways other aspect languages control aspect instantiation and activation. Our approach makes use of the fact that in Object Teams aspects are implemented as first-class objects, which have full support for inheritance and polymorphism.

1. MOTIVATION

The goal of AOP is to reduce tangling and scattering. AOP has successfully contributed to separating orthogonal and crosscutting code into aspect modules. Besides statically defined and deployed aspects, dynamism of aspects increasingly attracts researchers' interest. In some situations, it is desirable to invoke or change aspect behavior based on the dynamics of program execution.

In previous work, we have identified three dimensions to classify dynamism of aspects [14].

- The dimension of aspect *dynamism* addresses conceptually different life-cycles for aspects. Here we distinguish (a) static and dynamic definition of aspects, (b) static definitions of aspects which can be added and removed at runtime, and (c) static and dynamic activation of aspects.
- The dimension of *adaptation scope* distinguishes between type level and instance level control, assuming that aspects can be instantiated like classes in object-oriented languages.
- The dimension of *weaving* distinguishes between different points in time for the technical weaving process ranging from pre-compile time weaving to runtime weaving.

*This work has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

To give programmers maximal control over aspect dynamism it is desirable that these dimensions be clearly distinguished in language support. Therefore, we examine how this can be achieved in the context of the core language elements which have already been identified for AOP: *advice* (or comparable elements) containing the actual code of the crosscutting concern, and *pointcuts* for matching events in the execution of a program.

Existing AOP approaches support the above dimensions to different extents and in different ways. We observe deficiencies especially related to the concepts of dynamic activation control and instance level activation.

In approaches with static weaving and without explicit support for dynamic activation nor instance specific activation, these concepts have to be simulated using conditionals such as `if`-statements. Hence, advice code becomes tangled with code for activation control. This is especially dissatisfying because AOP made a start in avoiding one of the principal maintenance nightmares: nested conditionals spreading all over the code, resulting in programs that are more concerned with deciding what to do than with doing.

Approaches which have taken a step further support for separation of concerns include specialized support for controlling instantiation and activation of aspects. In AspectJ [16] keywords like `perthis` and `pertarget` allow to control aspect instantiation, but only a limited set of pre-defined strategies is available. Arbitrary querying of state is possible with the `if`-pointcut [16], but here pointcuts tangle the interceptions of events with activation control.

We feel that the separation of conditionals and actions can be taken one step further by elevating the conditions used to control different kinds of activation to a prominent language feature. Between join points and the action attached to them, this paper suggests a third concept: *guard predicates*. The main contribution of this paper is to report on an experiment integrating guard predicates into the aspect-oriented programming language ObjectTeams/Java.

Relating AOP to more fundamental concepts of programming should help to identify a general system of coordinates for describing AOP languages, which should eventually lead to a common theory of AOP. Guard predicates are a candidate for such a fundamental concept.

After providing explicit support for guard predicates, other parts of AOP languages — most notably join point languages — can be stripped down and will eventually reveal those atomic concepts that are new in AOP.

The paper is structured as follows: Sect. 2 gives a brief introduction to the language ObjectTeams/Java. Sect. 3 motivates the need for more flexible control over aspect activation. Sect. 4.1 explains how we integrate guard predicates into ObjectTeams/Java for controlling aspect activation. Sect. 4.2 relates guard predicates to issues of instantiation. Sect. 5 discusses the results in the context of related work. In Sect. 6 we conclude and outline some future work.

2. AN OVERVIEW OF OBJECT TEAMS

Before introducing the integration of guard predicates into ObjectTeams/Java, we give a brief overview of this language (see also [10, 20]). Object Teams introduces two new kinds of classes: `teams` and `roles`. A role is used to decorate an existing class, called its *base*. *Teams* are a structural concept for encapsulating collaborating roles. An instance of a team encapsulates a set of role instances.

A role instance intercepts method calls to the decorated object if a *callin* method binding between a role method and one or more base methods has been specified. A callin can be of type *before*, *after*, or *replace* (similar to an advice weave as found in AspectJ [16]) and inserts additional behavior into the control flow. A role together with its enclosing team can therefore be seen as an aspect.

Callin bindings of a role are only effective if the enclosing team has been explicitly activated. When a callin-bound base method is called the base object is "translated" to the corresponding role. The translation is called *lifting* [12]. Next, the bound role method is called. The lifting translation must be aware of any existing role-base pairs. To this end, every team internally has a role registry for storing and retrieving roles. If no appropriate role is found it is created on demand and stored in the registry.

2.1 Example

The following source code illustrates the concepts *team*, *role* and *callin bindings* by the example of an automatic teller machine (ATM). We start with a most simple `Account` class and step-by-step introduce new requirements which are to be added non-invasively.

```
1  public class Account {
2    private int balance;
3    public boolean debit(int amount) {
4      if (!(amount>balance)) {
5        balance -= amount;
6        return true;
7      }
8      return false;
9    }
10   ...
11 }
```

An `Account` is a regular (base) class offering the functionality of `debit`ing an amount (among other things).

```
12 public team class ATM {
13   public int payCash(
14         Account account, int amount)
15   {
16     boolean ok = account.debit(amount);
17     if (ok) return amount;
18       else return 0;
19   }
20   // role definition:
21   protected class FeeAccount playedBy Account {
22     callin boolean debitWithFee(int amount) {
23       int fee = calculateFee(amount);
24       return base.debitWithFee(fee+amount);
25     }
26     // replace callin binding:
27     debitWithFee <- replace debit;
28     int calculateFee(int amount) {...}
29   }
30 }
```

The `ATM` allows to withdraw money via the method `payCash(..)`[1]. So far only standard Java features have been used. However, the `team` modifier in line 12 denotes that this class is realized as a *team* class. An additional requirement to be fulfilled is to collect a fee when withdrawing money. This is realized by the *role* `FeeAccount` which decorates the base class `Account`. Inner classes of teams are per definition roles. The decoration relationship is stated by the `playedBy` keyword (see line 21).

Via a replace callin binding the role method `debitWithFee` is dedicated to intercept the base method `debit` (see line 27). The effect of this binding is the following: Whenever `Account.debit` is called the target is lifted to a `FeeAccount` role. The control flow is redirected to its `debitWithFee` method, which calculates the fee and calls the original base method with the increased amount via `base.debitWithFee`. In analogy to super calls, a base call uses the name and signature of the enclosing role method.[2] This *base call* is comparable to the *proceed* in AspectJ.

3. MORE CONTROL OVER ASPECTS

So far, the introduced concepts of Object Teams allow general control over the activity of aspects by activating and deactivating teams and all contained roles. In many cases, this may be enough, but in more complex applications a more flexible control will be needed.

Reconsidering the ATM example, it is questionable if the general collection of fees is fair. The role-base binding stated by the `playedBy`-clause generally attaches the fee-collecting role to every `Account` (base-)object. Actually, an ATM belongs to a certain bank and only for withdrawal from a foreign account an additional fee is debited. Let's assume also, that it should not be possible to query the balance of a foreign account.

[1]Authorization needs are omitted for simplicity.
[2]This syntax is further motivated by the goal to keep the callin method independent of any bindings to (possibly multiple) base methods.

In the following, a more realistic behavior for an ATM is achieved. The addition of a `Bank` attribute to an `Account` as well as to an `ATM` is straightforward. More importantly, in order to fine-tune the general behavior, we need a more flexible control over the activity of the aspect encapsulated in the `FeeAccount` role. So far, this is only possible with explicit hand-coded if-statements in the advice code. The following source code only shows the relevant additions compared to the first version[3]:

```
31 public class Account {
32   private Bank myBank;
33   public Bank getBank() { return myBank; }
34   ...
35 }

37 public team class ATM {
38   private Bank bank;
39   public int payCash(Account ac, int am) {...}
40   public int getBalance(Account account) {...}

42   public class ForeignAccount playedBy Account
43   {
44     callin void debitWithFee(int amount) {
45       if (ATM.this.bank.equals(getBank())) {
46         base.debitWithFee(amount);
47       } else {
48         int fee = calculateFee(amount);
49         base.debitWithFee(fee+amount));
50       }
51     }
52     callin int checkedGetBalance() {
53       if (ATM.this.bank.equals(getBank())) {
54         return base.checkedGetBalance();
55       } else {
56         throw new AccException("foreign account");
57       }
58     }
59     debitWithFee      <- replace debit;
60     checkedGetBalance <- replace getBalance;
61     abstract Bank getBank();
62     getBank           -> getBank;
63   }
64 }
```

The new requirement of restricting the query of balance, is realized as another callin method denying the call to `getBalance` for foreign accounts (see line 52). In line 45 and line 53 if-statements are used to ensure the desired behavior. If the callin methods are called for own accounts they just forward to the original base method (see lines 46 and 54). Otherwise, they perform the special behavior of foreign accounts. Lines 61 and 62 are used to define and bind the method `ForeignAccount.getBank` via *callout*[4] to the corresponding base method, which is needed in the conditions.

Although the required behavior is now achieved some bad smells can be observed. First of all, the condition is *tangled* with the advice code. Secondly, we see a *scattering* of the (identical) condition over the two role methods. What we really want to state is that *the role `ForeignAccount` is only valid for accounts belonging to a different bank than the ATM.*

We need a more flexible control over the activity of aspects and we want to get rid of the drawbacks observed in the hand-coded version. The following section shows how the integration of guard predicates can achieve this goal in a more elegant way.

4. GUARD PREDICATES IN OBJECT-TEAMS/JAVA

In the previous section we have illustrated the problem of condition tangling as it pertains even if some form of AOP is applied. In order to remedy this problem we go back to fundamental concepts of programming and try to identify ECA elements *("event-condition-action")* in AOP:

- We interpret join points as places in a program that emit *events.*
- We consider advice weaving as an adaptation of the *action* that happens at join points.
- The contribution of this paper is the prominent role which *conditions* play as a middleman between events and actions.

In order to distinguish conditions in this sense from conditions in the imperative part of the language we will use the notion of *guard predicates.* In slight adaptation of existing ECA formalisms, we define that a guard predicate is to determine whether the firing of a given event should actually cause the adapted action to be performed. Otherwise the original behavior is performed unchanged.

In this section, we will first introduce the places where guards can be attached to specific entities of an ObjectTeams/Java program (4.1) and show how aspect activation can be controlled using guards. Also aspect instantiation can be controlled by guards as we present in Sect. 4.2. We will then relate guards to other concepts in the language (Sect. 4.3).

4.1 Controlling aspect activation

Callin-bindings as described in the previous section cause the control flow to leave a base object and enter the corresponding role object. In our model, guard predicates are used to filter these added control flows. The effect of a guard evaluating to `false` is that the current callin trigger is rejected and the base behavior is performed unmodified. If a guard evaluates to `true` (or if no guard is present) the callin trigger is effective and the adaptation defined by the corresponding role is applied. We say, an aspect (team and its roles) is active if its callin bindings are effective. Thus, guard predicates are a means to control aspect activation.

We support guard predicates at four levels of granularity. The first level refers to method bindings. This is a straight forward adoption of tests for dynamic properties as they are supported, e.g., by the `if` pointcut designator in AspectJ. Semantically, guards of method bindings do not differ from the `if` designator, we just syntactically separate the guard to be evaluated at runtime from the statically computed set of join points.

[3]Note, that the role is now called ForeignAccount, because this name better states its meaning.
[4]*Callout* bindings declaratively specify the forwarding from a role method to a base method. Reverse to callin bindings, which are denoted with the symbol "`<-`", callout bindings use the symbol "`->`".

Here is the syntax of a guarded method binding (details of the `when` clause will be discussed below):

```
protected class MyRole playedBy MyBase {
  ...
  /* callin method binding with guard */
  void rmeth(int x) <- after void bmeth(int y)
       when (boolean expr. using this(refers to role) and x);
}
```

As a first means for generalization we also support guards attached to a method. In ObjectTeams/Java the action to be performed at a join point is defined by regular methods[5]. This way several method bindings may bind to the same role method. Attaching a guard predicate to a role method has the semantics of attaching the guard to all method bindings of this method. Only if a method is invoked due to a callin binding, guards are evaluated and may reject the current trigger.

In the next level of granularity guards are attached to role classes such as in the header of the following role class:

```
protected class MyRole playedBy MyBase
    when(boolean expression using this(refers to role))
{
    //class body omitted
}
```

In this case the idea is to enable or disable a role instance based on the evaluation of a guard predicate. The guard is evaluated before any callin bindings to an instance of the current role class are considered.

Finally, by attaching a guard predicate to a team class, all role classes of the team are consistently enabled or disabled based on the evaluation of the guard predicate.

The following table summarizes the locations where guard predicates can be attached and describes the respective effect. Note that guard predicates are only checked if a method is called via callin.

location/level	**affected role methods**
role method binding	this role method if called due to this binding
role method	this role method
role	all role methods of this role
team	all role methods of all roles in this team

4.1.1 Scope of guard predicates

First of all, all predicates may access the current team instance, because only with an existing instance of the team, any callin bindings are considered at runtime. Within a team level guard, the team instance is simply referred to as `this`. In practical examples this proves very useful, since a team instance can be used to store arbitrary context information that is valuable for evaluating the predicate. We will further discuss the role of teams for *context based programming* in Sect. 5.4.

[5]Only for `replace` callin bindings a special kind of method is required that is marked by the `callin` modifier.

All guards at the level of role classes or below also have access to the role instance whose method is about to be invoked. Also the role instance can be used to store context or history information, that will help to formulate the predicate. In a role level guard, `this` refers to the role instance and the notation *`MyTeam`*`.this` to the enclosing team. Guards of methods and method bindings also expose parameter values (except for the shorthand variant of callin bindings, which may omit method signatures).

With a role level guard predicate the ATM example from section 3 can now be rewritten as follows:

```
65 public team class ATM {
66   ...
67   public class ForeignAccount
68     playedBy Account
69     when (!(ATM.this.bank.equals(getBank())
70   {
71     callin void debitWithFee(int amount) {
72       int fee = calculateFee(amount);
73       base.debitWithFee(fee+amount));
74     }
75     callin int checkedGetBalance() {
76       throw new AccException("foreign account");
77     }
78     debitWithFee      <- replace debit;
79     checkedGetBalance <- replace getBalance;
80     abstract Bank getBank();
81     getBank           -> getBank;
82   }
83 }
```

The guard predicate attached to the `ForeignAccount` role in line 69 ensures that this role and therefore all its callin bindings are only effective for foreign accounts. This way, all previously necessary partial checks are substituted by one specification, expressing exactly our intended requirement. Also a clean separation of advice code and activation conditions is achieved.

4.2 Controlling aspect instantiation

In most AOP languages, instantiation of aspects is fairly different from instantiation of regular objects. Commonly, aspects are instantiated automatically when needed without using a `new` expression. Aspect instances are yet important because they allow to store state of the aspect. Technically, aspect behavior is usually implemented as instance methods of the aspect, with the only exception of "advice" which lacks some important properties of methods.

Aspect instantiation attaches an aspect instance (in Object Teams: role object) to one or more base objects. The Object Teams approach pays very close attention to providing all benefits of instantiation also to aspect related language features. Therefore, actual role objects are used instead of augmenting base classes with additional features. In this context, role instantiation is such an essential concept that the used strategy should not be hard-coded into the language as a fixed set of options. Rather should programmers have the chance to effectively control the strategy of role creation. Different from regular objects, the *default* for role objects is an implicit creation on demand. However, client code may force the creation of additional roles and may prevent the default creation of roles.

4.2.1 *Positive control*

Before we demonstrate how guards may prevent role creation, let us briefly review the means ObjectTeams/Java provides for triggering role creation.

The default constructor of each *bound*[6] role class expects as its only parameter an instance of the connected base class. This constructor can be used from within the context of a team. The language implementation ensures that role objects created using this constructor are also registered in the team for consistent integration with the mechanism of lifting.

As a second option, methods of a team may declare an argument with a dual type: the caller is required to pass a base object yet the method body receives a role object. The signature for this declaration looks as follows: `teamMethod(MyBase as MyRole r)`. When calling such a method, on demand creation of roles remains implicit, however, the existence of a role object for the given base object is ensured upon entry to this method.

4.2.2 *Negative control*

Whenever a callin binding is about to invoke a role method, first the base object is lifted to its corresponding role object. If that role object is not yet present in the team's registry, a fresh role is automatically created. The guards we have presented so far reside at the role side of a binding. Thus, lifting happens before evaluation of the predicate. This has the advantage, that information in a pre-existing role object can be used within the predicate.

By attaching the modifier `base` to any guard predicate, the semantics is changed such that the predicate is evaluated at the base side, i.e.: prior to lifting. If a base guard evaluates to `false` no lifting occurs and the aspect has no effect at all, not even creating a role.

It is important to see, that lifting potentially has side effects. First, creation of a role is an implicit side effect, affecting the team's registry. Second, a custom role constructor may implement arbitrary side effects. Only a base guard ensures, that in the negative case no side effect has been caused. A future version of the compiler for ObjectTeams/Java will also include an analysis, whether guard predicates are in fact free of side effects.[7] Syntactically, a guard predicate may invoke any boolean method in scope.

As an example, consider again the case of a predicate attached to a method binding:

```
protected class MyRole playedBy MyBase {
  ...
  /* callin method binding with base guard */
  void rmeth(int x) <- after void bmeth(int y)
       // optional parameter mapping omitted
       base when (boolean expression using base and y);
}
```

Please note that the scope of a base predicate is at the base side. The base object at which the join point trigger occurred can be accessed using the special identifier `base`. Method parameters available in the predicate are those of the base method (here: `y`). In anticipation of the control flow into an existing team instance, the team causing the callin interception can be accessed using `MyTeam.this`.

[6]A role with a `playedBy` clause is *bound*.

[7]The need for side effect analysis fits nicely with other work on Object Teams where a concept of `readonly` interfaces is used to enforce representation encapsulation [11].

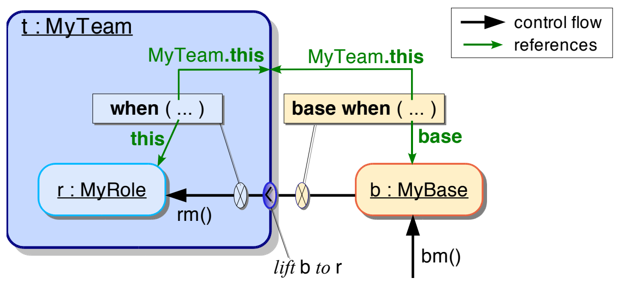


Figure 1: Characteristics of Guards in Object Teams

Figure 1 shows which objects a predicate can access, independently from the level of granularity. A call to `MyBase` intercepted by `MyRole` is controlled by a guard predicate. A `when` predicate can refer to the involved role via `this`, while a `base when` has access to the corresponding base object via `base`.

Now it is possible to further enhance the ATM example by controlling the instantiation of the `ForeignAccount` role. If an account belongs to the same bank as the ATM, it will never need the extra functionality of the `ForeignAccount` role. Using a base side guard prevents the creation of roles for such accounts, thus preventing all possible side effects.

```
84 public team class ATM {
85   ...
86   public class ForeignAccount
87     playedBy Account
88     base when (
89          !(ATM.this.bank.equals(base.getBank())
90   {
91     callin void debitWithFee(int amount) {...}
92     callin int checkedGetBalance() {...}
93     ...
94   }
95 }
```

Line 89 shows the mentioned `base when` predicate. We now have direct access to the `Account.getBank()` method through `base` in the guard predicate and do no longer need the indirect access via callout.

4.3 Relating guards to other concepts

The idea of guard predicates originates from the fundamental issue of controlling the flow of execution. The previous section has shown how guards in ObjectTeams/Java can be used to control also the instantiation of role objects where the default behavior is otherwise an automatic instantiation per team and per base object. In this section we investigate the relationship of guards to other fundamental concepts of (object-oriented) programming.

4.3.1 *Guard combinations*

When introducing predicates as a new language feature, rather than relying on regular methods and method calls only, care must be taken, not to break polymorphism and dynamic dispatch. More specifically, predicates should be able to exploit the inheritance structure of roles and base classes.

In some situations more than one guard watches over the callin to a single role method. On the one hand this can occur if a role predicate is inherited and overridden in the subclass. On the other hand a second dimension of combination appears if guards at different levels of granularity affect the same role method. In both cases a reasonable strategy for combining these predicates is needed.

Guards at different positions of the inheritance hierarchy:
The semantics of object-oriented inheritance is to refine the behavior in subclasses. In analogy to this, guards in a subclasses are combined by logical `and` with the guards of its superclass. Thus, the subclass can only further refine an inherited guard predicate.

Guards at different levels of granularity:
If looking at a team-level guard predicate, the programmer should be allowed to assume that this predicate is effective for the whole team. For reasons of locality guard predicates of inner entities may not invalidate outer ones. This means that inner guard predicates may only refine the conditions of outer ones. Thus, they are again combined by a logical `and` with all outer predicates.

The case of *polymorphism* only seems to be more difficult when considering base guards. Here, no role instance exists yet, thus dynamic dispatch to the most suitable role class seems impossible. However, guard predicates are integrated into the language in such a way, that dispatching can indeed exploit all kinds of polymorphism involved. A detailed description of this integration would require to dive into the mechanism of *smart lifting* as defined in [12], which establishes a new level of polymorphism called *translation polymorphism*. Such details are, however, beyond the scope of this paper. For the programmer, the situation could be interpreted as a role instance that exists *virtually*, whereas the point in time of evaluating the base guard is early enough to prevent the role object from materializing.

4.3.2 Reflection

Object Teams also aims at supporting reflection with respect to teams, roles, and role-base relationships. As mentioned before, each team instance internally has a registry of known role objects indexed by their base object. Programmers may make use of this registry using the following reflective methods defined in `org.objectteams.Team`[8]. The last one of these methods does not access the registry but can be used to inspect whether a control flow has already been intercepted by at least one callin binding.

```
boolean hasRole (Object aBase);
boolean hasRole (Object aBase,
                 Class  expectedRole);
Object  getRole (Object aBase,
                 Class  expectedRole);
void    unregisterRole(Object aRole);
boolean isExecutingCallin();
```

[8]The predefined super class of all team classes

It is desirable and possible to use these methods within guards. These methods allow to write the specification of guards in a more concise and more expressive way. Determined by the signature, the first three methods can only be used in a base-level guard because they require a reference to a base object.

4.3.3 Example using reflection

We now conclude the introduction of guard predicates in Object Teams with an example which summarizes several concepts and illustrates the use of reflective predicates.

We want to extend the account example as follows: If an account is registered to collect a special bonus, every time an amount of more than 1000 is deposited, additional 1% of the amount is credited. To this end, we create another team `SpecialConditions` which is responsible for the new functionality.

```
 96 public class Account {
 97   private int balance;
 98   public void credit(int amount) {
 99     balance += amount;
100   }
101   ...
102 }

104 public team class SpecialConditions {
105   public void participate
106               (Account as BonusAccount ba) {}

108   public class BonusAccount
109     playedBy Account
110     base when(SpecialConditions.this.hasRole(
111                       base, BonusAccount.class))
112   {
113     public void creditBonus(int amount)
114       when (amount > 1000)
115     {
116       base.creditBonus(amount+(amount/100));
117     }
118     creditBonus <- replace credit;
119   }
120   ...
121 }
```

This team provides a registration method `participate` which is used to register an account for the special conditions (see line 105). Here, the explicit role creation mechanism as described in 4.2.1 is used.

The base side guard predicate in line 110 checks, if the base object already has a role in this team. If this is not the case it prevents lifting (and thus role creation). In combination with the registration method this means that `BonusAccount` roles are never created automatically via lifting but have to be explicitly registered first. This is exactly what we want for our bonus collection.

The callin method in line 113 implements the collection of the bonus. It replaces the original `Account.credit` method (see line 118) and performs a base call with the increased amount of money. In line 114 we use an additional predicate to ensure that bonus is only credited for amounts greater than 1000.

The following code snippet shows a usage example. Only the account a1 is registered for the bonus collection, while a2 does not have any special conditions.

```
... // main method:
Account a1 = new Account();
Account a2 = new Account();
SpecialConditions sc = new SpecialConditions();
sc.activate();
sc.participate(a1);
a1.credit(2000); // -> balance += 2020
a2.credit(2000); // -> balance += 2000
...
```

5. COMPARISON TO RELATED WORK

There is a long tradition in specifying program behavior using triplets of events, conditions and actions (ECA). This has been made popular by the statecharts notation [9]. In this context the fundamentally different nature of events and conditions has been elaborated in depth. This suggests that also programming languages might benefit from a clear separation of pure events that happen at a specific point in time versus conditions reasoning about state that persists over a span of time.

The idea of ECA triplets has been taken up in the active database context [5]. Typically, the events, also called triggers, are changes in the database such as an insertion into a table. The condition evaluation determines whether a certain action will be executed or not.

5.1 Event-based programming

A similarity of ECA systems and AOP systems has been discussed in [4]. Also *event-based AOP* (EAOP) has been proposed ([6]). Here, aspects are defined in terms of events emitted during program execution. Identifying each hit of a join-point with an event that can trigger further behavior is well in line with the common understanding that AOP opens new options for reasoning about points in the control flow of a program. However, the EAOP approach does not provide explicit support for conditions, leaving this to the imperative part of the language. This leads to the tangling of conditions and actions, which motivated the introduction of guards predicates.

5.2 Predicates in programming

The expressiveness of predicates has been exploited for object-oriented programs by the concept of *predicate dispatch* [3, 19].

Predicate Classes were first introduced as an extension of the programming language Cecil [3]. They complement normal classes. An object is automatically an instance of a predicate class whenever it satisfies a predicate expression associated with the predicate class. Any of these predicate classes can override a method of that object if it formulates more specific conditions in a predicate. Consequently, method look-up depends not only on the dynamic types but also on dynamic object states (as captured by predicates). In this approach, ill-defined predicates could lead to situations where several methods or no method at all will be selected during method look-up producing fatal run-time errors. To prevent such errors, programmers have to follow specific disciplines.

Recently, an extension of Java with *predicate-based dispatch* called JPred has also been presented [19]. In this approach the problem of ambiguity and incompleteness of predicates is addressed by modular type-checking using a theorem prover.

Both approaches have the benefit that they avoid large switch statements and instance-of checks in individual methods, which are also difficult to maintain. Instead, each method is responsible for one case only, and the most specific method according to the most specific predicate is chosen. Such a system is easier to extend by new cases as each concern is encapsulated in a method. This benefit has also been identified by [22].

While these approaches improve the separation of concerns and flexibility, at the same time they introduce a complex analysis problem that either has to be dealt with by the programmer or by employing tool support based on formal methods. The question remains whether this is always tractable for the programmer.

By contrast, guard predicates in ObjectTeams/Java do not select *necessary* methods for execution, but only filter whether *additional* aspect behavior will be triggered or not (before/after). If `replace` bindings are involved, semantic consistency is of course subject to the implementation of the replacing callin method, but this problem is intrinsic to replace/around aspects and even to regular method overriding.[9] In our approach each explicit method call (base level) will always select exactly one method. In *no* situation the evaluation of guards will result in a run-time exception signaling the inability to dispatch the method call.

Although this type safety in ObjectTeams/Java is achieved by constraining the original idea of predicate dispatch, a role class can be used in quite similar ways as a predicate class. The difference is, that a base object does not completely change its type based on its state, but it may acquire additional roles which can augment its state and adapt its behavior.

5.3 Guards in Aspect-Oriented Programming

The *Rondo* model [18] defines aspect–like elements called *adjustments*. In Rondo adjustments have an event and/or a condition. Events are explicitly sent using a `raise(...)` primitive. When the event associated with an adjustment is fired, the adjustment is added to a given base module (another adjustment). While this model has an explicit distinction between events and conditions, both parts are optional. Also a strategy for evaluating conditions on demand is included. Unfortunately, this model has not been followed up after its definition.

AspectS [13] supports *activation blocks* by which method wrappers can be configured to filter trigger events based on program state. The examples seem to imply that activation blocks are designed for the purpose of defining join-points from a set of pre-defined activation blocks. While the concept is actually quite similar to guard predicates in Object Teams, AspectS supports activation blocks only at the level

[9]In ObjectTeams/Java, flow analysis is used to warn about replace callin methods, that do not issue exactly one base call on each control flow.

of method wrappers, thus leaving activation at the level of larger program units to the discipline of the programmer[10].

In the *Composition Filters* approach [2], filters are used to achieve method interception. Filters can accept certain messages and thus intercept their execution. The matching process is based on the target object, the message selector and an optional *condition*. Conditions are reusable side-effect free boolean methods, which offer an abstraction of the state of the implementation object. They allow the separation of the filter itself from the implementation details of the object. Thereby the reusability of the filter is increased.

The successful application of predicates in the Composition Filters approach suggests that predicates are indeed a desirable concept for AOP languages. In Composition Filters, filters are very specific entities, existing for the sole purpose of filtering messages. Conditions serve the purpose of controlling the flow of execution. By contrast, team and roles in ObjectTeams/Java are regular classes producing regular instances.

We could not find prior work concerning AOP where predicates where used to control aspect instantiation as it is supported by `base` predicates in ObjectTeams/Java.

5.4 Context based programming

Teams define a context for all executions of role methods. It is an advantage that this context is available already when evaluating guards. When acting as a mediator between its role objects, a team can be used to detect certain execution patterns at the base level. Mediated by a team, a sequence of callin triggers can be regarded as a *situation*, which includes a memory of passed triggers as well as a place for storing data that where collected along the path of execution. In this setting, guards are used to filter events depending on the current state of the team and on any data available at the join points. Thus, Object Teams supports different levels of designing program *modes*: Team activation defines which team instances influence the system at a given point in time. Guards allow an active team to focus on specific subsets of emitted events. This could mean to consider certain base objects only, it can exclude method calls based on their parameter values, or different role classes can be turned on and off in different modes of a team.

Other approaches exist that allow to define behavior based on some context information. Lasagne [15, 24] defines per-client contexts for distributed applications. Also, context relations [23] and environmental acquisition [7] share some ideas with Object Teams. Caesar [17] and Chameleon [8] are other programming models similar to Object Teams. Of these approaches, only Lasagne supports a limited notion of predicate dispatch. In Lasagne a deployment-specific interceptor can be used for "global predicate dispatch". On the other extreme, methods may be guarded by a precondition for so-called "local predicate dispatch". An in-depth comparison of our approach with the details of the recently published thesis [24] will certainly be very instructive, but from a first look it seems, that predicates in Lasagne cannot be used to control aspect instantiation.

It is a contribution of our approach to provide guards at four different levels of granularity and at different points in the control flow (at the base vs. at the role). Also among the presented approaches the additional dispatch mechanism of Object Teams (lifting) is unique. By their close connection to the lifting mechanism, guards can also be used to control role instantiation, which is not parallelled in the other approaches.

6. CONCLUSION

We have introduced guard predicates as a clean way to control the activation and instantiation of aspects. The feature has been implemented in the OTDT, the eclipse based Object Teams Development Tooling[21]. First experiments support the predicted benefits.

We are convinced that explicit support for guard predicates in aspect languages helps to untangle language concepts, thus leading to more orthogonal language designs. In this sense, the introduction of guards as a separate construct allows to construct a leaner join point language. It was an important step of AOP to extract many conditionals from the imperative code, because cluttered conditionals may become a real danger to comprehending a program over its evolution. We take extracting conditionals one step further by introducing guard predicates at a prominent level in the language.

The number of approaches that use guards as one of their core concepts illustrates the fundamental importance of guards. With explicit language support for guards, AOP can actually be explained as a specific kind of ECA system. Join points define *events* that are emitted whenever the control flow passes a certain point. Guards define the *conditions* by which events are filtered. *Action* in aspect languages actually means an adaptation of an existing action in the base program further specified as either before, after or replace.

Of course, role guards (`when`) could always be encoded by weaving the condition into all relevant role methods. We see three main reasons for raising guards to the level of a language feature:

1. Guards at the level of classes (role and team) avoid the scattering that would occur in the hand-coded solution.
2. Guards as a language feature encourage programmers to think in terms of modes, conditions, situations etc.
3. Base-level guards allow to prevent the creation of role objects that should not exist, an effect that can only be obtained by a close connection to the runtime semantics of callin binding and lifting.

After the introduction of guard predicates, ObjectTeams/Java now provides these styles of controlling aspect instantiation: First, teams are instantiated and activated explicitly, which gives programmatic control not only over which

[10]Being a MOP-based approach, AspectS certainly has the flexibility to support many different styles, but it can give no static guarantees like team-level predicates, e.g., can.

aspects have an effect on the system, but also on their relative priority. Second, role creation can be triggered by constructor calls from within the team or by declared lifting in team-level methods. Third, role creation caused by a callin binding can be rejected by a guard. So the default behavior for roles is implicit creation on demand, but client code may explicitly add roles and prevent other roles from being created.

6.1 Future work

After academic experiments have identified many situations where guard predicates indeed facilitate very modular designs, an industrial case study using ObjectTeams/Java is currently in progress.

The language design of ObjectTeams/Java is not complete, yet. To date, the only way to specify join points is to enumerate methods by name. Therefor, a full analysis of orthogonality and expressiveness can not yet be done. However, we consider explicit support for guards more fundamental than the exact choice of a query language by which join points can be identified. With guards separated from join points we can now focus on which events actually should be observable.

7. REFERENCES

[1] M. Aksit, M. Mezini, and R. Unland, editors. *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2003.

[2] Lodewijk Bergmans and Mehmet Aksit. Principles and design rationale of composition filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. ISBN 0-32-121976-.

[3] Craig Chambers. Predicate classes. In *Proc. of ECOOP'93*, 1993.

[4] M. Cilia, M. Haupt, M. Mezini, and A. Buchmann. The convergence of aop and active databases: Towards reactive middleware. In *Proceedings of 2nd International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830 of *LNCS*, pages 169–188, 2003.

[5] K. R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of a ADBMS features. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985, pages 3–20. Springer, 1995.

[6] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.

[7] J. Gil and D. Lorenz. Environmental Acquistion — a new inheritance like abstraction mechanism. In *Proc. of OOPSLA'96*, pages 214–231. ACM, 1996.

[8] Kasper B. Graversen and Johannes Beyer. Chameleon, August 2002. Masters thesis. IT-University of Copenhagen.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] Stephan Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In Aksit et al. [1].

[11] Stephan Herrmann. Confinement and representation encapsulation in object teams. Technical Report 2004/06, Technical University Berlin, 2004.

[12] Stephan Herrmann. Translation polymorphism in Object Teams. Technical Report 2004/05, Technical University Berlin, 2004.

[13] Robert Hirschfeld. AspectS - aspect-oriented programming with squeak. In Aksit et al. [1].

[14] Christine Hundt. Introducing Dynamic AOP to Object Teams. Poster at European Interactive Workshop on Aspects in Software EIWAS'04, `http://www.topprax.de/EIWAS04`, 2004.

[15] B. Jørgensen and E Truyen. Evolution of collective object behavior in presence of simultaneous client-specific view. In *Proceedings of the 9th international Conference on Object-Oriented Information OOIS'03*, volume 2817 of *LNCS*, pages 18–32. Springer Verlag, 2003.

[16] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, and J. Palm. An overview of AspectJ. In *Proc. of 15th ECOOP*, number 2072 in LNCS, pages 327–353. Springer–Verlag, 2001.

[17] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proc. AOSD'03*, Boston, USA, March 2003. ACM Press.

[18] Mira Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publisher, 1998.

[19] T. Millstein. Practical predicate dispatch. In *Proceedings of OOPSLA 2004*, October 2004.

[20] Object Teams home page. http://www.ObjectTeams.org.

[21] Object Teams Development Tooling download page. http://www.ObjectTeams.org/distrib/otdt.html.

[22] Doug Orleans. Separating behavioral concerns with predicate dispatch, or, if statement considered harmful. In *Workshop Advanced Separation of Concerns in Object-oriented Systems at OOPSLA'01*, 2001.

[23] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of Object Behavior using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.

[24] Eddy Truyen. *Dynamic and context-sensitive composition in distributed systems*. PhD thesis, Katholieke Universiteit Leuven, October 2004.