Composition&Variability'2010. First International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines

Philippe Lahire, Geri Georg, Mourad Oussalah, Jon Whittle, Naouel Moha, Stefan van Baelen

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR6070

Composition&Variability'2010

First International Workshop on

Composition: *Objects, Aspects, Components, Services and Product Lines*

15 March 2010

To be held in conjunction with
The 9th International Conference on Aspect-Oriented Software Development
(AOSD.10)

Rennes & Saint Malo, France

Editors: **Philippe Lahire, Geri Georg, Mourad Oussalah, Jon Whittle, Naouel Moha, Stefan Van Baelen**

Preface

Separation of concerns is an interesting design concept which is more or less addressed in various paradigms objects, aspects, components and, services in order to achieve software reusability and adaptability. Composition of these concerns is a key issue in software development.

The goal of the workshop is to bring the researchers to discuss on software composition according the paradigm which is used, the degree of dynamicity, the stage in the software life cycle, the application domain and the software variability. More generally, the unique contribution of this workshop is to view composition as it is impacted by several points of variation associated for example to the context of reuse, the time of composition or the business domain.

Composition can be applied in particular on Objects, Aspects, SOA, Component-Based architectures and may address various phases of the development process such as: GUI, design, programming, deployment, and maintenance. We have been particularly interested in having contributions dealing with any combination of a topic taken in "*Composition and paradig* " and "*Composition and product line*" in order to get a view of the composition process colored with variability issues.

We had eleven submissions and the program committee selected only seven of them on the basis of novelty, relevance to the AOSD community, and adequacy to workshop objective. According to the papers which had been selected the workshop will address in particular the following topics:

- ✓ Dynamic (re) configuration, adaptation and composition,
- ✓ Language features for composition and Software Product Lines.

The proceedings are printed as an internal report of the I3S Laboratory[1] and will be included as the Vol-564 on CEUR-WS.org. They may be reached at http://CEUR-WS.org/Vol-564/.

On behalf of the organizing committee

[1] Internal reports may be reached at the URL : http://www.i3s.unice.fr/I3S/

Organisation Committee

(alphabetical order)

- Geri Georg, Colorado State University, U.S.A., (georg@CS.ColoState.EDU)
- Philippe Lahire, I3S laboratory CNRS/University of Nice Sophia Antipolis, France, (Philippe.Lahire@unice.fr)
- Naouel Moha, IRISA laboratory CNRS/University of Rennes, France, (moha@irisa.fr)
- Mourad Oussalah, LINA laboratory CNRS/University of Nantes, France, (mourad.oussalah@univ-nantes.fr)
- Stefan Van Baelen, Department of Computer Science of the K.U. Leuven, Belgium, (Stefan.VanBaelen@cs.kuleuven.be)
- Jon Whittle, Department of Computing, Lancaster University, united Kingdom, (whittle@comp.lancs.ac.uk)

Program Committee

(alphabetical order)

- Olivier Barais (IRISA - Univ. of Rennes, France)
- Djamel Benslimane (LIRIS - Univ. of Lyon, France)
- Christophe Dony (LIRMM – Univ. of Montpellier, France)
- Betty Cheng (Michigan State University , USA)
- Geri Georg (Colorado State University, USA)
- Jeff Gray (Univ. of Alabama at Birmingham , USA)
- Oystein Haugen (SINTEF, Norway)
- Patrick Heymans (FUNDP Namur, Belgium)
- Joerg Kienzle (McGill University, Canada)
- Gunter Kniesel (Univ. of Bonn, Germany)
- Philippe Lahire (I3S – Univ. of Nice Sophia Antipolis, France)
- Naouel Moha (IRISA - Univ. of Rennes, France)
- Manuel Oriol (Univ. of York, United Kingdom)
- Markku Sakkinen (Univ of Jyväskylä, Finland)
- Lionel Seinturier (LIFL – Univ. of Lille, France)
- Stefan Van Baelen (K.U Leuven, Belgium)
- Gilles Vanwormhoudt (LIFL – Univ. of Lille, France)
- Jon Whittle (Lancaster University, united Kingdom)

List of accepted papers

- Tom Dinkelaker, Ralf Mitschke, Karin Fetzer and Mira Mezini. *A Dynamic Software Product-Line Approach using Aspect Models at Runtime*.

- Fredrik Sørensen, Eyvind W. Axelsen and Stein Krogdahl. *Dynamic composition with package templates*.

- Bholanathsingh Surajbali, Paul Grace and Geoff Coulson. *ReCycle: Resolving Cyclic Dependencies in Dynamically Reconfigurable Aspect Oriented Middleware*.

- Tom Dinkelaker, Martin Monperrus and Mira Mezini. *Supporting Variability with Late Semantic Adaptations of Domain-Specific Modeling Languages*.

- Wilke Havinga, Christoph Bockisch and Lodewijk Bergmans. *A Case for Custom, Composable Composition Operators*.

- Christoph Bockisch and Andreas Sewe. *Generic IDE Support for Dispatch-Based Composition*.

- Stefan Walraven, Bert Lagaisse, Eddy Truyen and Wouter Joosen. *Aspect-Based Variability Model for Cross-Organizational Features in Service Networks*.

A Dynamic Software Product Line Approach using Aspect Models at Runtime

Tom Dinkelaker[1] Ralf Mitschke[1] Karin Fetzer[2] Mira Mezini[1]

[1]Technische Universität Darmstadt, Germany
{dinkelaker,ralf.mitschke,mezini}@cs.tu-darmstadt.de

[2]SAP Research Dresden, Germany
karin.fetzer@sap.com

ABSTRACT

Dynamic software product lines (DSPLs) are software product lines, which support late variability that is built into the system to address requirements that change at runtime. But it is difficult to ensure at runtime that all possible adaptations lead to a correct configuration. In this paper, we propose a novel approach for DSPLs that uses a dynamic feature model to describe the variability in the DSPLs and that uses a domain-specific language for declaratively implementing variations and their constraints. The approach combines several trends in aspect-oriented programming for DSPLs, namely dynamic aspects, runtime models of aspects, as well as detection and resolution of aspect interactions. The advantage is, that reconfigurations must not be specified for every feature combination, but only for interacting features. We have validated the approach in an example dynamic software product line from industry and preliminarily evaluated the approach.

Keywords

Dynamic Software Product Line Engineering, Dynamic Feature Models, Domain-Specific Languages

1. INTRODUCTION

Large scale information technology infrastructures are the backbone of many enterprise processes. Yet these systems are driven to continuous adaptation, due to changing requirements [10]. However, the evolutionary transitions for crucial enterprise information systems must be smooth and not hamper current running business processes [18]. Hence, methods and mechanisms for dynamic adaptation of the software system are required.

The challenge of building dynamically adaptable software systems is how to define suitable methods and mechanisms for the dynamism. An ad-hoc approach is to use existing variability mechanisms (e.g., *if*-statements, method dispatch) directly in the architecture, and/or the underlying implementation. However, lacking appropriate methodologies for building such software systems, the rising complexity (i.e., number of configurations, complex re-configuration relationships) can limit the number of dynamic re-configuration points to a few well-defined ones (e.g., all points at which variability must be supported must be known at design time, and the corresponding if-statements and all possible variations must be provided).

Dynamic software product lines (DSPLs) [11, 13] are an emerging field that can systemize the configuration space in dynamically adaptable software system. Thus, DSPLs break down the complexity of managing dynamic re-configuration points by modeling them explicitly in a product line approach as *late variability* [22]. Central to software product lines (SPLs) are features, where a feature is a distinct property of the software product. The *late variability* can be represented through *dynamic features*, i.e., features that can be (de-)activated in a running software system.

A research challenge for DSPLs is to find suitable variability mechanisms to support *dynamic features* in the underlying architecture and implementation. The mechanism must not constrain the range of existing techniques used to build product lines, i.e., it should peacefully co-exist with model-driven and generative techniques. Further, the mechanism should be able to cope with dynamic features that affect several modules and require modification of several classes or components in the product line. It should also detect and resolve interactions between features, in particular feature interactions that are not statically detectable but arise for a set of dynamic contexts of the configuration.

An interesting research question for *aspect-oriented programming* (AOP) [16] is how far dynamic AOP [2, 20, 19, 5, 7] is capable as a variability mechanism for dynamic SPLs. For example, dynamic aspects [4] have been used to implement business rules. Although dynamic AOP solutions provide a flexible variability mechanism, they do not provide appropriate support for declaring, detecting, and resolving dynamic interactions. Most dynamic AO solutions only provide support for defining the precedence of aspects [2], i.e., defining the execution order of their advice. But these precedence relations are inappropriate for expressing exclusions, i.e., one cannot declare that one aspect does not allow another aspect to be present. Furthermore, dynamic dependency relations between the features implemented by aspects cannot declare that one aspect needs another aspect to work correctly. The works in [20, 5] support static declaration of exclusions and dependencies between aspects, but do not address dynamic interactions.

First Workshop on Composition and Variability'10 Rennes, France colocated with AOSD'2010

In DSPLs the dynamicity in the exclusion of features is of great interest. The problem with static exclusions and dependencies is that they must be declared permanently (e.g., aspect `A` always excludes aspect `B`) and are enforced independent of the fact that aspects `A` and `B` may not actually conflict because the aspects are never applied at the same points at runtime. This is too conservative and disallows meaningful compositions of dynamic features.

For example, a workflow requires an approval step, which can be automatic manual. Modularizing the variability in the approval step entails interaction between the automatic and the manual variants, because the goal of immediate automated approval is violated by waiting for a manual approval. The repetitive manual approval following an automated appoval would be a waste of human resources.

However, using more powerful declaration mechanism, we can declare that both dynamic features be selected at once and affect different parts of the DSPL, with the exception of cases in which conflict occurs. For example, when the automatic approval is only applied to orders that have a certain state (e.g., exceeds a certain amount), a conflict occurs only for those particular orders and must be resolved only when this runtime condition is satisfied. The other orders are only affected by the manual approval without any conflicts.

In this paper, we propose mechanisms to detect and resolve such context-dependent interactions between features in a DSPL. The contribution of this paper is twofold. On the one hand, we adopt DSPLs as a systematic framework in which complex dynamic software systems can be planned and managed by modeling late variability. On the other hand, we provide support for detecting and resolving context-dependent interactions by validating an aspect-oriented (AO) model at runtime.

First, we extend existing DSPL approaches by a novel notation termed *dynamic feature model* that allows us to model *late variability*. Feature models are a widely used notation, that models the configuration space of software product lines, yet they lack explicit support to model *late variability*. Our notation extends feature models to capture *dynamic features*, i.e., features that may be (de-)activated at runtime, and to model their runtime constraints. Thus we provide an explicit representation of dynamic re-configuration points in an adaptable software system.

Second, we propose a novel approach for DSPLs on top of a dynamic AO runtime environment with a meta-aspect protocol [7] that is used as a *dynamic feature manager*. We map dynamic feature models to aspect-oriented models that are available as first-class entities in running products. DSPLs are delivered with the AO runtime environment, through which (de-)activation of dynamic features is enabled, which updates dynamic feature model representation, takes care of the composition of dynamic features, as well as it detects and resolves dynamic feature constraints.

We have validated the approach by evolving a static software product-line from industry into a DSPL in a case study. Furthermore, we have evaluated the performance overhead of dynamic aspects in the context of SPLs. The results indicate that the approach copes well with performance requirements, also in the presence of the special scalability requirements of SPLs.

The remainder of this paper is structured as follows. Sec. 2 illustrates the example SPL and motivates late variability. In Sec. 3, we present dynamic feature models for modeling late variability and give an overview of the methodology. In Sec. 4, we describe how dynamic aspects can be used to realize dynamic SPLs. Sec. 5 presents the case study. Sec. 6 evaluates the approach. Sec. 7 discusses related work. Sec. 8 concludes the paper and outlines future work.

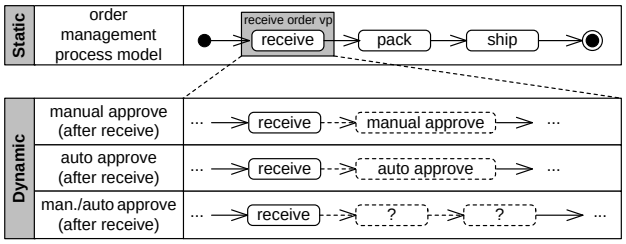

Figure 1: Late variation point for approval step in the order management business process

2. LATE VARIABILITY IN AN SPL

Our example SPL is the *Sales Scenario*, which is a software system for the management of business data, including central storage and user-centric information inquiry. The main focus of the *Sales Scenario* is on stock sales processes, where the core processes are customer order management, payment, account management, stock management and communication. The main goal of the *Sales Scenario* is to integrate all processes and corresponding data of an organization into one system. The system addresses sales processes for both mid-sized and large enterprises. The business processes themselves are customizable, often in multiple independent variations. For example the order management process can include a quotation management (i.e., placing strategic offers to customers) or sales order processing (i.e., tracking of individual orders).

The customer needs for these features vary depending on the business size of the customer, thus making it advantageous for the software provider to implement the system as a SPL. The products resulting from the SPL might be as small as a simple way to keep track of executed orders, or as large as a complete sales management system, in which everything from the first idea of a sales opportunity to the delivery and payment of the sold product can be managed.

The processes in the *Sales Scenario* are modeled using model-driven techniques. Thus, we can identify variation in the processes on the model level. For example, Fig. 1 depicts a short version of the order management process, which consists of at least three steps: "receive" (order is received from a customer), "pack" (order items are prepared for delivery), and "ship" (order is sent to the customer). The lower part of Fig. 1 models an example of late variability in the order management process. The modeling elements with solid lines describe the static part of the process. The receive step is subject to a dynamic variation that is described using modeling elements with dashed lines.

As an example for late variability, we model an additional approval step after an order is received. We have identified two subcategories of approval, namely a manual approval, which requires human interaction with the system, or an automated approval, e.g., a check if the order is issued by credit-worthy customers. To provide flexibility we allow customers of the product line to adapt their software system

from one process model to another without the need to redeploy the whole system. Therefore, these two variants of approval processes are modeled as *dynamic features*.

A conceptual problem arises when multiple *dynamic features* are composed, e.g., when a customer activates both of the above features at once, as indicated in the last row of Fig. 1. When composing the features, it is necessary to take into account their semantics. One solution would be that the automated approval manages all orders and that we require human intervention only in cases where the automation does not approve of an order. However, we cannot describe such compositions with the current technology.

Existing approaches support a form of linear composition for features, by controlling their order through precedences [1, 19, 5]. But, declaring that the automated approval step precedes the manual approval step is not enough, since we want to declare that the automated approval must be executed and that the manual approval must be skipped. The above approaches also provide composition strategies to declare and enforce a static exclusion constraint between features. For example, the manual approval feature always excludes the automatic approval feature in all configurations.

The problem is that static precedences and exclusions are too conservative. If the constraint between features only occurs in a certain (runtime-) context, such strategies are inappropriate. A correct strategy must take into account the application context. For example, a manual approval is selected only for a certain group of orders, e.g., orders with a high quantity, while the rest is approved automatically. Without knowledge of the domain and application semantics, we cannot express such a composition scenario at the level of feature modeling.

3. MODELING DYNAMIC SPLS

Dynamic feature models are an extension of existing 'static' feature model notations, such as [6], and provide a specialized notation to specify the product lines dynamism and runtime constraints. The dynamic constraints allow expressing i) that the activation of one dynamic feature requires that another dynamic feature be active as well. This constraint is termed *implies* in dynamic feature models and allows SPL designers to model requirements on the reconfiguration logic of the DSPL. Furthermore, we model ii) that two features must not be simultaneously active, by constraining them with *excludes*. This constraint allows restricting the running system from activating both features, if an SPL designer deems their combination harmful due to possible interactions. The iii) *precedes* constraint declares that an interaction of features is allowed and states a resolution strategy, which grants one feature precedence over another. The precedence is not exclusive, thus all features are active but at interacting points, the precedence defines an order in which the features are taken into account.

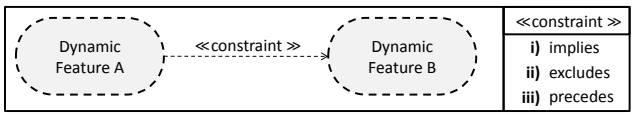

Figure 2: Notation of dynamic features

Fig. 2 depicts the visual representation of dynamic feature models and the possible constraints. Dynamic features are depicted with dashed border lines. Likewise dynamic constraints are modeled using dashed arrows.

We impose some limitations in the usage of our constraints. First of all, the conjunction of *implies* and *excludes* is forbidden, as this combination is not satisfiable by the DSPL. For multiple features with multiple *precedes* constraints we disallow cycles, thereby all involved precedence constraints must form a chain. Furthermore, the combination of *precedes* and *excludes* ii) + iii) is allowed and states that the feature with the highest precedence is the only one taken into account at points where these features interact.

Static constraints may also be formulated between dynamic and static features. For example, a dynmic feature requires the presence of a static feature. The static constraints must be enforced with the same semantics as having static constraints between static features.

4. DSPLS USING DYNAMIC AOP

Our approach realizes a DSPL by mapping dynamic features and their interactions to an *aspect-oriented model* at runtime. The AO model consists of first-class entities, such as *dynamic aspects* and their *pointcuts-and-advice*, *primitive pointcut designators* that match *join points*, and *rule base* with declared constraints on aspect interactions. Dynamic features are mapped to dynamic aspects, which adapt *late variation points* in the DSPL. To enforce the modeled constraints, feature interactions are mapped to constraints on dynamic aspects. The aspect model is used as a first-class runtime representation of the dynamic feature model. The AO model is validated to ensure consistency when features are dynamically activated and deactivated at runtime.

To express dynamic features in terms of dynamic aspects, SPL developers define a DSL. This language, also denoted as *dynamic feature language* (DFL), incorporates the necessary abstractions for the specification of dynamic features in terms of domain concepts. In addition, the DFL provides the required aspect-oriented machinery to declare constraints on dynamic features, as well as semantics for the composition of dynamic features. The DFL enables a safe feature composition, because dynamic feature interactions are automatically detected.

As the underlying language technology for implementing the DFL, we use Popart [7], which is a dynamic aspect-oriented language that provides generic AO mechanisms and that is extensible for new domain-specific syntax and semantics. Domain extensions are integrated into Popart to create a complete SPL-specific aspect language.

The process of defining a DFL consists of the steps: 1) late variation points are identified and modeled as a *domain-specific join point model* (DS-JPM) [8], 2) late VPs are made available for the DFL using a *domain-specific pointcut language* DS-PCL [8] that quantifies over the DS-JPM, 3) the results are integrated with a generic declarative aspect-oriented language, that provides commonly used AO concepts (e.g., before/after/around advice). The resulting *dynamic feature language* is thus summarized as: DFL = DS-JPM + DS-PCL + Generic AO Concepts.

4.1 Modeling Late VPs (DS-JPM)

The first step of the SPL developer is to analyze the design of the SPL for possible *late variation points* and model them in a dynamic DS-JPM. For each late variation point, the developer defines the *context* for this VP, i.e., a properties

map that defines identifiers referring to relevant values, e.g., the identifier "customerOrder" refers to the business object in the dynamic context of the running application.

Late VPs can be defined at various levels of abstraction in the SPL and thus identify different artifacts, e.g., model elements or source code points. At the model level, late VPs are modeled as annotations on modeling elements. For example, in a UML activity diagram, one activity is annotated to be dynamically variable. These annotated modeling elements are treated in a special way by code generators. Late VPs require a facility for dynamic de-/activation, thus, the respective source code elements for the model elements are generated, e.g., classes or methods, and the de-/activation is provided by generating domain-specific aspects for these source code elements. At the code level, the late VPs refer to code elements, such as a class, an attribute, a method, or an expression in the body of a method, e.g., when an activity is implemented in a method, the late VP casts on the call to this method.

An excerpt of late VPs that we have identified in the *Sales Scenario* inside the order management workflow is presented in the following:

1) Payment type selection: The customer chooses a specific payment type, e.g., credit card or cash-on-delivery. This step presents various payment types to the customer. Variation at this point can restructure the choice of payment types, e.g., filtering to a more specific list. The context made available at this late variation point are the choice of payment types presented to the customer as well as the customer's concrete choice.

2) Price calculation: The price of an order or a quotation is calculated as the sum of the prices of the contained order items. This is a late variation point, that allows to introduce new pricing strategies and override existing strategies, e.g., to define a discount and allowances on the price. The context is the order for which the price is calculated. From the domain model this implies the exposure of the individual items in the order, since they are accessible via the order.

3) Receive order: The first step in the order management is the reception of new orders. In the *Sales Scenario* orders enter the workflow with all information on the customers and the payment modalities. This step is a late variation point such that we can insert an approval step before packing and shipping the order. Such an additional step then approves whether the customer is trustable before an unpaid cash-on-delivery order is shipped. The available context is the customer, the order, and the selected payment method.

From the identified late variation points the SPL developer builds the DFL for a particular dynamic software product line. To allow the dynamic aspects to intercept the late variation points of the DSPL, the SPL developer declares an instrumentation [1] of the SPL implementation, that reifies SPL concepts. Using this instrumentation we define a *domain-specific join point model*. In this DS-JPM, each late variation point is represented through a SPL-specific *join point type* as a sub-class of `JoinPoint` and its context is a properties map. For the above late variation points, the SPL developer defines join point types: 1) `PaymentTypeSelectionJP`, 2) `PriceCalculationJP`, and 3) `ReceiveOrderJP`.

[1]In the case-study AspectJ aspects are used to reify runtime information.

```
1  class SalesScenarioDFL extends PointcutDSL {
2    ...
3    Pointcut receive_order (long quantity) {
4      return new ReceiveOrderPredicate(quantity);
5  } }
6
7  class ReceiveOrderPredicate extends PrimitivePCD {
8    ReceiveOrderPredicate(long quantity) {...}
9    ...
10   boolean match(ReceiveOrderJP jp) {
11     long orderQuantity = computeQuantity(jp.context.get("order"));
12     if ( orderQuantity < quantity ) return true;
13     return false ;
14 } }
```

Figure 3: Excerpt of the *Sales Scenario* DFL implementation for selecting an example late VP

The AO instrumentation of the SPL binds context values at late variation points to instances of these join point types. The full technical details of the definition of a DS-JPM are elaborated in [8].

4.2 Quantification over Late Variation Points (DS-PCL)

To allow the dynamic features to quantify over late variation points, the SPL developer declares predicates on the late variations points. In principle, a dynamic feature can select every one of the defined late variation points. All late variation points are made available by a set of predicates, which can be combined using logical expressions (and, or, not). For each late variation point, a predicate is defined that selects this point. For the above late variation points, the following predicates are defined: 1) `payment_type`, 2) `price_calculation`, and 3) `receive_order`.

Further, the predicates can be parameterized, e.g., to constrain late variation points depending on runtime values of the application context. For example, `receive_order(quantity)` defines a parameterized predicate that filters late variation points, where the quantity of the order is less than a certain threshold, e.g., `receive_order(1000)` selects the late variation points of all orders with a quantity of less than 1000 units. Fig. 3 depicts the implementation of the `receive_order(quantity)` predicate, which constitutes a domain-specific keyword in the *Sales Scenario* DFL. Using the POPART framework, the `receive_order(long)` method becomes a keyword in the aspect runtime that can be used to declare where a dynamic feature is active. The concrete matching happens in the `ReceiveOrderPredicate`, where the `match` method of the framework is adopted to match at join points (`ReceiveOrderJP`) that have an order in their runtime context, that contains a quantity lower than the threshold. For every predicate, the *domain-specific pointcut language* (DS-PCL) defines a domain-specific keyword that selects the join points of the corresponding late variation point. Each DS-PCL keyword creates a *primitive pointcut designator* as a sub-class of `Pointcut` in the AO model used to filter `JoinPoint` objects. More details about implementing a DS-PCL are elaborated in [8].

4.3 Generic AO Concepts

To define dynamic features as aspects the following general-purpose AO concepts are provided by POPART and reused in the DFL. The `aspect` keyword defines a new dynamic aspect module, parameters define its name and initial activation

status (`deployed`), i.e., active (`true`) or inactive (`false`). POPART allows to define where to insert actions at a late variation point using `before/after` advice, which execute before or after reaching the late variation point. In addition, `around` advice replaces the actions of a late variation point and `proceed` invokes the replaced actions in an around advice. To define constraints from the feature model the following mechanics are used: i) `assert` validates a boolean expression when loading the aspect and is used to model dependencies to static features. ii) `declare_dependency`, `declare_exclusion`, and `declare_precedence` are used to declare aspect interaction constraints that are detected and resolved at runtime by POPART.

4.4 DSPL specific AO language (DFL)

To instantiate the DFL, the SPL developer mixes the existing generic AO language with the specific parts for the SPL. POPART take care that all common and specific parts of the AO syntax are integrated together into a SPL-specific aspect language

Using the DFL, each dynamic feature is mapped to a dynamic aspect. The `aspect` is declared with a unique name, that maps to the corresponding feature and the aspect is either active or not depending on the (default) choice of the user. For each specific variation at a late variation point, the aspect defines a pointcut-and-advice. Its pointcut uses the DS-PCL to quantify over join points (i.e., it intercepts the execution of a late variation point) and its advice defines how to adapt selected variation points by inserting or replacing certain actions at the join point (i.e., it adapts a late variation point). We will discuss concrete examples of dynamic aspects in the next section.

Each constraint on dynamic features is implemented as an aspect interaction in one of the aspects. A *dynamic constraint* is defined using one of the `declare`-keywords. For *implies*, an aspect declares `declare_dependency` between the two aspects with the corresponding feature names. For *excludes*, the aspect uses `declare_exclusion`; and for *precedence*, the aspect uses `declare_precedence` instead. Note that for symmetric constraints such as *excludes* it does not matter which aspect actually declares the interaction. Recall that dynamic feature models are an extension to 'static' feature models. A *static constraint* on features can also be defined using the `assert` keyword.

Using our AO model at runtime for the composition of features and the detection and resolution of constraints has several advantages. First, there is no need to consider all combinations of feature selections. When dynamic features are selected or deselected at runtime, the DSPL is automatically adapted by POPART as aspects are composed at join points in the runtime model of the application. Second, the dynamic AO mechanisms allow the declaration of runtime context-dependent feature interactions in conjunction with a continuous enforcement of these constraints by validating possible aspect interaction as specified in the rule base of the AO model. Thus the DFL allows the safe specification of features that interact with each other, because advice are ordered, conflicting advice are never executed at the same time, and dependencies are enforced. We will see example resolutions in the next section.

5. CASE STUDY

To validate our concept, we have implemented the *Sales Scenario* as an example dynamic SPL, parts of which were introduced in Sec. 2. While static variability is modeled and implemented using existing technologies, the late variability is modeled and implemented using the technology that is presented in this paper and that helps to manage late variability. The late variability technology seamlessly integrates with the above technologies in the Eclipse-based workbench. In the remainder of this section, we first summarize the static part of the *Sales Scenario*, then we elaborate how the dynamic features are implemented using the feature DSL.

For the implementation of the static part of the SPLs, we used the Eclipse based facilities for developing SPLs, provided by the feasiPLe research project [9]. The static variability is modeled and implemented using extension of existing methodology, adapted by feasiPLe to better support model-driven and aspect-oriented software development of SPLs.

To give a short overview of the methodology: 1) we designed *domain-specific languages* (DSLs) for the different application domains (e.g., process, business objects, graphical view, etc.) as Ecore[2] metamodels, and instantiated them into *variant independent models* (VIMs). 2) The model elements in these models were then mapped to features using the *FeatureMapper* [15], and 3) using this mapping the VIMs were transformed into *variant specific models* (VSMs) using *pure::variants*[3] For each DSL, we also implemented 4) one code generator in Xpand[4] , and generated Java and AspectJ [1] code based on these VSMs using the code generators. In summary, the *Sales Scenario* has 27 static features and six dynamic features. The implementation consists of 4,000 Java hand-written lines of code (LOC), 17.000 LOC generated Java, 10.000 LOC related to oAW artifacts, 6,000 LOC AspectJ, and 130 LOC Groovy/POPART.

In Fig. 4 the dynamic feature model of the *Sales Scenario* is presented, of which we will discuss first the dynamic approval feature, and then the dynamic pricing strategy feature. The purpose of the dynamic approval feature is to validate customer creditability to reduce risk for large quantity orders. An implementation of the dynamic approval feature from the *Sales Scenario* is shown in Fig. 5. In lines 1–8, the class `OrderManager` (realizing the `Customer Order Mgmnt` feature) is shown that is part of the static part of the SPL. It implements one method for each step in the order management use case, i.e. `receive`, `pack`, and `ship`. The execution of the method `receive` (lines 3–5) constitutes a late VP as modeled in the previous section. In Fig. 3, the class `ReceiveOrderJP` represents executions of the `receive` method and is used to declare the `receive_order` predicate. This predicate is used in lines 10 and 15 (parameter `quantity` is optional), to specify where the different approval steps are inserted.

For the dynamic features, the three aspects in the example are deployed to the running system. The `Manual-Approval` aspect defines a pointcut that selects the late VP of the `receive` step by using the corresponding predicate `receive_order` defined in Sec. 4. The advice extends the SPL at the selected variation point by opening a dialog (line 11)

[2]http://www.eclipse.org/modeling/emf/
[3]http://www.pure-systems.com/
[4]http://www.openarchitectureware.org/

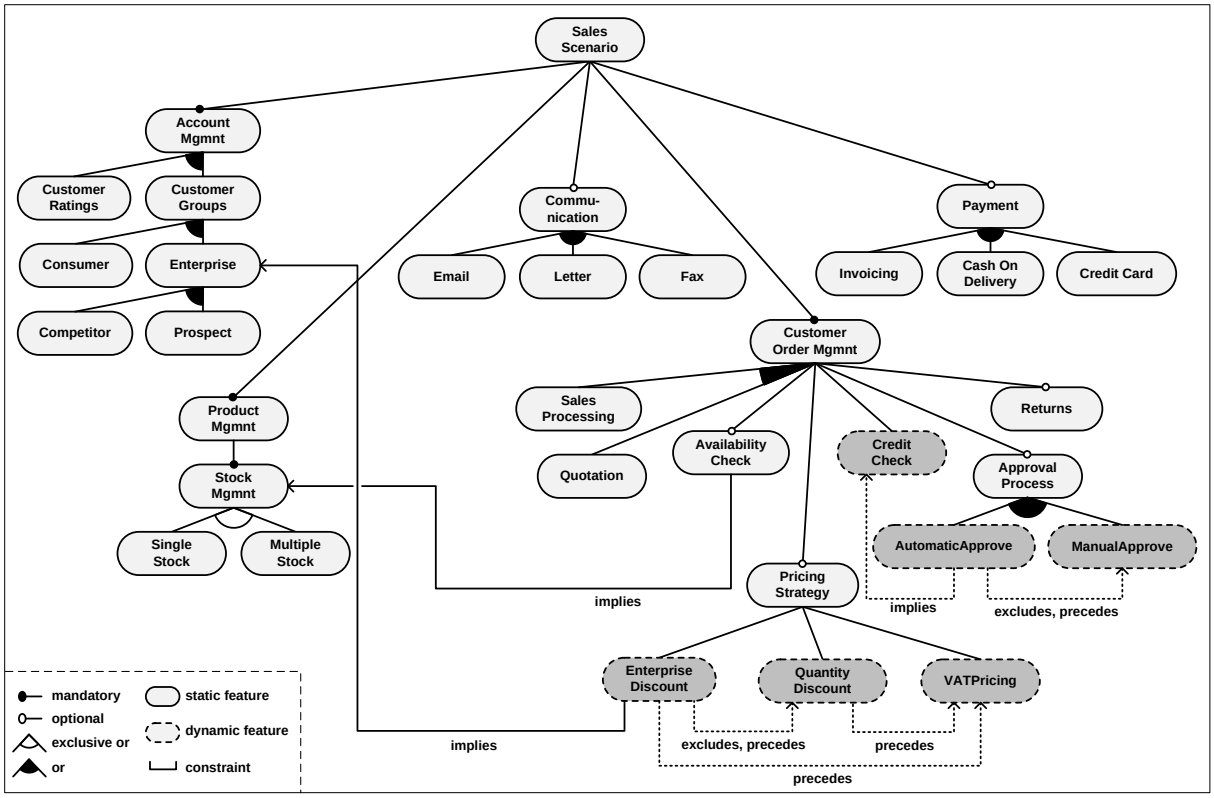

Figure 4: Dynamic feature model of the *Sales Scenario*; extending notation from [6] with dynamic features

```
1  class OrderManager {
2    ...
3    void receive(Order order) {
4      //receive the order from customer (is a dynamic VP)
5    }
6    void pack(Order order) {...}
7    void send(Order order) {...}
8  }
9  aspect(name:"ManualApproval", deployed:true) {
10   after( receive_order () ) {
11     boolean trustable = UI.openApprovalDlg(order).isCustomerTrustable();
12     if (! trustable ) // clarify customers creditibility
13 } }
14 aspect(name:"AutomaticApproval", deployed: false ) {
15   after( receive_order (100) ) { ... } }
16 aspect(name:"CreditCheck") {...}
17 aspect(name:"ApprovalInteraction ", deployed:true) {
18   declare_exclusion "ManualApproval", "AutomaticApproval";
19   declare_precedence "AutomaticApproval", "ManualApproval";
20   declare_dependency "AutomaticApproval", "CreditCheck";
21 }
```

Figure 5: Dynamic feature: order approval

for the manual approval in a user interface. If the user is not trustable, a sub-workflow is invoked (line 12) to clarify the status of the customer, e.g., the customer presents further credentials, pays the order before shipment, or the process is aborted. The aspect `AutomaticApproval` (line 15) is implemented similarly to the manual approval feature, but only executes at late VPs where the quantity of the received order is smaller than 100. As an automation, customer creditability is checked via the corresponding credit card. To illustrate the SPL dynamicity, the `AutomaticApproval` aspect is not deployed (i.e. inactive) during startup. At a later point, i.e. when the company exceeds a certain amount of placed orders, the automated approval is deployed.

The feature interaction between the two approval features is mapped to an aspect constraint, which is delcared in a separate aspect (`ApprovalInteraction`). Line 18 declares the aspects `ManualApproval` and `AutomaticApproval` to be mutually exclusive, i.e., they may not affect variation points at the same time. The following line declares the precedence constraint between the two features. We choose to declare the interaction in a separate aspect, because this has the advantage that the implementation of the two aspects is independent from each other. When the `AutomaticApproval` aspect is deployed, the interaction at the late VP is detected and resolved according to the constraints. Because of the dynamic exclusion constraint in line 18, a conflict is detected that is resolved by taking into account the dynamic precedence constraint in line 19. As the `AutomaticApproval` has a higher precedence, its advice will be executed and the advice of `ManualApproval` are skipped. Because of the dynamic dependency constraint in line 20, the advice of `AutomaticApproval` requires the `CreditCheck` feature to be deployed.

For the *Sales Scenario*, we have implemented a flexible pricing strategy feature using our late variability support, to introduce new pricing strategies as dynamic features into a running product line. The static part of our SPL comes with a simple pricing strategy that calculates the price of an order by calculating the sum of the price of its order items. However, in the context of discounts, allowances and taxes, the actual price of an order depends on various requirements

```
1  aspect(name:"VATPricing") {
2    final float VAT_FACTOR = 1.19; //Currently the German VAT is 19%
3    around ( pricing ()) {
4      return proceed() * VAT_FACTOR;
5  } }
6  aspect(name:"EnterpriseDiscount") {
7    assert Class.forName("EnterpriseCustomer") != null;
8    ...
9  }
10 aspect(name:"QuantityDiscount") {...}
11 aspect(name:" PricingInteraction ") {
12   declare_precedence "QuantityDiscount", "VATPricing";
13   declare_precedence "EnterpriseDiscount ", "VATPricing";
14   declare_exclusion "EnterpriseDiscount ", "QuantityDiscount";
15   declare_precedence "EnterpriseDiscount ", "QuantityDiscount";
16 }
```

Figure 6: Dynamic feature: pricing strategy

from the business domain, e.g., there are business rules that add the value added tax (VAT) to the order price, depending on the customers country or rules that give various discounts to certain customers. In the context of discounts, the interaction of features is again of high interest. Depending on the actual context, two discounts are applicable to one order at the same time, or only one discount is allowed to be applied.

A late VP has been inserted into the price calculation of orders that exposes the necessary context, such as the `order`, its `items`, and the `customer`. We use 3 aspects shown in Fig. 6 for realizing the dynamic pricing features: 1) `VATPricing`: calculating the VAT, 2) `EnterpriseDiscount`: giving a discount to enterprise customers[5], and 3) `QuantityDiscount`: a special discount is applied when the order contains a large quantity of items. Note that all aspects advise the same late variation point through the `pricing` predicate.

In this scenario the dynamic feature interactions between the pricing strategies must be handled by appropriate aspect constraints (Fig. 6), as declared in the `PricingInteraction` aspect. The tax calculation is performed after all other calculations have been applied, consequently the `VATPrice` aspect is declared to have the lowest precedence, using the aspect precedences in lines 12 and 13. The `EnterpriseDiscount` feature requires that the static feature *Enterprise* has been selected for the product. To check the presence, an assertion is used to check whether the class `EnterpriseCustomer`, corresponding to the *Enterprise* feature, is available in the product. This static assertion is checked during startup of the application.

The `EnterpriseDiscount` and the `QuantityDiscount` exclude each other (line 14), since for these particular discounts in the product line we choose to disallow double discounts. Such a situation arises only if an order contains a significant quantity of items and is ordered by an `EnterpriseCustomer`. Because of the exclusion constraint in line 14, the interaction of `QuantityDiscount` and `EnterpriseDiscount` is detected as a conflict. Because in line 15 the `EnterpriseDiscount` is declared to have a higher precedence than `QuantityDiscount`, POPART can resolve this conflict by not excluding the effects of the dynamic feature with a lower precedence, i.e. `QuantityDiscount`, in the composition.

[5]The order business object (BO) refers to the corresponding customer BO, which is typed as a representative of a company or a private customer. We use the type to decide if the discount should be applied.

6. EVALUATION

There are certain limitations in the current implementation that prevent our technology to be used in production.

1) In a real-world business scenario, new business rules would need to be defined and loaded to the *Sales Scenario* during its lifetime. In the case study, we provide support only for de-/activation of features via a management console. For convenient runtime evolution a special management console is required, through which new dynamic feature can be uploaded into a running system. POPART comes with the necessary support, since it allows to deploy aspect definitions provided as a String, due to its roots in Groovy.

2) Our approach does detect feature interactions if the interacting aspects affect the same join point, but omits certain indirect interactions, e.g., two aspect accessing shared state. Such interactions are also possible using our approach, i.e. through the contexts available to aspects. Using these interactions in a structured way can prove advantageous. For example in the *Sales Scenario* this allows us to define a manual approval, that checks in the context of the join point, that the order was not automatically approved and only in this case asks the user with a feedback. However, such interactions are currently not detected by POPART and are not modeled at the level of the feature model. How to capture such feature interactions in a structured way at the modeling level is an interesting research question.

3) When deploying new dynamic features at runtime, the integrity of internal state of adapted use cases is not ensured by POPART. Adapting a running use case that has an internal state is difficult, as for example previously started stackframes may be omitted from the aspects execution leading to erroneous internal state. In our case study, we did not experience such problems because deploying aspects was only allowed after all running instances of a business process, e.g. the order management use case, were completed.

To evaluate our approach w.r.t. performance scalability, we have determined the relative instrumentation overhead incurred by the AO runtime. We executed our *Sales Scenario* case study with and without our AO runtime, i.e., in POPART and in Java. To measure the bare instrumentation overhead, we do not apply any dynamic aspects at the declared late VPs. Thus the basic AO instrumentation is in place and delegates to our matching algorithm, which does almost nothing, i.e. iterating over an empty list of potential dynamic aspects. We measured the relative overhead incurred by the instrumentation for repeating execution of the variation point and found the approach to scale well with the number of late VPs. When the VP is executed only once there is a large overhead of 97%, but when the late VPs is visited more often, the overhead is reduced to a value as low as 0.8% (1000 executions). This is due to a *dynamic adaptive optimization* applied by the Java virtual machine, which identifies frequently called methods at runtime and performs more advanced optimizations such as inlining.

7. RELATED WORK

Most AOP tools only support aspect precedences similar to AspectJ [1]. Several AOP tools allow expressing aspect dependencies (such as [19, 7]) but there is little work on context-dependent interactions [14, 17, 7].

Context-oriented programming [5] supports modularizing features into layers of functionality that can be activated

and deactivated at runtime. This work supports only static dependencies between interacting features.

Research on dynamic product-lines [12] [13] is particularly relevant. In [11], DSPLs are specified as a set of components that can be exchanged at runtime. The components follows the design of *software reconfiguration patterns* and have a set of state charts that define all valid reconfiguration cases. In contrast to our approach, components have to implement patterns and interfaces, features with a crosscutting character are not modularized, and runtime evolution is disallowed because all possible reconfigurations must be known and enumerated into the state chart models at design time.

Cetina et al. [3] discuss possible architectures of dynamic software product-lines and distinguish connected and disconnected architectures for DSPLs, depending on whether the DSPL or the product is responsible for reconfiguration. They propose to follow a hybrid approach that combines the best of both, our approach can be used to implement such a hybrid approach, because every product is delivered with a runtime model of the DSPL. Our approach complements their discussion by prosing a concrete realization.

Trinidad et al. [21] propose the realization of DSPLs through a mapping of features onto components in a component model. Their component architecture introduces the specialized concepts of feature component that can be de-/activated and feature relationship that can be un-/linked. However, the approach does not consider crosscutting features, it enforces only static constraints on features, and it does not allow to consider runtime context.

8. CONCLUSION

We have proposed a novel approach for dynamic software product-lines that uses a dynamic feature model to describe the variability in the DSPLs. The approach combines several trends in aspect-oriented programming for DSPLs, namely dynamic aspects, runtime models of aspects, as well as detection and resolution of aspect interactions. We have implemented and validated the approach and preliminarily evaluation results show its scalability.

Although, current support for managing aspect interactions is weak in existing dynamic AOP tools, we strongly believe that dynamic AOP solutions in general can be used for dynamic product-lines. The biggest challenges for dynamic AOP for DSPLs are a) addressing the limitations found when building (static) SPLs that are also present in dynamic AOP, b) improving the support to handle aspect interactions in particular context-dependent interactions, and c) scalability requirements, such as performance in case of large DSPLs.

Future work will address the current limitations. In particular, we would like to provide better means to scope feature constraints and wildcards in constraint expressions, e.g., to specify that a constraint must be enforced a global application scope, for all sub-features of a certain feature, and for all features that names starts with a certain prefix.

9. REFERENCES

[1] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[2] CaesarJ Homepage. `http://caesarj.org/`.

[3] C. Cetina, V. Pelechano, P. Trinidad, and A. Cortes. An Architectural Discussion on DSPL. In *Software Product Line Conference*, pages 59–68, 2008.

[4] A. Charfi and M. Mezini. Hybrid Web Service Composition: Business Processes meet Business Rules. In *International Conference on Service Oriented Computing*, pages 30–38, 2004.

[5] P. Costanza and T. D'Hondt. Feature Descriptions for Context-oriented Programming. In *Workshop on Dynamic Software Product Lines [12]*, 2008.

[6] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[7] T. Dinkelaker, M. Mezini, and C. Bockisch. The Art of the Meta-Aspect Protocol. In *AOSD*, 2009.

[8] T. Dinkelaker, M. Monperrus, and M. Mezini. Untangling crosscutting concerns in domain-specific languages with domain-specific join points. In *Workshop on Domain-specific Aspect languages (at AOSD)*, 2009.

[9] The feasiPLe Homepage. `http://feasiple.de/`.

[10] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, pages 24–32, 2007.

[11] H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. *LNCS*, pages 435–444, 2004.

[12] S. Hallsteinsen and et al. International Workshop on Dynamic Software Product Lines (DSPL). In *International Software Product Line Conference*, 2007-2009.

[13] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[14] J. Hannemann, R. Chitchyan, and A. Rashid. Analysis of aspect-oriented software. *LNCS*, 2004.

[15] F. Heidenreich, J. Kopcsek, and C. Wende. Featuremapper: Mapping features to models. In *ICSE*, 2008.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, 1997.

[17] G. Kniesel. Detection and Resolution of Weaving Interactions. *Transactions on Aspect-Oriented Software Development V*, page 186, 2009.

[18] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. Krämer. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(2):223–255, 2008.

[19] É. Tanter. Aspects of composition in the Reflex AOP kernel. *LNCS*, 4089:98–113, 2006.

[20] E. Tanter and J. Noye. A Versatile Kernel for Multi-language AOP. In *GPCE*, 2005.

[21] P. Trinidad, A. Ruiz-Cortés, J. Pena, and D. Benavides. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *DSPL [12]*, 2007.

[22] J. van Gurp. *Variability in software systems: the key to software reuse*. PhD thesis, Dept. of Software Engineering & Computer Science, Blekinge Institute of Technology, 2000.

Dynamic Composition with Package Templates

Fredrik Sørensen
University of Oslo
Department of Informatics
P.O. Box 1080, Blindern
N-0316 Oslo, Norway
fredrso@ifi.uio.no

Eyvind W. Axelsen
University of Oslo
Department of Informatics
P.O. Box 1080, Blindern
N-0316 Oslo, Norway
eyvinda@ifi.uio.no

Stein Krogdahl
University of Oslo
Department of Informatics
P.O. Box 1080, Blindern
N-0316 Oslo, Norway
steinkr@ifi.uio.no

ABSTRACT

We show how *package templates*, a mechanism for code modularization, can be extended with features for dynamic loading. We pose the question of whether or not this is may a useful mechanism with respect to software composition and dynamic configuration of software.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*

General Terms

Languages, Design

Keywords

OOP, Modularization, Dynamicity, Templates

1. INTRODUCTION

There are several challenges when working with a large software system, including modularization, separation of concerns and reuse. These challenges are large enough in themselves when writing, testing and maintaining a large system. However, even more challenges arise when there are many different variations of a system, when the environment changes over time and when new requirements and extensions may arrive after the initial system has started running and should not be taken down.

Given these challenges, it seems beneficial to have similar support for such dynamic features as one has for the static build of large systems. One mechanism that is useful in this respect is the use of interfaces and abstract classes in writing the system and the possibility of loading different implementations into a running system based on configurations and runtime events. These implementations may be written and separately compiled after the system they are loaded into has been started.

Although a set of such classes may be written, compiled and loaded together, they are still considered as separate entities and not checked as one coherent entity upon loading. There is no assurance that the individual classes belong to the same version of the extension. In this work, we are looking at a way to dynamically load an entity that represents a group of classes in a package or template. We believe it to be useful if one is allowed to take a group of statically checked classes, that are compiled and tested together, and adapt them in a coordinated fashion to an existing system.

A recent paper [15] describes a new language mechanism called PT (short for Package Templates), which is meant to be a useful tool for software composition. The mechanism is intended for object oriented languages in order to support better code modularization, where statically type-checked templates can be written independently and subsequently be instantiated in programs with such flexibility that classes may be *merged*, methods renamed and new classes added. Templates may contain hierarchies of classes (single inheritance) and these hierarchies are preserved when a template is used. Also, different instantiations of the same templates are independent of each other, creating unique types.

The basic PT mechanism allows flexibility in package reuse, program composition and support for readability and reusability in large systems. The way that multiple classes may be affected by instantiating a template gives the language an AOP-like quality. More AOP-specific extensions to PT have also been studied in [3].

In this work, we look at a possible extension to the basic package template mechanism that supports dynamic loading of package templates. We ask to what extent this will be a useful tool for dynamic configuration of software systems. Furthermore, we discuss some of the properties of this mechanism.

We introduce an extension to PT where templates are parameterized by templates. A template with template parameters must be instantiated with actual templates that "extend" the formal parameters' bounds. In standard PT, all instantiations of templates are done at compile-time. However, in this work we look at extending this concept so that templates may be loaded dynamically into a running sys-

tem, not unlike how classes may be loaded dynamically in languages like Java. We discuss how this can be achieved with template extensions and with parameterized templates.

Being able to load classes dynamically into a system is useful since it allows extensions to be written after the system has started running. It also allows one to configure and reconfigure a running systems and for a system to configure itself based on discovery of its environment.

Dynamically loading classes in a controlled type-checked way has advantages over other dynamic linking mechanisms. A compiler and loader will together have checked that the loaded class can be used in a type safe way. Doing this at the levels of templates, each containing several related classes, extends the reach of this checking.

The mechanism proposed here for dynamically loading templates in PT preserves the relation between the classes in the template and thereby supports family polymorphism [9]. It has the advantage of the flexible adaption and name change mechanisms of PT while still being dynamic. Since new types are created when a template is loaded, different versions of the same software package can safely be used simultaneously in the same runtime without name clashes. Thus, PT extended this way becomes more a mechanism of dynamic composition than a static mechanism of reuse and extension.

We will first present an overview of the basic package template mechanism. Then we will present templates that extend other templates and template parameterized templates. After that, we will present dynamic loading before a discussion and a survey of related work.

2. OVERVIEW OF THE PACKAGE TEMPLATE MECHANISM

We here give a brief and general overview of the package template mechanism as it is described in [15]. The mechanism is not in itself tied to any particular object-oriented language, but the examples will be presented in a Java-like syntax, and the forthcoming examples will be based on Java.

A package template looks much like a regular Java package, but the classes are always written in one file and a syntax is used where curly braces enclose the contents of the template, e.g. as follows:

```
template T {
  class A { ... }
  class B extends A { ... }
}
```

Valid contents of a template (with a few exceptions) are also valid as plain Java programs. As such, templates may also be type checked independently of their potential usage(s).

In PT, a template is instantiated at compile time with an `inst` statement like below. This has some significant differences from Java's `import`. Most notably, an instantiation will create a local copy of the template classes, potentially with specified modifications, within the package where the `inst` statement occurs.

```
package P {
  inst T with A => C, B => D;
  class C adds { ... }
  class D adds { ... } // D extends C, see text
}
```

In this example, a unique instance of the contents of the package template `T` will be created and imported into the package `P`. In its simplest form, the `inst` statement just names the template to be instantiated, e.g. `inst T`, without any other clauses. In the example above the template classes `A` and `B` are also renamed to `C` and `D`, respectively, and expansions are made to these classes. Expansions are written in `adds`-clauses, and may add variables and methods, and also override virtual or implement abstract methods from the template class.

An important property of PT is that everything in the instantiated template that was typed with classes from this template (`A` and `B`) is *re-typed* to the corresponding expansions (`C` and `D`) at the time of instantiation (PT rules guarantee that this is type-safe). Any sub/super-type relations within the template is preserved in the package where it is instantiated. Therefore, implicitly `D` extends `C` since `B` extends `A`.

Another important property is that classes from different, possibly unrelated, templates may also be *merged* to form one new class, upon instantiation. Consider the simple example below.

```
template T {
  class A { int i; A m1(A a) { ... } }
}
template U {
  class B { int j; B m2(B b) { ... } }
}
```

Consider now the following usage of these templates:

```
inst T with A => MergeAB;
inst U with B => MergeAB;

class MergeAB adds {
   int k;
   MergeAB m2(MergeAB ab) { return ab.m1(this); }
}
```

These instantiations result in a class `MergeAB`, that contains the integer variables `i`, `j` and `k`, and the methods `m1` and `m2`.[1] Note that both `m1` and `m2` now have signatures of the form `MergeAB` → `MergeAB`.

Summing up, some of the useful properties of PT are: It supports writing reusable templates of interdependent, cooperating classes which may be statically type checked without any information about their usage. Upon instantiation, a template class may be customized, and merged with other template classes. References within a template to a template class will be re-typed according to the instantiation. After all the instantiations, a PT program will have a set of regular classes with single inheritance, like an ordinary Java program.

[1]The handling of potential name clashes resulting from a merge is beyond the scope of this article, but PT has rules and the rename mechanism discussed to deal with this

3. TEMPLATE EXTENSIONS AND TEMPLATE PARAMETERS

In this section we propose an extension to PT where templates may have bounded template parameters. Dynamic loading of templates will be presented in the next section.

To be able to enforce a bound on template parameters, we also introduce the concept of a template *extending* another, and thus also of *sub-templates*. A skeleton of a parameterized template may look as follows.

```
template W <template S extends U, template T extends V>
{ ... }
```

`S` and `T` are template parameters and `U` and `V` are statically known templates that serve as parameter bounds for the respective parameters.

When a parameterized template is instantiated, one must provide an actual template for each parameter, which must be an extension of the bound of the formal parameter.

Below is a template called `Ext`. This template could be part of a framework and programmers would be supposed to write extensions to it in order for their code to use the functionality of the framework. Other templates in the framework (like `Use` below) may have parameters bounded by the template and may hence be instantiated with a programmer's sub-templates of `Ext` as parameters. The template `Ext` and a sub-template `ExtOne` may look as follows.

```
template Ext {
  class A { void m1(B b) { ... } }
  class B { void m2(A a) { ... } }
}
template ExtOne extends Ext {
  class A adds { ... }
  class B adds {
    void m2(A a) { ... } // redefines m2
    void m3(B b) { ... } // new method m3
  }
  class C { ... }        // new class C
}
```

There may be an open ended number of extensions to a template, written separately and without knowledge of each other. The extensions may override method implementations and add methods and properties to classes, and they may also add new classes and instantiate other templates. Extension templates can only extend one template, and there is an implicit instantiation (`inst`) of the extended template within the extension. For now, we will not consider the possibility of allowing name changes in extension templates.

Below, the template `Use` is defined with a template parameter `E` bounded by `Ext`. The template parameter can be used in an `inst` statement in `Use`, but the actual instantiation is postponed until an actual parameter is provided.

Within `Use`, it is known from the bound `Ext` and the `inst` statement that the template will have at least the classes `A` and `B` from `Ext`, and they may be used in `Use` in the same way as classes from a regular `inst` statement.

Thus, through the use of `adds`-clauses (such as `A` and `B` below), the parameterized template may add fields and methods to the classes defined in the parameter bounds, as well as override methods from these classes. Furthermore, it may instantiate other templates using a normal `inst` statement, and add its own classes (such as `C` in the example below).

```
template Use<template E extends Ext> {
  inst E;
  class A adds { ... }
  class B adds {
    void m2(A a) { ... } // redefines m2
    void m3(B b) { ... } // new method m3
  }
  class C {              // new class C
    void foo(B b) {
      new A().m1(b);
    }
  }
}
```

In a program, the template `Use` can be instantiated with `ExtOne` as its actual parameter, as shown in the example below. In the package[2] `Program`, the contents of the parameterized template and of the actual parameter are statically known, and make up the available classes (and interfaces) accessible from the instance of `Use<ExtOne>`. For these classes, additions may be supplied in the normal PT manner, as shown below for `A`, `B`, `C` and `D`. Other templates may be instantiated as well, and merging may be performed as for normal PT instantiations.

```
package Program {
  inst Use<ExtOne> with Use.C=>C, ExtOne.C=>D;
  rename ExtOne.B.m3=>m4;
  class A adds { ... }
  class B adds { ... }
  class C adds { ... }
  class D adds { ... }
  class E { ... }
}
```

Both the actual parameter (here `ExtOne`) and the parameterized template (here `Use`) may have a class that overrides a method in a class defined in the template bound `Ext` (like `m2` above). In that case, the general rule is that changes from the parameterized template (`Use`) override changes from the extending template (`ExtOne`). This rule fits into a programming pattern where it is the programmer of the parameterized template who is in charge and wants to use the method in the parameterized template regardless of the extension to the base template. However, we envision that there might be a need for users to change this precedence, but we explicitly leave that topic open for future work.

A similar issue is the question of what should happen when the extension (actual parameter) and the parameterized template both have defined new classes with the same name (like `C` above). In such situations, these new classes are considered to be separate classes, and must be renamed in the package `Program` in the regular PT fashion to avoid ambiguity (as in the example above). The same goes for methods within an existing class (like `m3` above). A regular package may also add classes (like `E`).

Below is an example that illustrates some of the different situations that might occur with regards to method overrides.

[2]Like templates, packages are written within curly brackets in PT.

```
template U {
  class A { void m(){ ... } }
  class B extends A { void m(){ ... } }
}
template V extends U {
  class A adds { void m(){ ... } }
  class B { void m(){ ... } } // extends A implicitly
}
template X <template UU extends U> {
  class A adds { void m(){ ... } }
  class B { void m(){ ... } } // extends A implicitly
}
package P {
  inst X<V>;
}
```

Package `P` will have classes `A` and `B`, both with a method `m`. The methods will be the ones from template `X`, since they override the others. As with regular single inheritance a call to `super` in `B` will invoke the method defined in `A`. If, on the other hand, `A` in `X` did not define an override of `m`, a call to `super.m` in `X.B.m` would call the implementation in `V.A`.

However, if one wants to reuse the methods that are overridden by the template mechanism as opposed to by ordinary class inheritance, the keyword `tsuper` may be used, and a call to `tsuper` in `A` in template `X` will call the method defined in `A` in the template that is given as the actual parameter (in `P` this is `V`). A call to `tsuper` in `A` in template `V` will invoke the one in `A` in template `U`. Combining `super` and `tsuper` yields a useful and flexible mechanism for reuse.

A package can be used as a regular Java package in other compilation units and its classes are regular classes. A regular Java class may, for example, refer to (and import) the class `P.A`. We will see later that regular classes can also have template parameters, and these work in a different way.

We have not worked out rules for visibility or access restriction at the package level yet. Hence, there is no mention of public or private classes or methods.

There are obviously many other questions around templates with template parameters that are not fully answered in the text above, but to keep this exposition fairly short, we will not pursue all of these questions here.

4. DYNAMIC INSTANTIATIONS

We saw in Section 2 how to instantiate templates, and how to merge classes from different templates by using the compile-time `inst` construct. In section 3 we saw how templates can have template parameters and how to write sub-templates. In this section we introduce *dynamic* instantiation and adaption of templates. We believe this is very useful, as which features (in the form of templates) should be used is often not known until after the execution has started. For simplicity and to keep this short we limit our detailed discussion here to instantiating templates dynamically without any name changes or merges.

The approach we use is based on the hierarchies of templates formed through the *extends*-relation, and on using this somewhat like the hierarchies of subclasses in traditional object-oriented programming. Thus, we can type e.g. a variable with a *template based type*, and it can thereby refer to instances of that template, or to instances of a sub-template.

However, we shall also use template types in a way that is somewhat unusual, by saying that each *template instance* has a type of it own which includes both the template of which it is an instance, and the identity of the instance. Thus, two instances of the same template have different types. This is done to make it easier to handle the fact that the "same" local class in different instances of a template are indeed different classes. To form a consistent model, we also say that the full type of an instance is a subtype of the template it is an instance of.

In the following discussion, we will use as an example the three templates below. Templates `U1` and `U2` can be written after a program referring to `U` has started running. They can be separately compiled and neither of `U1` and `U2` need any knowledge of the other (nor does `U` need any knowledge of its descendants, obviously).

```
template U { class A {...} class B {...}  }
template U1 extends U {
  class A adds {...}   class B adds {...} }
template U2 extends U {
  class A adds {...}   class B adds {...} }
```

In this context, `U` can often be seen as a sort of "template interface", providing mostly abstract classes (in a template sense), and `U1` and `U2` can then be different implementations of this interface. At runtime, a program referring to U may load one of the templates `U1` or `U2` (or further sub-templates of these) and create one or more instances of it. Such instances can be kept track of by template-typed variables and can be passed around by assignments etc. according to normal object-oriented polymorphism rules, e.g as the following code.

```
Instance<instance ? of U> u = /* A dynamically
            generated instance of U, U1 or U2 */;
Instance<instance ? of U1> u1 = /* A dynamically
            generated instance of U1 */;
u = u1;
```

Here, `Instance<T>` is a class much like the class `Class` in Java. It is parameterized by an instance type `T` and has the signature `class Instance <instance T>`. It represents a template *instance* (and not a template) in the same way that `Class` represents a class. The reason that we use a class that represents the instance and not the template is that every instance generation results in the creation of a new instance type and new types for all the classes in the instantiated template. The concrete mechanism for performing a dynamic load and instantiation will be explained shortly, but the result is an object of the class `Instance<T>`. The special syntax `<instance ?  of U>` tells the compiler that the exact instance is not known statically, but that it is a sub-template of `U`. The parameter `T` of `Instance` will be bound to the type of the instance. As is shown in the last line above, template instance references may be assigned to a template variable having a more general type. Also, an obvious form of casting can be used for the opposite case.

In the program, a method can have template parameters bounded by U in the following way:

```
<instance T of U> void method(){
  T.A a = new T.A();
  a.doStuff();
}
```

Within such a parameterized method, elements can be typed with classes from the template using the template type as a prefix, like `T.A` above. Code like this makes it possible to use classes and invoke methods in dynamically instantiated templates in a type-safe way. Type safety depends on the fact that, in the scope, `T` is bound to an instance and `T` (a type parameter) does not change like object variables.

We are considering whether a syntax like the following should be allowed:

```
Instance<instance ? of U> u = /* A dynamically
               generated instance of U1 or U2 */;
...
method<u.TYPE>();
```

Here, the formal parameter `T` in the method is bound to the current type of `u` (which is identical with the instance identity), and will remain so throughout the body of the method. This will fail if `u` is a `null` value.

A class can be written with the same kind of template parameter. This can look as follows:

```
class P <instance T of U> {
  public T.A a;
  public P(){
    ...
    a = new T.A();
    ...
  }
}
```

This class can be statically and separately checked in the same way that a generic class can be type checked, with the difference being the *dot-named* classes, like `T.A`. `T` will be the same type in this scope and `T` will be an instance of `U` or of a subtype of `U`. Thus, the class `T.A` can be seen as a type just like any type and it has all the properties of `A` in `U`. Note that if `A` has a method `void m(B b)` it can be invoked with `m(new T.B())` here. `T.A` and `T.B` will, since `T` is the same, be from the same instance and at runtime the actual type of `B` will match up with the method signature.

The class `P` can then be used e.g. as follows:

```
Instance<instance ? of U> u = /* Instance of U or of a
                        sub-template */;
P<?> pu  = new P<u.TYPE>();

...   // Maybe another assignment to u

P<?> pu1 = new P<u.TYPE>();

pu = pu1;       // OK
pu.a = pu1.a;   // COMPILE TIME ERROR !
```

Here, `P<?>` is a type where `?` is similar to `?` in Java generics in that `pu` can point to any object of `P`. Similarly, `u` can point to an instance class for any instance of `U` or instance of a sub-template of `U`. Since it is not known statically in this scope if `pu` and `pu1` point to an object created with the same template instance (because of the question mark), the last assignment is not legal.

Objects of the classes of a template instance may be passed around like the template instance itself. This can be illustrated by the following two methods:

```
<instance T of U> void method_1() {
  T.A a1 = new T.A();
  T.B a2 = new T.B();
  method_2<T>(a1, a2);  // <T> may be omitted as it
}                       // can be inferred

<instance Z extends U> void method_2(Z.A a1, Z.B b) {
  ...; a1.m(b); ...;
}
```

When `method_1` is invoked, `T` is bound to the type of some template instance `u`. The clause `T.A` is bound to the type of `A` for that particular instance. The second method takes two formal parameters that are of types called `Z.A` and `Z.B` where `Z` is a the type of the template instance (a sub-type of `U`). The invocation of this method in the first method can be type checked since both the actual parameters are typed with class `A` from the same template which is also a sub-template of `U`. Inside the second method, the methods (for example `m`) defined in template `U` can be called.

In a scope, there is often only one known template that is being used and it would be nice not to have to write the instance type parameter `T` all the time. Therefore, we propose the shorthand notation shown below. Within the `with`-block, class names can be written without the type prefix.

```
<instance T of U> void method(){
  ...
  with(T) do {
    ...; .A a = new .A(); ...;
  }
  ...;
}
```

Other times, one may want to work with two (or more) different instances of a template (or more likely, two instances of different sub-templates) at the same time. Below we assumes that `A` in `Ext` has a field `b` of type `B` and that `B` has a field `x` of type `int`.

```
<instance T of Ext, instance U of ExtOne>
                    U.A[] method(T.A tas[]) {
  U.A uas[] = new U.A[tas.length];
  for (int i = 0; i<tas.length; i++){
    uas[i] = new U.A();
    uas[i].b = new U.B();
    uas[i].b.x = tas.[i].b.x; // A and B from different
  }                           // instances are never
  return uas;                 // mixed up
}
```

Dynamically generated instances of templates are produced by a special loader, with a method `instantiate` that has a template parameter, and a normal `String` parameter. The first parameter should be a statically known template, and the second should be a filename (or net-address, etc.) where a sub-template of the template parameter can be found. The loader will check that this is the case, and maybe also compile the template if necessary. Thus, a dynamic instantiation may look as below. The exact details of the loader and its implementation are not worked out, but at runtime it can be checked that it will only return an instance of the given template or a sub-template.

```
Instance<instance ? of U> u =
                TLoader.instantiate<U>("-file-");
```

Just as metods and classes in regular classes can be parameterized with regular template instances and used with any

instance of any sub-template, they can also be parameterized with a parameterized template. The example below is a class that uses the template `Use` from earlier as a parameter bound.

```
class StartOff<instance T of Use>{
  run(){ ...
     new T.C().foo(new T.B());
     ...  }
}
```

An object can be created of this class using any instance of `Use` instantiated with any sub-template of `Ext`. All the classes known in `Ext` can be used within this class, prefixed by `T`. Below is an example of instantiating an instance of `Use` with `ExtOne` and using `StartOff`.

```
Instance<instance ? of Use> u =
          TLoader.instantiate<Use>("-Use<ExtOne>-");
StartOff<?> s = new StartOff<u.TYPE>();
s.run();
```

Note that the use of the name `C` in `StartOff` refers to the one originating in `Ext` and that the one from `ExtOne` is not visible in `StartOff`.

All these examples of regular classes and methods parameterized by templates are mainly there to be starting points for the code within the templates. The interesting code will probably be inside templates `Ext` and `Use` and classes like `StartOff` will usually just set this off.

There are more details about dynamic instantiations of package templates that have not been discussed here and open questions obviously still exist, but we nevertheless believe that the mechanism should be useful as a starting point for developing a language mechanism for dynamically configured and composed systems.

5. DISCUSSION AND FURTHER WORK

The aim of this work is to develop tools that allow parts of larger software systems to be written as independent pieces and that can be merged in a flexible way. This should provide flexibility in both organization of a system and reuse of components. To be truly flexible, merging and composition of independently written parts of software should even be allowed at runtime. However, one would like to do this with some sort of static checking to avoid some of the often occurring errors with uncoupled code. Also, there is value in the ability to run different versions of a library or framework in the same runtime without having to deal with conflicting types. We try to solve this with an extension of the package template (PT) mechanism.

PT and the extensions proposed here should provide some of the apparatus not only for merging and composing unrelated templates of cooperating classes, but also for doing so in a running system.

One feature of the PT approach is that the classes within the templates form a whole and that the relations and inheritance between the classes are preserved during instantiation.

New types are created whenever a template is instantiated. Each instance is kept separate, and in addition to allowing flexibility in merging and renaming, this can be used to make sure that objects created from different template instances have their own type. This enables a form of family polymorphism [9].

The dynamic loading and composition proposed is not as dynamic as some other systems. Requiring that the loaded template be a sub-template of some bound, the program can be type checked and if the loading itself does not fail, the system will not cause type errors during execution.

Loading single classes, as is usual in Java, means that one can only depend on a single interface with named methods that have parameters that are of statically known and unchangeable types. Loading a complete template not only allows one to view several classes as a whole, but in PT the types of the parameters of methods and variables in the template itself are adapted to new types. This is a useful new feature of package templates and dynamic package templates in particular – which does this composition at runtime.

PT is more flexible in renaming, etc, than is discussed here. There is also more flexibility in using template parameters than what has been discussed. There is work going on looking at merging and other type safe mechanisms for creating instances from multiple dynamically loaded templates and on abstract methods in templates.

We have not discussed how this proposal would work together with the aspect oriented extensions discussed in [3]. A consistent combination of the two extensions should be worked out. For an even more dynamic approach, there is also a study of a similar mechanism for the dynamically typed language Groovy [2].

We are also looking at doing dynamic updates to running code, that is updating already loaded template instance, by for example redirecting calls to new methods. Allowing this in some restricted way, could create even more useful aspect oriented features in the language.

The most important future work is to settle the open questions concerning the rules of the language, prove its type safety and building a compiler.

6. RELATED WORK

The authors of the trait mechanism [22] approach the problem of composition from the angle that the primary unit to be composed is the class. A trait is as such a construct that encloses a stateless[3] collection of provided and required methods. Traits may subsequently be used to compose new traits or as part of a class definition. The composition of traits is then said to be *flattened*. This entails that (1) the trait ordering in each composition is irrelevant, and (2) that a class composed from traits is semantically equal to a class in which all the methods are defined directly in the class. When used to compose a class, all requirements must be satisfied by the final composition. Traits were originally developed for the dynamic language Squeak, and supports method aliasing and exclusion upon composition. A statically typed version also exists [20]. Still, neither the static

[3]Traits were originally defined to be stateless, although a more recent paper [5] has shown how a stateful variant may be designed and formalized.

nor the dynamic version have explicit support for *runtime* selection of which features that should be composed.

Mixins [6] are similar in scope to traits, in that they target the reuse of small units of code. Mixins also define provided and required functionality, and the main difference between them and traits is arguably the method of composition. Mixins traditionally rely on inheritance, by defining a subclass with as-of-yet undefined parent, and thereby requiring that mixins are linearly composed.

Functionality similar to that of traits and mixins can quite easily be mimicked with PT. For instance, to create a reusable collection of methods (with or without accompanying state), one might simply define a template with a single class, consisting of the methods that are subject to reuse. This class may then be merged with other classes where the functionality is needed. When it comes to specifying required methods, PT provides no such concept out-of-the-box, but a solution might be to define abstract and/or virtual methods in the template class. As is the case with traits, merge/composition order is not significant in PT.

Perhaps the biggest conceptual difference between mixins/traits and PT comes in form of intended scope, in the sense that PT is targeted towards reusing and specializing larger chunks of code as one coherent unit. In that regard, the former two can be seen as a special case of what can be accomplished with PT, admittedly with a slightly more involved syntax and some 'glue code'.

Aspect-oriented programming (AOP) [14] involves several concepts related to PT. For instance, intertype declarations in AspectJ [7] may (statically) add new members to existing classes, and may as such be used to compose previously unrelated features. An example of this is exemplified through the Observer design pattern [10] in [11]. However, this implementation, and on a higher level the general approach employed to composition, is arguably less than optimal, given that it suffers from the fact that the aspect itself entangles several conceptual roles within a single aspect, and that this aspect also exists as a unit at runtime, lacking a clear mapping to objects from the problem domain. The Caesar language [1, 19] supports both aspect-oriented programming constructs and code reuse and specialization through the use of virtual classes. It also supports wrappers for defining additional behavior for a class, and dynamic deployment of aspects at runtime (through use of the `deploy` keyword). Dynamically deployed aspects are in effect from all calls propagating down the call stack with respect to the lexical scope of a `deploy` construct. Expanding on the notion of dynamic deployment, Tanter [24] describes a mechanism for controlling the scope of dynamically deployed aspects (including propagation down the call stack and to new objects). Note, however, that these aspects may affect behavior only, and not class structure or hierarchy, as opposed to dynamic instantiations in PT.

Context-oriented programming (COP) [8] provides a way to activate and deactivate *layers* of a class definition at runtime. Layer activation can be nested, and propagate down the call stack (for the current thread).

Like PT, Mixin layers [23] is a mechanism for writing an addition with affect accross multiple entities like classes. Mixin layers can be composed by instantiating a layer with another as its parameter and thus mixin layers are both reusable and interchangeable. They are also nested. However, there does not seem to be a way to build hierarchies withing a mixin layer.

BETA [18, 17], gbeta [9] and J& [21] (pronounced "jet") are systems that in many ways are similar to each other and in many respects can achieve similar end results to those of PT. A common property of all of them (except PT, that is) is that they utilize virtual classes (as introduced by BETA) to enable specialization and adaption of hierarchies of related classes. gbeta and J& support multiple inheritance, and this may to a certain extent be used to "merge" (in the PT sense of the word) independent classes. Neither BETA, gbeta nor J& support concepts similar to runtime template instantiations.

As we now introduce dynamicity and more free-standing template instances, the mechanism we present will become more similar to a solution with virtual classes and family polymorphism, as e.g. in gbeta [9]. However, the rules and restrictions used to keep the system consistent will be different in our version.

In a subject-oriented [13] programming (SOP) system, different subjects may have differing views of the (shared) objects of an application. There is no global concept of a class; each subject defines 'partial classes' that model that subject's world view. What is called a *merge* in SOP, is somewhat different from a merge in PT. In SOP, a merge is an example of a composition strategy (and there may be many of them), that tells the system how to compose separate subjects with overlapping methods and/or state. Like with mixins and traits, there seems to be a difference in intended scope when comparing SOP with PT; SOP targets a broader scope, with entire (possibly distributed) systems (that may even be written in different languages) being composed. One could, however, picture an extended PT-like mechanism as a basis for an implementation of SOP.

Our approach to typing classes and methods with instance types to keep different instances of a template apart at runtime is based on a clever idea for keeping different implementations of a single API apart at runtime using Java generics and tying the classes of an implementation together using a type parameter. This idea is found in [12].

Ada originally (in 1983, [16]) had no mechanisms supporting object-orientation, but it had a mechanism called generic packages with some of the same aims as PT, in that packages can contain type definitions and that you get a new set of these each time the generic package is instantiated. Generic packages also have type parameters.

In Ada 95 [4] a slightly untraditional mechanism for object-orientation was introduced, and it was further elaborated in Ada 2005. Thus, the potential for PT-like mechanisms should be there, but as far as the authors understand it, there is nothing similar to virtual classes (at compile-time or at runtime) in the language, and the mechanims for adapt-

ing a package to its use during instantiation are not very advanced.

7. CONCLUSION

We have proposed an extension to the package template mechanism that will allow dynamic loading and instantiations of templates. We have discussed some of the properties of the proposed language. It is in some crucial ways different from other mechanisms that try to solve similar challenges. The practical consequences of these differences need to be worked out.

Further studies need to be done to find out if the language is really useful for dynamic software composition, and details of the language need to be worked out based on a study of what makes most sense in a practical language.

Although our initial work on finding a translation of the mechanism to Java generics suggests that the language is type safe, a proof of this needs to be worked out and a compiler must be built.

8. REFERENCES

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In Trans. AOSD I, volume 3880 of LNCS, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] E. W. Axelsen and S. Krogdahl. Groovy package templates: supporting reuse and runtime adaption of class hierarchies. In DLS '09: Proceedings of the 5th symposium on Dynamic languages, pages 15–26, New York, NY, USA, 2009. ACM.

[3] E. W. Axelsen, F. Sørensen, and S. Krogdahl. A reusable observer pattern implementation using package templates. In ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software, pages 37–42, New York, NY, USA, 2009. ACM.

[4] J. Barnes. Programming in Ada95. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[5] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. Computer Languages, Systems & Structures, 34(2-3):83–108, 2008.

[6] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, Proc. Conf. O-O. Prog.: Syst., Lang., and Appl. / Proc. ECOOP, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[7] A. Colyer. AspectJ. In Aspect-Oriented Software Development, pages 123–143. Addison-Wesley, 2005.

[8] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In DLS '05: Proceedings of the 2005 symposium on Dynamic languages, pages 1–10, New York, NY, USA, 2005. ACM.

[9] E. Ernst. gbeta - a language with virtual attributes, block structure, and propagating, dynamic inheritance, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns -Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[11] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. SIGPLAN Not., 37(11):161–173, 2002.

[12] W. Harrison, D. Lievens, and F. Simeoni. Safer typing of complex api usage through java generics. In PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, pages 67–75, New York, NY, USA, 2009. ACM.

[13] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, pages 411–428, New York, NY, USA, 1993. ACM.

[14] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, Proceedings European Conference on Object-Oriented Programming, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[15] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. Journal of Object Technology, 8(7):59–85, 2009.

[16] H. Ledgard. Reference Manual for the ADA Programming Language. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[17] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications, pages 397–406, New York, NY, USA, 1989. ACM.

[18] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, 1993.

[19] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In AOSD '03, pages 90–99, New York, 2003. ACM.

[20] O. Nierstrasz, S. Ducasse, S. Reichhart, and N. Schärli. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.

[21] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In OOPSLA '06, pages 21–36, New York, 2006. ACM.

[22] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In ECOOP 2003, volume 2743 of LNCS, pages 327–339. Springer Berlin / Heidelberg, 2003.

[23] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Softw. Eng. Methodol., 11(2):215–255, 2002.

[24] É. Tanter. Expressive scoping of dynamically-deployed aspects. In AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development, pages 168–179, New York, 2008. ACM.

ReCycle: Resolving Cyclic Dependencies in Dynamically Reconfigurable Aspect Oriented Middleware

Bholanathsingh Surajbali, Paul Grace and Geoff Coulson
Computing Department,
Lancaster University
Lancaster, UK
{b.surajbali, p.grace geoff} @comp.lancs.ac.uk

ABSTRACT
In aspect-oriented middleware systems, the aspect modules are typically composed as chains of aspects within the connectors (or bindings) that join the base software components. However, this approach can lose or hide information about the dependencies between multiple aspects in the chain; this is particularly important when dynamically reconfiguring such a system at run-time. Without knowledge of these dependencies the system could reconfigure a new aspect with a dependency to a prior aspect in the chain resulting in a cyclic dependency and subsequent deadlock. Furthermore, the problem is harder to detect with the presence of remote aspects within the connectors as their dependencies are hidden across address spaces. To resolve cyclic dependencies that may occur when reconfiguring both local and remote aspects we propose the use of a *re*configuration *cycl*ic dependency r*e*solution (ReCycle) model. This approach can be employed generally in dynamic AOP middleware platforms, and in this paper we evaluate it within the AO-OpenCom middleware.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: D.2.11 Software Architectures – Languages (interconnection), Patterns.

General Terms
Design, Management

Keywords
Middleware, dependency, aspect, dynamic reconfiguration.

1. INTRODUCTION
Aspect-oriented middleware platforms provide solutions to create distributed component-based systems into which aspect modules representing cross-cutting concerns can be woven. Aspects are made up of individual code elements that implement the concern (*advices*), which are deployed at multiple positions in a distributed system (*join points*) that are expressed by *pointcuts*—a particular form of composition language. AO-OpenCom[11], AspectOpenCom[4] ,CAM/DAOP [3], FAC [9], FuseJ [13], DyMAC [5], and DyReS [14] are examples of aspect-oriented middleware which allow aspects to be composed and adapted at runtime. The aspect runtime composition of aspects in such AO middleware platforms differs from the standard component to component binding (where there is a direct reference from the provided interface to the required interface). In these AO middleware aspects are advice components which are woven non-invasively at their connector (between the required and provided interfaces of the base software components) in advice chains with the aspect reference stored in the advice chain. Then, the aspects are *invoked* from the connector chain when a call or execution occurs from the call or execution of the provided or required interface.

Unlike components, the dependency of an aspect to another aspect is not explicitly defined, such that an aspect within a chain may have a dependency with another aspect located earlier in the chain, and cause a *cyclic dependency* while performing reconfiguration. The potential problem of cyclic dependency is that it may cause the running system to enter into a *deadlock* after performing reconfiguration, when an invocation occurs at the join point. The cyclic dependency problem is hard to detect since an AO-Connector, maintains both local and remote advices. For an AO-Connector containing solely local advices, inspection of the AO-Connector can reveal the possibility of cyclic dependencies. However, this is non-trivial when the AO-Connector contains both local and remote advices, since for remote advices the visibility of the methods invoked by remote advices is located in the remote address space from where the AO-Connector is.

In this paper, we present a *re*configuration *cycl*ic dependency r*e*solution (ReCycle) model for dynamic aspect-oriented, component-based middleware; this provides the capability to describe the various kinds of built-in dependency inconsistencies that affect aspect configuration and reconfiguration at runtime. This is coupled with a graph-based tool which detects and resolves cyclic dependency inconsistencies at run-time while reconfiguration is performed.

We evaluate our approach within the AO-OpenCom platform for developing dynamic reconfigurable middleware solutions; this demonstrates the following contributions of our approach:

- *Resolution of reconfiguration cyclic dependency.* We show that cyclic dependency inconsistencies can be resolved for one case-study with minimal performance overhead.
- *Transparency.* We apply consistent reconfiguration with minimal developer effort or change to the underlying component model.
- *Flexibility.* New dependency consistency can be described dynamically to evolve with the running application or domain context without breaking the implementation details of the instantiated aspect. Moreover, the approach can be applied in different compositions approaches and tools; for example we show how both node-local and distributed reconfiguration cyclic dependency consistency can be avoided in this paper.

The remainder of this paper is organised as follows. Section 2 examines the types of aspect reconfiguration cyclic dependency that may occur. Then, section 3 describes the design of our ReCycle model, followed by section 4 which validates the proposed ReCycle model. Finally we describe related work in section 5 and offer our conclusions in section 6.

2. ASPECT RECONFIGURATION

In aspect-component middleware, aspects (which are themselves implemented as component modules) are composed with the base components (hereafter termed components) using AO-Connectors [4, 8, 11, 12, 14]. AO-Connectors are the architectural element offering aspectual composition (weaving) of aspects between a receptacle and a provided interface of components. AO-Connectors maintain the meta-data containing references to aspects instances in an advice chain. For example, it maintains details of all advised aspects and their types and allows these to be queried to determine the operations they support and the aspects currently advising them. It also supports the runtime manipulation of the chain to add a new advice, or remove or reorder aspects in the chain of advices.

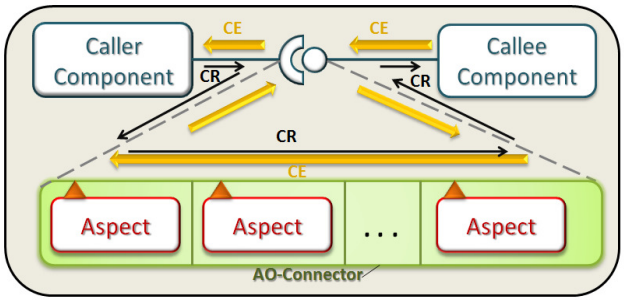

Figure 1: Aspect-Component Model

A list of advices is attached to the connector between the required and the provided interface. This capability is illustrated in Figure 1, which shows *a caller component* connected to a *callee component*, and an AO-Connector containing a list of aspects that get called. Where a call comes from the caller component (arrows marked CR) then the aspects in the chain are executed first or otherwise in case an execution is triggered from the Callee component, the aspect chain is executed in the reverse order, as highlighted with arrows marked CE) in Figure 1.

We now identify and classify the types of dependency inconsistencies that can occur in the aspect-component model.

2.1 Use case scenario

To motivate the requirement to resolve cyclic dependency for AO reconfiguration we present its occurrence within the distribution framework stack. The AO composition is as follows (see Figure 2): when the message handler is called on the communication module, the following aspects are enforced, before the execution of the communication module operation:

i.) *Selecting the format of transportation*. Format selection handles the formatting of the message such that it can be serialised and deserialised for remote invocations and replies.

ii.) *Selection of the transportation*. Transport selection creates a transport listener and transport request and binds them to a socket.

iii.) *Deployment handler for the message transfer*. The deployment handler creates the skeleton and binding for the message transfer as well extracting the object name in the URI to lookup the correct instance in case of a normal method call.

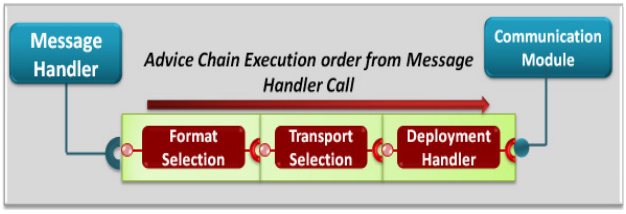

Figure 2: Distribution Stack AO Composition scenario

Whenever, the Message Handler component calls the Communication Module, the list of advices within AO-Connector chain gets invoked and executed in the following order:

Format Selection Aspect → Transport Selection Aspect → Deployment Handler Aspect.

2.2 Cyclic Dependency Occurrence

To cope with the application and environmental demand the following two dynamic (re)configurations may be required: (i) new users with limited bandwidth may join, requiring a *Compression aspect* to be configured to split data before being sent; (ii) data may be required to be encrypted using a *Security aspect* to protect the users' privacy.

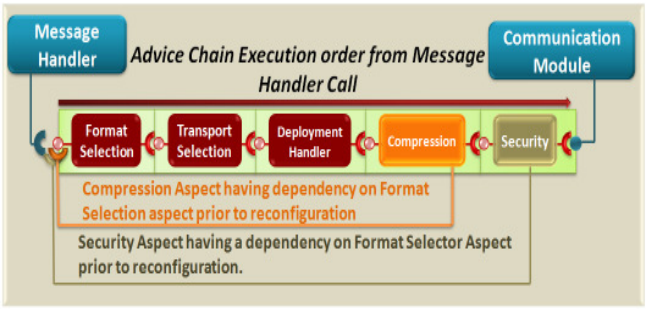

Figure 3: Reconfiguration with Cyclic Dependency Occurrence

The reconfiguration proceeds by weaving the Compression Aspect after the Deployment Handler Aspect in the AO Connector chain and the Security Aspect woven after the compression aspect in the chain, as illustrated in Figure 3. However, both the Compression aspect and the Security Aspect may have a dependency on the Format Selection aspect prior to the reconfiguration, causing cyclic dependencies to occur at the AO Connector such that calls may not return back to the Security Aspect, causing a deadlock to occur if the reconfiguration is allowed to proceed. The cyclic call dependency for Figure 3 when called proceeds as follows:

Format Selection Aspect → Transport Selection Aspect → Deployment Handler Aspect → Compression Aspect → Format Selection Aspect.

A more complicated cyclic dependency occurrence is when remote aspects are attached to the AO-Connector. In the case of remote aspects, they may have dependencies with other aspects located on different address spaces, causing the dependency to be unnoticed while performing reconfiguration.

2.3 Analysis

An aspect represents a crosscutting functionality that may be referenced and shared by other software modules in a running system. That is AO middleware typically just add/reconfigure at runtime without knowledge of the chain or taking into consideration the existing aspects dependencies that may already be present. So doing, as described above, can potentially lead to cyclic dependencies. Furthermore, creating two versions of the aspect by replicating the aspect functionality is not a feasible solution either and can potentially result in an exponential growth in versions of the same aspect. To solve the above problems, we propose the ReCycle model.

3. ReCycle: RECONFIGURAION CYCLIC DEPENDENCY RESOLUTION MODEL

In this section we describe our ReCycle model to support the detection of cyclic dependencies that may result in the configuration and reconfiguration of aspects as well as its resolution by supporting the following dimensions: (i) describing aspect dependency; (ii) attaching metadata to entities in the aspect-component model; (iii) using graph based detection with a resolution engine capable of parsing the AO-Connector to detect the occurrence of any cyclic dependency inconsistencies. Each of the dimensions is now examined in turn.

3.1 Aspect Dependency Metadata

In order to detect cyclic dependencies each aspect-component is attached with metadata that describes and explains its functionality as well as the dependency they may have on other aspect-components. This is used to inform the deployment of the aspect—i.e. to help manage compositional and reconfiguration cyclic detection between aspect-components in the aspect-component model as illustrated. These descriptions are written by the AO middleware developer in the format as illustrated in the BNF form of Figure 4.

```
aspect-instance :== <{ aspect-scope, aspect-required-interfaces, aspect-provided-interfaces }>
aspect-dependency : == <{ (dependency-aspect-instance | dependency-aspect-type),
                          dependency-aspect_scope, <AO-Connectors> }
aspect-scope  :== local | remote
AO-Connectors  :== { list of ao-connectors dependent aspects are connected}
aspect-required-interfaces :== {list of required interfaces attached with aspect-component}
aspect-provided-interfaces :== {list of provided interfaces attached with aspect-component}
dependency-aspect-instance :== { expressions describing dependency aspect instance}
dependency-aspect-type :== { expressions describing dependency aspect type }
dependency-aspect_scope :== { expressions describing dependency aspect scope }
```

Figure 4: ReCycle Model BNF Metadata representation

The *aspect-instance* defines the aspect-component instance aspect scope, list of aspect required interfaces of the aspect-component instance and list of provided interfaces for the aspect-component.

Aspect-dependency defines the list of aspect-instance aspect-type to which the aspect is dependent on as well as the AO-connector to which it is currently bound with.

The *aspect scope* refers to the aspect-component instance of whether it is deployed on the local host, or is remote.

The *AO-Connectors tag* refers to the list of connectors to which the aspect-component instance or type is bound with. This can be zero in case there is no connection dependency for the aspect-component.

3.2 Attaching Metadata

As described in our previous work in [12] tagged metadata needs to be kept separate from the main source functionality. This is because:

1. aspect-components are considered as black-boxes which provide advices in the form of operations within the provided interface (but hide their implementation);
2. aspects represent crosscutting functionality such that adding descriptions by extending the implementation, e.g. through a new interface, will restrict its applicability to different applications and domains because it couples the consistency checking with the aspect-component functionality.

Keeping metadata separate allows both the core functionality and metadata to be reconfigured independently and transparently from each other.

Metadata is attached to the aspect-component interfaces and receptacles at load-time, as they are the only access points available to other aspect-components to be inspected and inform runtime decisions. Then to provide for runtime reconfiguration, since aspect-components are *invoked* through their operations, aspect-component operations also need to be annotated. This is because when reconfiguration is performed at runtime, already woven aspect metadata might be required to detect cyclic dependencies at the join point the aspect is accessed via its operations.

3.3 ReCycle Model

A ReCycle model provides the tool to query and reason about the annotated aspect-components; and resolve possible sources of cyclic inconsistency that may result from a dynamic reconfiguration. The latter retrieves the associated aspect-component metadata as illustrated in Figure 5, by getting the annotation file path from the aspect-component and parsing the *Aspect Metadata* file (retrieved from the *Aspect Metadata Repository*) to extract respective dependencies tags for the aspect-component (structured as described by the BNF Cycle Metadata representation from Figure 4). Then, the ReCycle model builds a graph using the aspect-component instance and its dependencies if they are connected for the corresponding AO-Connector involved with the reconfiguration. After the graph is built, the graph is traversed from the root, the aspect-component contained in the first-order to the end of the graph.

In case the validation is successful the reconfigured AO-Connector, chain list is first stored in a reconfiguration repository having transactional capabilities and the reconfiguration is then allowed to proceed. However, in case of any cyclic dependency issue found, based on the composition policy, two alternative remedy actions can be taken by the ReCycle model in terms of:

either the ReCycle Configurator stopping reconfiguration from proceeding by calling the *rollback* operation to drive the system to the state prior to when the reconfiguration started by restoring the AO-Connector chain from the reconfiguration repository; or if appropriate resolution policies are specified these can be deployed by the ReCycle model and the reconfiguration can proceed (e.g. removing the cyclic connector or adding a null binder to return the call and exiting the cyclic loop). If a connector is removed or updated, the associated AO-Connector meta data is updated for the respective aspect-component (by updating the aspect component associated AO-Connector tag meta data).

Moreover, to avoid the potential occurrence of semantic interactions, the Semantic Resolution model from [12] may be called by the ReCycle model to reason about the resolved reconfiguration interaction. In case a semantic conflict is detected and no policies are defined, the reconfiguration gets aborted by calling the rollback operation. Otherwise if appropriate resolution is defined, the semantic valid reconfiguration is allowed to proceed while ensuring with the ReCycle model it does not result in any cyclic dependencies.

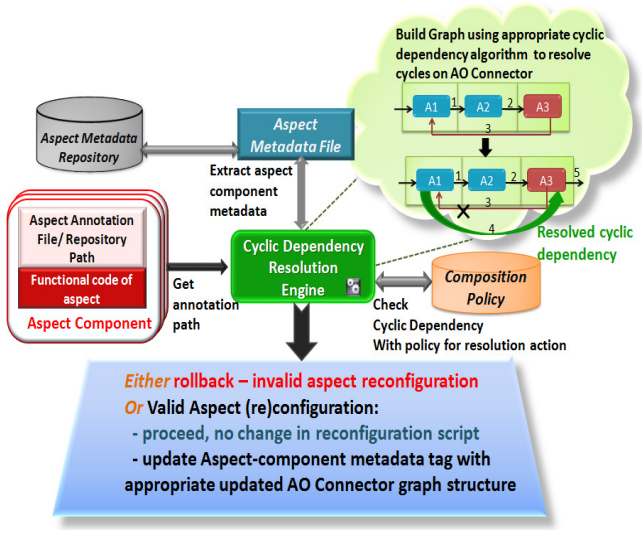

Figure 5: ReCycle model to resolve Cyclic Dependency

4. VALIDATION

In this section we validate our approach using AO-OpenCom [11]. We first provide some background on AO-OpenCom and then validate the extent to which our ReCycle model achieves the stated goals of cyclic dependency resolution, transparency and flexibility. Finally we measured the performance and resource overhead of deploying the ReCycle model.

4.1 AO-OpenCom

The purpose of AO-OpenCom is to build on OpenCom and its associated reflective meta-models and component frameworks architectures [2], to provide a distributed AO composition service, and to allow aspectual compositions to be dynamically reconfigured. The programming model employs components to play the role of aspects—i.e. an aspect is simply an OpenCom component. The AO-OpenCom aspect framework comprises a set of components that are instantiated across each host. The set of components is as follows (see Figure 6):

The **Configurator** manages the other components in the framework as it is responsible for accepting and handling *(re)configuration requests* that will apply to a set of hosts. The Configurator also caches join point information it receives from Pointcut-Evaluators in case similar behaviour needs be applied in the future. The **Aspect-Repository** holds a set of instantiable aspect-components e.g. the cache aspect, encryption aspect, etc.

The **Pointcut-Evaluator** evaluates the pointcuts provided by the Configurator and returns a list of the matching join points found within the local address space. Finally, the **Aspect-Handler** acts on instructions from the Configurator to weave advices at join points as well as supporting the invocation of remote aspects.

The main API provided by an AO-OpenCom-enabled instance for AO (re)configuration is as follows:

Configurator.reconfigure(pc, command, aspect);

The *pc* argument specifies a pointcut that picks out the join points in the target nodes at which the desired reconfiguration should occur. The *command* argument offers options for the action to be taken at the indentified join points: the 'add' action is used to weave the specified aspect at the join points; 'remove' is used to remove it, and 'replace' is used to add the specified aspect after removing an existing aspect of the same type that is assumed to be already there. The *aspect* argument can be a direct reference to a local aspect-component, or an indirect reference to an aspect stored in an Aspect-Repository, or a reference to an already-instantiated remotely-accessible singleton aspect. The *aspect weaving order* and the *type* of aspect in terms of (before, after, around) are also specified in the aspect argument.

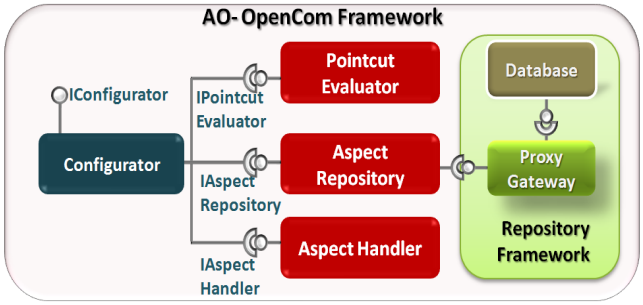

Figure 6: AO-OpenCom platform Architecture

4.2 Applying the ReCycle Model to AO-OpenCom

To ensure semantic consistency, the ReCycle model and the Composition-Policy modules are both encapsulated as aspects and woven at the AO-connector component join point connecting the Configurator and the pointcut evaluator component as an 'after' advice in the AO-OpenCom platform. Moreover, the *Aspect Metadata* file of the ReCycle model is implemented in an XML file with each aspect annotated with the path to the XML metadata file.[1]

4.3 Qualitative Validation

To illustrate the ReCycle model preserving reconfiguration consistency, we consider the use case scenario reconfiguration. To

[1] Since AO-OpenCom also supports remote aspects [11], the respective URL path to the XML file *Annotation Metadata Repository* is provided for remote aspects.

perform the reconfiguration outlined in Section 2.1, the application developer would provide a reconfiguration request by writing code as shown in Figure 7 (the code is simplified for presentational purposes).

The *Configurator.reconfigure()* call takes the given pointcut and aspect specifications and also specifies that the specified aspect should be *added.* This reconfiguration specification however fails to capture the cyclic dependency by adding the two aspects at the AO Connector as shown in Figure 3.

```
Pointcut pc = new Pointcut ("*", "Communication-Module", "ICommunication", "communication*");
List aspectList = new LinkedList();
aspectList = new ArrayList();
Aspect aspectCompression = new Aspect(Compression, after);
Aspect aspectSecurity = new Aspect(Security, after);
aspectList.add(aspectCompression);
aspectList.add(aspectSecurity);
Configurator.reconfigure(pc, add, aspectList);
```

Figure 7: Aspect Reconfiguration specification example

4.3.1 Resolution

The Security and Compression aspects in the AO-OpenCom *Application Repository* is tagged with appropriate metadata describing its dependencies on other aspect-components, that is: the Security and Compression aspects interface is tagged with the location path of the xml file containing the metadata having the *aspect-dependency tags* specifying a corresponding Compression and Security aspects each have a dependency with the Format Selection aspect and with an active AO-connector.

When *Configurator.reconfigure()* is called on the Configurator of one of the nodes (referred as the 'initiator'), the latter calls the Pointcut-Evaluator to locate all the target join points. On returning the located join points, the ReCycle aspect gets *invoked.* The latter evaluates the AO-Connector to build a *aspect dependency graph* and using the annotation metadata from the Format Selection aspect, the graph is updated to detect any cyclic dependencies that may occur.

In this case, a cyclic dependency is detected such that the Cyclic Resolution Dependency Engine checks with the Composition Policy or any 'condition-action' policies to resolve such a cyclic dependency.

The Composition-Policy aspect, as illustrated in Figure 8 specifies the 'condition-action' rules in terms if a cyclic dependency is located and aspect-instance is Security aspect, and the latter aspect has a dependency connection with a Format Selection aspect, then the connection needs to be removed, as the messages format are already set. (Otherwise if the connector cannot be removed based on the Composition-Policy specification then the reconfiguration is aborted to avoid the occurrence of cyclic dependency.)

The Cyclic Resolution Dependency Engine aspect then instructs the AdviceHandler to remove the AO-Connector connecting the Security with the Format Selection aspect, thus resolving any potential cyclic dependencies issue for this reconfiguration scenario. In case remedy policies were not specified, the reconfiguration would be aborted with the rollback operation deployed for any changes.

```
<policy name="security-formatSelection" reconfiguration-action="add">
    <condition>
        <cyclic-dependency aspect-instance="security" condition-logic="and"
                connector-instance="bound" aspect-dependency="format-selection" />
    </condition>
    <action>
        <cyclic-dependency-action connector-instance="unbound"
                aspect-dependency="formatSelection" />
    </action>
</policy>
<policy name="compression-formatSelection" reconfiguration-action="add">
    <condition>
        <cyclic-dependency aspect-instance="compression" condition-logic="and"
                connector-instance="bound" aspect-dependency="format-selection" />
    </condition>
    <action>
        <cyclic-dependency-action connector-instance="unbound"
                aspect-dependency="formatSelection" />
    </action>
</policy>
```

Figure 8: Composition Policy Example

4.3.2 Transparency

The approach naturally supports a *selectively transparent* approach as the ReCycle aspect and the Composition-Policy aspect can be pre-configured at application start-up time so that the application developer who wishes to initiate a run-time reconfiguration needs only to make the appropriate call to *Configurator.reconfigure()*. This achieves complete transparency of consistency-related mechanisms from the code to invoke a reconfiguration. At the other extreme, the developer can be explicit specifying the ReCycle and Composition-Policy aspects should be put in place for each reconfiguration. In this case, both aspects are woven on-the-fly (if they are not already present) before proceeding to perform the requested reconfiguration. Note that this extreme is still *partially* transparent as the developer is protected from the low level details of actually weaving ReCycle.

4.3.3 Flexibility

The use of a separate *Aspect Metadata* file to attach dependencies of the aspect-components allows new metadata updates to be applied without having to recompile existing source-code. Moreover, our approach adds the ReCycle as an independently-deployable service which can be used for both local and distributed reconfiguration. This means that ReCycle imposes no overhead when it not used, and can be dynamically woven/unwoven where and when required. We also believe that the approach, being based upon applying metadata and behaviour at common architectural elements (i.e. interfaces), can be applied generally to other AOM not just AO-OpenCom; indeed we see important future work in the deployment of our model in a wider range of systems.

4.4 Overhead of ReCycle

We next evaluate the overheads incurred by ReCycle to perform dynamic reconfiguration. The baseline for our experiments is as follows; we reconfigure aspects at one join point using AO-OpenCom without ReCycle (in this case there are no cyclic dependencies to detect). This was performed as follows:

- the compression aspect and security aspect both instantiated on a local aspect repository;
- the compression aspect instantiated on the same local node as the join point (AO-Connector) and the security aspect instantiated on a remote node;

- both the compression aspect and security aspect instantiated on separate remote nodes from the join point.

Each node ran on a separate Core Duo 2 processor 1.8 GHz PC with 2GB RAM, using the Java-based version of the AO-OpenCom platform. Each measurement was repeated ten times and the mean value was calculated to discount anomalous results. The cyclic dependency algorithm used is the *single-source negative-weighted acyclic-graph shortest-path algorithm* [6] and the results of the experiment are shown in Table 1.

It can be observed that on a single node the use of ReCycle added an average overhead of 5.6% when no conflicts where managed; there was an extra 8 % when aspects with a cyclic were woven on the node. The overhead of the ReCycle is mainly attributed to the use of XML and the parsing of the file structure before the proper metadata are retrieved, which accounts for the extra overhead of using ReCycle when detecting cyclic dependency.

Table 1. Overhead of using ReCycle in AO-OpenCom

Reconfiguration:	Reconfiguration Time in (ms)		
	Setup A	**Setup B**	**Setup C**
Without ReCycle	390	1356	2651
With ReCycle with no cyclic dependency	411	1432	2810
With ReCycle with cyclic dependency	*442*	*1541*	*3024*

Setup A – Security and Compression Aspect woven locally.
Setup B – Security as remote aspect and compression as local aspect.
Setup C – Both Security and Compression woven as remote aspects.

A final point to note is that overhead of the ReCycle is determined by the cyclic graph detection algorithm. An optimised algorithm detection could be used to reduce the induced overhead of ReCycle in detecting cyclic dependency.

5. RELATED WORK

There are several cyclic dependency algorithms developed to detect cyclic cycles among software modules at runtime. JooJ [7] checks source code of java classes to detect for cyclic dependencies among java classes. However, JooJ requires the developer intervention to resolve the occurrence of any detected cyclic dependencies. Our approach differs from Jool in that the reconfiguration is entirely managed by the ReCycle Configurator and in case of cyclic dependencies based on the attached metadata the Configurator can apply appropriate resolution or rollback from invalid reconfiguration without the developer assistance.

ByeCycle [15] is a tool that is very similar to JooJ in that it checks for cycles among java packages. As a result only packages on which classes depend on are analysed to detect cyclic dependencies, such that internal invocations occurring within classes are not detected. AOR [1] tackles cyclic referential dependencies by reverting the dependency between modules such that the references points in one direction only. However, this can potentially lead to semantic interactions concerns, whereby one aspect could be in mutually exclusive of another. ReDac [10] uses a configuration framework to detect cyclic dependencies while composing components. The configuration framework works for multi-threaded component. However, the approach does not detect cyclic dependencies in the connector component.

With respect to AOM platforms: CAM/DAOP [3], FAC [9], FuseJ [13], DyMAC [5], and DyReS [14] none of the existing platforms provide mechanisms to detect the occurrence of cyclic dependency while composing and reconfiguring the platforms.

6. CONCLUSION AND FUTURE WORK

In this paper we have demonstrated the need to consider the occurrence of cyclic dependencies in aspect chains to better support and ensure consistent reconfiguration in dynamic AO middleware. We have illustrated the ReCycle model, a general approach for validating distributed dynamic reconfiguration, catering for potential cyclic dependencies following a dynamic distributed reconfiguration. Moreover, our solution does not change the implementation of the aspect-component, which would result in breaking the encapsulation of its functionality; this allows aspects dependencies to be dynamically evolving without changing the source-code of running aspects. The essence of our approach is that ReCycle can be encapsulated as an aspect to resolve any occurrence of cyclic dependency at configuration and reconfiguration. This means that ReCycle model can be independently woven and unwoven as required. We believe this gives the approach strong flexibility and generality that will allow it to be deployed in a number of AO-Middleware platforms not just AO-OpenCom.

Turning to future work, we first plan to investigate extending our approach to cover cyclic dependency in multi-threaded aspects environments. Then, we also plan to integrate our semantic resolution model [12] and the ReCycle model to ensure consistent aspect reconfiguration when building large-scale distributed middleware applications.

7. REFERENCES

[1] Apel, S., Kastner, C., Batory, D. 2008. Program refactoring using functional aspects. In Proc. 7th Conference. on Generative programming and component engineering. ACM Press, New York, 161-170.

[2] Coulson, G., Blair, G., Grace, P., et al. 2008. A Generic Component Model for Building Systems Software. In ACM Transactions on Computer Systems, Volume 26, Issue 1, February 2008. ACM Press, New York, Article 1.

[3] Fuentes, L., Pinto, M., Troya. J.M.. 2007. Supporting the Development of CAM/DAOP Applications. In Journal Software Practice & Experience John Wiley, Vol. 37. 21-64.

[4] Grace, P., Lagaisse, B., et al. "A Reflective Framework for Fine-Grained Adaptation of Aspect-Oriented Compositions". In Proceedings of 7th Software Composition, 2008.

[5] Lagaisse, B. and Joosen, W. 2006. True and Transparent Distributed Composition of Aspect-Component. In Proceeding Middleware Conference. ACM Press, New York, NY, 42-61.

[6] Lingas, A., Lundel, E., Efficient approximation algorithms for shortest cycles in undirected graphs. In Elsevier Publications Volume 109, Issue 10, 30 April 2009, 493-498.

[7] Melton, H., Tempero, E., 2007. JooJ: real-time support for avoiding cyclic dependencies. In Proc. conference on Computer science. Vol. 62. ACM Press, New York, 87-95.

[8] Pawlak, R., Duchien, G., et al. 2004. JAC: an aspect-based distributed dynamic framework. In Journal Software Practice, Volume 34, Issue 12, 1119 - 1148.

[9] Pessemier, N., et al., "Component-based and Aspect-oriented Systems". In Proc. Software Composition, 2006.

[10] Rasche, A., Polze, A. ReDAC Dynamic Reconfiguration of Distributed Component-Based Applications with Cyclic Dependencies. In Proc. IEEE on OO Real-Time Distributed Computing, 2008.

[11] Surajbali, B., Coulson, G., Greenwood, P., and Grace, P. 2007. Augmenting reflective middleware with an aspect orientation support layer. In Proc. 6th Int. workshop Adaptive and Reflective Middleware, ACM Press, Article 1.

[12] Surajbali, B., Grace, P and Coulson. G. 2009. Surajbali, B., Grace, P and Coulson. G. 2009. A Semantic Composition Model to Preserve (Re) configuration Consistency in Aspect Oriented Middleware. In Proc.8th International workshop Adaptive and Reflective Middleware, ACM Press, Article 6.

[13] Suvee, D., et al., "A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development". In Proc. 9th International SIGSOFT Symposium on CBSE, 2006.

[14] Truyen, E., Janssens, N, Sanen, F., et al., 2008. Support for distributed adaptations in AOM. In Proc. of the 7th Conference AOSD. ACM Press, New York, 120-131.

[15] Wuestefeld K., Rodrigo Oliveira, B., Beck, K., "ByeCycle", http://byecycle.sourceforge.net/.

Supporting Variability with Late Semantic Adaptations of Domain-Specific Modeling Languages

Tom Dinkelaker Martin Monperrus Mira Mezini

Technische Universität Darmstadt, Germany
{dinkelaker,monperrus,mezini}@cs.tu-darmstadt.de

ABSTRACT

Meta-object protocols are used to open up the implementations of object-oriented general-purpose languages to support semantic variability. They enable performing application-level semantic adaptations to the language even at runtime. However, such meta-object protocols are not available for domain specific-modeling languages. Also, existing approaches to implementing domain-specific modeling languages do not support semantic adaptations, where the application basically redefines specific parts of the language semantics. We propose a new approach for the implementation of domain-specific modeling languages that uses meta-objects and meta-object protocols to open up the implementation of domain-specific abstractions. This approach enables runtime semantic variability of the form of application-specific late semantic adaptations of domain-specific modeling languages that depend on the runtime application context.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Language Constructs and Features—*Classes and Objects, Frameworks*; D.2.11 [**Software Architectures**]: Languages

General Terms

Design, Languages

Keywords

Domain-Specific Modeling Languages, Variability, Semantic Adaptation, Meta-Object Protocols

1. INTRODUCTION

Domain-specific modeling languages (DSMLs) facilitate the development of software in a certain application domain by providing direct means to express domain-specific abstractions and operations. DSMLs are supported by domain-specific interpreters or compilers, which implement DSML syntax and semantics [22].

Previous work showed that most of the current methods for implementing DSMLs are closed with respect to changes in their semantics [30, 22]. For instance, van Deursen pointed that extensible DSL compilers and interpreters [30] have been little explored and Mernik stated [22] that *"building DSLs [...] in an extensible way"* is an open problem. To support the need for extensible modeling languages, note that even UML2 [25] defines an extension mechanism of its semantics called *semantic variation point*. This is where this makes its contribution: we propose a new approach for implementing DSMLs which supports semantic variability.

This approach allows DSML users to define the DSML semantics that exactly fits their needs, in the spirit of semantic variation points of UML2 [25]. For illustration, consider a model of a travel package booking Web service defined in a DSML for composing Web services. Let us assume that the initial DSML semantics only supports synchronous events consumption, thus DSML programs can only handle synchronous Web service partners. What happens if the default Web service for booking flights fails and that the only other partner available works asynchronously? The DSML application has to be rewritten using another modeling language. If the user could change the DSML semantics in an application-specific manner, she could implement an adaptation of the DSML semantics in order to enable asynchronous event consumption to also support asynchronous partners, while still reusing the initial DSML application and most of the default DSML semantics. If the DSML application has to support at runtime both synchronous and asynchronous partners, i.e. be self-adaptive to recover dynamically if a partner fails [2], the semantic adaptation has to depend on the execution context.

In [31], van Gurp coined the term *late variability* in the context of product lines, where it means changing a product after its delivery. In this paper, we explore the use of *late variability* in the context of DSML, which means being able to change the DSML semantics after the default interpreter or compiler has been delivered. To do so, we define the concept of *late semantic adaptation* as a replacement of one or more parts of the default semantics of the DSML within a DSML program; *late* meaning that the adaptation occurs after the delivery of the DSML and even as late as during the execution of a DSML program.

Let us now list and define what could be adapted in a DSML: A *domain type* is a type of the metamodel of the DSML, i.e. a type representing a domain abstraction. Adapt-

First International Workshop on Composition and Variability colocated with AOSD'2010 Rennes, St. Malo, France

ing a domain type means that every instance created after the adaptation will have the new semantics. Domain types contain *domain operations* that may change the state of domain objects. A *domain object* is an instance of a domain type. Adapting a domain object means that the semantics (the implementation) of its domain operations (and only the operations of this particular object) are changed.

Existing approaches to implementing DSMLs (e.g. DSML compilation [1], domain virtual machine [23], and polymorphic embedding [14]) do not support such late semantic adaptations, where the application basically redefines specific parts of the language semantics. In existing approaches, changing the semantics at runtime is only possible if the semantic adaptations had already been anticipated at design time. However, it is not possible to envision every possible semantic adaptation a priori at design-time. Even if it would be possible to embed into the DSML variation points for the known adaptations, the resulting implementation of the DSML semantics would be bloated with additional attributes and conditional logic. Such a *one-size-fit-alls* solution hampers the design of the default semantics which is used by most of the DSML programs. Last but not least, the DSML semantics could not be causally connected to the application state (i.e., dependent on the application state).

Our contribution is a method to implement DSMLs which are able to support runtime semantic adaptations. Meta-object protocols (MOPs) are interfaces to change the semantics of object-oriented programming languages [17]. MOPs define meta-objects that for instance handle the dispatch of method calls. Our key insights are that: 1) domain objects can be linked to a meta-object and 2) by implementing a DSML in a specific manner, an existing general-purpose MOP enables to change the semantics of the DSML itself. The method supports unanticipated semantic adaptations after the default DSML implementation has been delivered to a particular domain, as late as during the execution of a DSML program.

To evaluate our approach, we instantiate the method by building a DSML for state machines. This DSML supports semantic adaptations discussed previously in literature [25, 3].

The remainder of this paper is structured as follows. Section 2 presents different dimensions along which semantic adaptations may be defined. The proposed DSML method is presented in Section 3. Section 4 evaluates the support for semantic adaptations of a DSML implemented following our method. Related work is discussed in Section 5. Section 6 concludes the paper and discusses future research directions.

2. DIMENSIONS OF LATE SEMANTIC ADAPTATIONS

It's not straightforward to adapt the DSML semantics at the application level. DSML programmers require analysis means to design their adaptations. Hence, we have identified the following dimensions of semantic adaptations.

2.1 Scope of Variability

The first dimension along which we classify DSML semantic adaptations is the scope of variability. This dimension is discrete and has two values: (a) *domain type semantic adaptation* and (b) *domain object semantic adaptation.* A domain type semantic adaptation affects the semantics of all domain objects of a given domain type. On the contrary, a domain object semantic adaptation affects only one particular domain object.

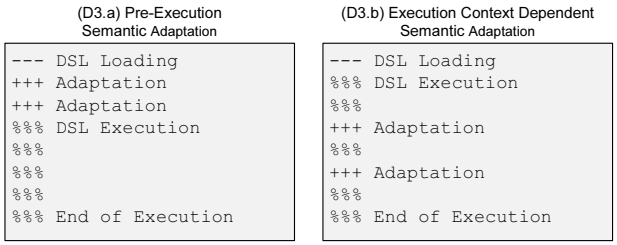

Figure 1: Semantic Adaptations and DSML Program Execution

2.2 Granularity of Changes

The second dimension of semantic adaptations is the granularity of adaptations that are made. The size of these adaptations ranges from one single domain operation to multiple parts of the DSML semantics. Indeed changing one part of the semantics often requires also changing another part of the semantics, and multiple "elementary" semantic adaptations have to be packed into a unit of change.

2.3 Relation to DSML Execution

The third dimension characterizes the relation between the point in time in which the semantic adaptations happen and the point in time when DSML programs are executed. In most cases, the right semantics for a program execution can be determined beforehand and stays fixed for a complete program run. Sometimes, the selection of the right semantics depends on the execution state of a DSML program, i.e., a change in the DSML program context triggers a semantic adaptation.

Let us consider the following two abstract examples of execution traces that illustrate the two possible points in time when adaptations may take place. Figure 1 shows the difference in the execution traces of a pre-execution adaptation and a context-dependent adaptation. In both traces, the first step is to load the DSML program of which the corresponding trace is prefixed by "$---$". Then, two kinds of execution steps can occur: semantic adaptation ("$+++$") and normal DSML execution ("$\%\%\%$").

The left-hand side of figure 1 schematically depicts a pre-execution semantic adaptation: the semantics of the DSML changes before any domain object is created, or any call to a domain operation has occurred. Note that multiple adaptations can be applied as indicated. Then, the DSML program is evaluated until completion. In this case, the adaptation is independent of the DSML execution. The right-hand side of figure 1 depicts an execution context-dependent semantic adaptation, as used in the running example. Unlike the previous trace, the adaptation happens during DSML execution, depending on the concrete values of domain objects. This is symbolized by the interlacing of several regular domain operation execution and semantic adaptation steps. Such context-dependent adaptations enable semantic self-adaptation of DSML programs.

3. A NEW METHOD FOR IMPLEMENTING DSMLS

This section presents a method for implementing DSMLs. DSMLs interpreters implemented with this method have the particularity to allow late semantic adaptations (as described in 2), i.e. semantic adaptations of the DSML inside DSML programs. We use the Groovy programming language to demonstrate the feasibility of the approach, as well as to fully instantiate the approach later in section 4.

3.1 Using Groovy to Implement DSMLs

Groovy [5, 18] is an object-oriented scripting language that nicely integrates with Java [12]. We have selected Groovy as the implementation language of our method for the following reasons:

1. Groovy provides a runtime MOPs in which meta-objects are first-class entities that can be directly accessed and modified by users[1].

2. Groovy has a *flexible syntax* that enables the definition of embedded DSMLs with a small syntax overhead. While for other host languages, such as Haskell, a large syntax overhead has been measured [19], Groovy supports *named parameters* and *command expressions* that allow the DSML implementer to design the syntax of the embedded DSML more openly.

3. Groovy is accessible to a broad community of developers since it has a syntax that is close to the Java syntax. Groovy is seamlessly integrated into Java and Groovy code can be called from Java code and vice versa. Hence, DSML programs can be called from Java and DSML programs can call existing Java libraries. All these argument allow an easy dissemination of our method.

While our method for implementing DSMLs could be implemented using other programming languages that come with a meta-object protocol (Smalltalk [10], CLOS [17], Ruby [27]), none of these languages satisfy all the aforementioned requirements.

Let us now give a quick overview of the features of Groovy that our method uses for implementing DSMLs[2]. Every Groovy object is bound to a meta-object [17]. This meta-object has several responsibilities: 1) it contains the logic related to introspection (e.g. the method `getMethods`) and 2) it handles every method call to this object. It is possible to change or replace this meta-object at runtime.

Also, there is a registry that links a class name to its default meta-object. Every new instance of a class, say `x`, is bound to the meta-object for its class in the registry. Hence, when the registry is updated, already existing objects keep the old meta-object and the new generation of objects is bound to the updated meta-object.

Groovy supports first-class closures. A closure can be created dynamically, passed as parameters to methods and functions, and executed. Listing 1 illustrates these points

[1]Note that users do not have to understand and use the MOP as long as they use the default semantics of a DSML and do not need to adapt it.

[2]Note that our approach is not bound to Groovy specifically, but to dynamic languages with a MOP. For instance, our approach is completely applicable in the context of Ruby.

```
1  // creating a closure
2  aClosure = {x->
3  print "hello "+x }
4
5  def m(Closure c) {
6    c("world") // executing the closure
7  }
8
9  // passing the closure as parameter
10 m(aClosure)
```

Listing 1: Closures in Groovy

```
1  // creating a closure
2  aClosure = {-> bar() }
3
4  // two different contexts
5  class Context1 {
6    def bar() { println "bar" } }
7  class Context2 {
8    def bar() { println "bar2" } }
9
10 // executing the closure with Context1
11 aClosure.delegate=new Context1()
12 aClosure() // output "bar"
13 // executing the closure with Context2
14 aClosure.delegate=new Context2()
15 aClosure() // output "bar2"
```

Listing 2: Delegates in Groovy

Also, an important feature of Groovy closures is that their execution can be parameterized by a delegate context. By default, a closure has access to the *lexical context* in which it has been created. The lexical context contains all local variables of the closure. If it has been created within a method body, all variables available in the method are also available for the closure. In particular, all instance attributes and methods of the object that has created the closure are available when the closure is executed. This creating object is called the *owner* of the closure. In addition to the lexical context, the available context can be extended. When using a *delegate* for the closure, by changing its `delegate` attribute to refer to the *delegate*, the lexical context of the delegate becomes accessible in the closure. This way, the instance attributes and methods of another object than its owner can be used. If a function is not found in the closure's lexical context, a method with same signature is looked up in the delegate context, as shown in listing 2. Note that depending on the current binding of the delegate, the execution of the same closure can produce different results.

Technically, extending the available context for a closure is possible because Groovy uses a special meta-object for every closure. This meta-object first tries to lookup attribute accesses and method calls in the lexical context of a closure, i.e., in the local variables and in the owner. If no attribute or method with a corresponding name or signature is available in the lexical context and if the closure's delegate attribute is set, then the meta-object tries to lookup the attribute or method in the class of the delegate object. Only if the attribute or method can be found neither in the lexical context nor in the delegate, Groovy throws a runtime error.

3.2 The Embedding of DSMLs

Our method is based on Hudak's method to implement DSLs [15], i.e., no parser and compiler has to be written.

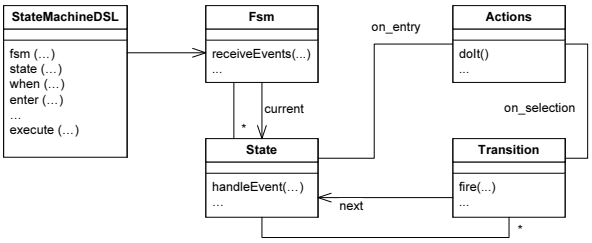

Figure 2: Architecture of the Default Interpreter

Implementing a DSML relies on two main steps. First, a metamodel specifies domain types, domain operations, and associated semantics in terms of a set of interrelated Groovy classes. Second, a syntactic language interface – a Groovy class – maps DSML syntax to DSML semantics by mapping DSML keywords to domain objects. There is a method in the syntactic language interface for each keyword in the DSML. DSML programs are enclosed in Groovy closures and the latter are assigned an instance of the language interface class as their delegate. The delegation mechanism of Groovy closures then maps DSML keywords to the corresponding method calls to a closure's delegate.

For instance, let us consider the implementation of the default semantics for a DSML for state machines. Figure 2 depicts the design of this DSML. The metamodel consists of classes `Fsm`, `State`, `Transition`, `Actions`. The `Fsm` class maintains a set of states, refers to a current state and implements some default semantics of state machines, e.g., the method `receiveEvents(...)` defines the dispatch mechanism of events received by a state machine. `State` instances maintain a set of outgoing transitions and may have an `on_entry` action and implement the state semantics. For instance, the method `handleEvent(...)` defines the state event handling mechanism. A `Transition` points to the `next` state. The `fire(...)` method is called whenever a transition is selected. The class `Action` encapsulates a set of domain-specific actions; the semantics of an action execution is encoded in the `doIt` method. Finally, the syntactic language interface is implemented in class `StateMachineDSML`. There is a method in `StateMachineDSML` for each keyword in the DSML. When called, these methods instantiate domain objects.

Listing 3 shows an excerpt of an embedded DSML program for state machines. The DSML program is contained in the closure `dslPackage` (cf. line 3) which is configured in line 22 to have an instance of `StateMachineDSML` as its delegate and whose evaluation starts at line 24. The evaluation is performed in two steps.

The first step transforms the textual DSML program (from line 5 to 9), embedded into the host language syntax, to a representation as a network of interrelated domain objects (instances of the domain classes from the metamodel, e.g., the instance `MyFsm` of class `Fsm`). During the execution of the closure, keywords, e.g. `fsm`, `state`, and `when`, are encountered in the DSML program. These keywords are turned into method calls due to the flexible syntax of Groovy. When using curly brackets at the end of a keyword method call, Groovy creates a closure and passes the closure to the method call as the last parameter. For instance, the program segment `fsm 'MyFsm', {...}` is turned

```
1  // this closure contains
2  // the DSML program + adaptations
3  def dslPackage = {
4    // the DSML program
5    fsm 'MyFsm', {
6      state 'S1', { ... }
7      ...
8      state 'SX', { ... }
9    }
10
11   // will execute the DSML program
12   // when the closure dslPackage is called
13   // with the event list passed as a parameter
14   MyFsm.execute({'ok','error',...})
15 }
16
17 // in Groovy a delegate is
18 // the interpretation context for closures
19 // we set the interpretation context for dslPackage
20 //  to be the default interpreter for state machines
21 dslPackage.delegate =
22   new de.tud.statemachine.StateMachineDSML()
23
24 dslPackage(); // evaluates the closure
```

Listing 3: State Machine Embedded in Groovy

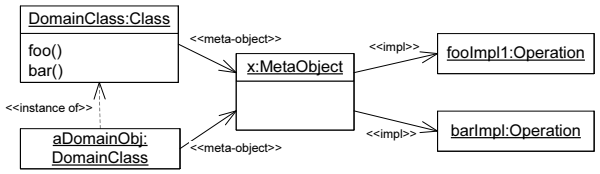

Figure 3: A Meta-Level for Domain Objects

into a method call of the form `fsm('MyFsm', closure-in-brackets)` with the `dslPackage` closure as the receiver. These calls are dispatched to closure's delegate, in this case a `StateMachineDSML`, of which the method with the corresponding keyword name and signature is called. These methods serve mostly as factories of domain objects. The second step, in line 14, is the execution of the DSML program as a method call to a domain object (resp. `MyFsm` and `execute`), given a specific execution context (`{'ok','error',...}`). This triggers a cascade of method calls on domain objects created during the first step.

3.3 How to Support Late Semantic Adaptations in DSMLs

In the following, we explain how to use meta-objects to enable late semantic adaptations. The meta-level introduced by our method is schematically depicted in Figure 3. Every domain type is mapped to a domain class A domain class, e.g., `DomainClass` in Figure 3, defines the domain operations of its instances – the domain objects, e.g., `aDomainObj`. The semantics of domain objects is reified in *meta-objects*, which are responsible for handling the execution of domain operations. Every domain class is associated with a meta-object, which is the default meta-object of any new instance of the domain class. Meta-objects, e.g., `x` in Figure 3, dispatch method calls received by domain objects to concrete implementations of domain operations. The links `meta-object` and/or `impl` can be changed at runtime, which is the key to allow dynamic semantic adaptations.

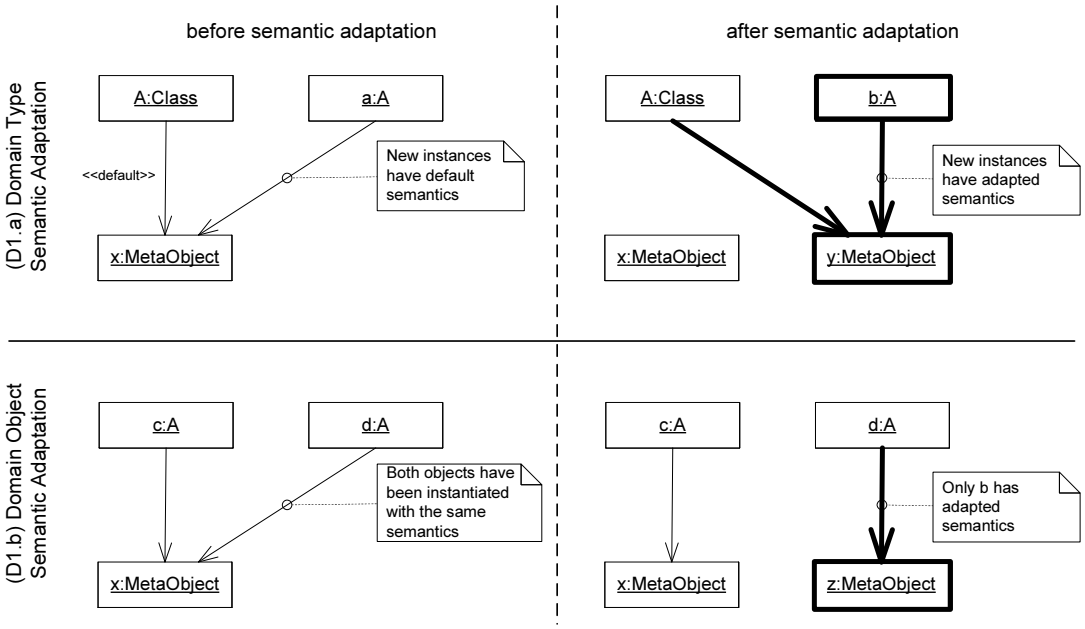

Figure 4: Dimension 1 – Scope of Variability

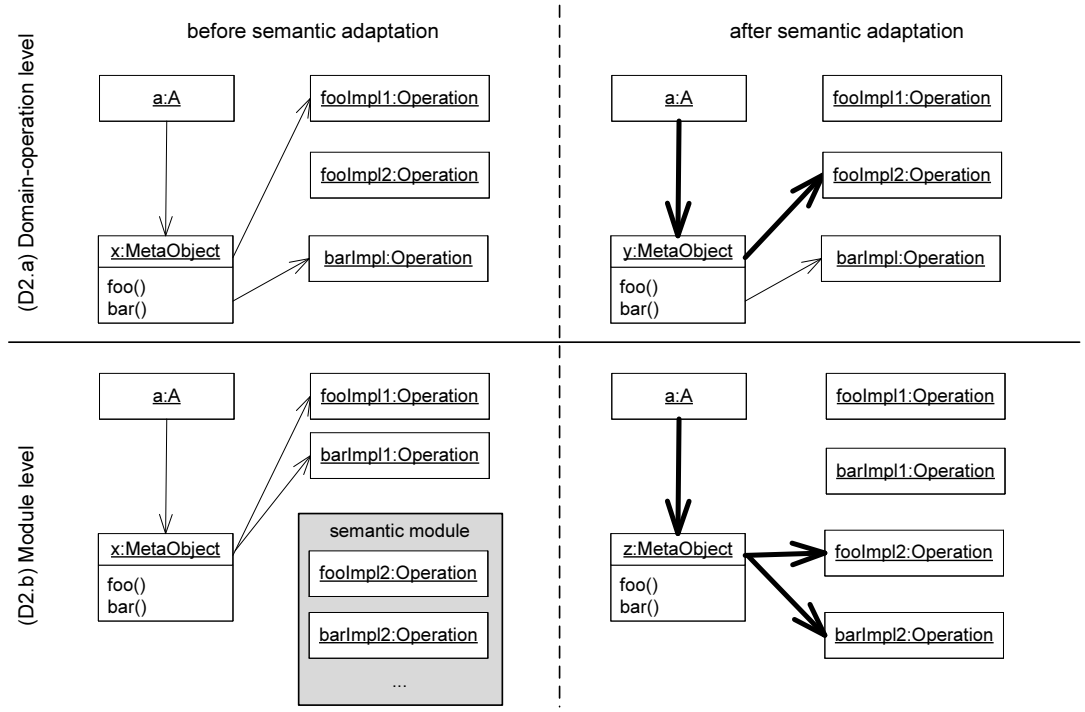

Figure 5: Dimension 2 – Granularity of Changes

Our base embedding method in Groovy presented above supports this meta-level: 1) all domain classes are Groovy classes whose semantics can be modified at runtime; 2) all domain objects are Groovy objects, and the corresponding meta-object can be changed for a single instance only.

3.3.1 Scope of Variability

Figure 4 shows how the two kinds of variability with regard to the scope dimension – domain type versus domain object – are supported in the proposed meta-level.

The upper part shows the effects of semantic adaptations whose scope is an entire domain type; the lower part corresponds to an adaptation that is specifically scoped for a particular domain object, thus, only domain objects are shown there. Both parts show the relation between domain types and domain objects to their corresponding semantics (encapsulated in a meta-object) before and after the adaptation.

In the upper left quadrant, the domain type `A` is bound to the default semantics represented by the meta-object `x`. Every new domain object that is created, e.g., `a`, runs under the default semantics. The semantic adaptation defines new semantics for domain type `A`. In the upper right quadrant, a new meta-object `y` is defined to represent the new semantics and `A` is associated with it. Any domain object created subsequently runs under the new semantics: the domain object `b` is created after the adaptation, hence, it is linked to the new meta-object `y`. Objects that were created before the semantic adaptation continue to run under the previous semantics, e.g., `a` is still linked to the meta-object `x`.

In the lower left quadrant, the domain objects `c` and `d` are created with the same semantics. The object-level semantic adaptation depicted here modifies the semantics of `d` only. The lower right quadrant shows the domain objects, meta-objects, and their relations after the semantic adaptation has taken place. While `c` keeps the former semantics, `d` uses the new semantics represented by meta-object `z`.

3.3.2 Granularity of Changes

Figure 5 depicts how adaptations at different levels of granularity are also supported by the proposed meta-level. The upper part shows the most fine-grained semantic adaptation at the level of an atomic domain operation. Unlike figure 4, the meta-objects are represented along with the domain operations. Doing so, we can highlight that the adaptation can separately impact a particular operation. In the upper left quadrant, the object `a` is attached to meta-object `x`; in the upper right quadrant, the same object is attached to a new meta-object `y`, which is the result of cloning `x` and binding `foo` to a new implementation, called `fooImpl2`. This way, the whole default semantics gets reused except the re-bound domain operation(s).

In general, it is likely that a semantic adaptation affects several places in the default implementation of the semantics. Obviously, it is preferable to apply the changes together as a unit of semantic adaptation. In contrast to the atomic adaptation at the level of a single domain operation, in this case the changes have to be packed into a bigger variability unit. This abstract unit is depicted as the gray rectangle in the lower left part of figure 5. When an adaptation happens, all changes of this adaptation unit are performed in concert. The impacted domain objects are then bound to a new meta-object, which is the result of mixing the previous meta-object and the semantic adaptation unit. In figure 5, after the change, the domain object `a` is attached to the meta-object `z`, which binds both `foo` and `bar` to new implementations.

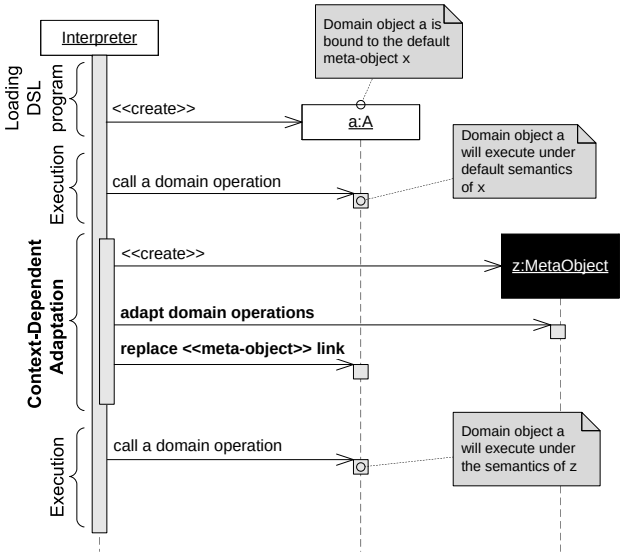

Figure 6: Context-Dependent Semantic Adaptation

3.3.3 Relation to DSML Execution

Semantic adaptations may occur at loading time or at runtime of DSML programs. Figure 6 depicts the sequence diagram of context-dependent semantic adaptation. Normal execution and semantic adaptation are interlaced. After *loading DSML program* the first *execution* phase starts. In this phase, while executing the DSML program and when entering a context that requires a semantic adaptation, a special phase is started that consists of applying semantic adaptations onto domain objects (respectively domain types). At the end of the adaptation phase, control is passed back to DSML execution. The subsequent *execution* phase will run under the new tailored semantics. It is worth mentioning that several adaptation phases can be executed, e.g., the adaptation can be reverted or other semantics can be installed.

4. APPLYING THE METHOD

This section discusses an implementation of a DSML for finite state machines (FSM DSML) using the method described in section 3. We show why the DSML interpreter supports semantic adaptations and how to implement them in at the level of DSML programs.

4.1 Possible Semantic Adaptations for the FSM DSML

The UML specification [25] discusses several semantic adaptations for state machines. We consider here two of them.

4.1.1 Synchronous vs. asynchronous event handling.

Listing 4 shows two possible implementations of `State`'s domain operation `handleEvent`. The first implementation is the default one and encodes synchronous event handling; the second implementation supports asynchronous event handling.

```
1  // default: synchronous event handling
2  def handleEvent(Event e) {
3    this.fsm.currentState =
4      this.transitionSelection(e).fire()
5  }
6
7  // alternative: asynchronous event handling
8  def handleEvent(Event e) {
9    if (this.queue.isEmpty) {
10     this.fsm.currentState =
11       this.transitionSelection(e).fire()
12   } else {
13     this.fsm.queue.add(e)
14   }
15 }
```

Listing 4: Two Implementations of handleEvent

```
1  // default semantics: deterministic transition selection
2  def transitionSelection(Event e) {
3    return this.transitions.findAll(event).first
4  }
5
6  // alternative semantics: random transition selection
7  def transitionSelection(Event e) {
8    return this.transitions.findAll(event).getRandom()
9  }
```

Listing 5: Two Implementations of transitionSelection

4.1.2 Deterministic vs. random transition selections.

A given state of a state machine can have several transitions matching a given event. In this case, a state machine implementation has to provide a transition selection policy. Following our method, the semantics of transition selection is encoded in the `transitionSelection` method of the domain class `State`. Listing 5 shows two possible implementations of `transitionSelection`. The default implementation of the transition selection policy is a deterministic one. It returns the first element of the collection of matched transitions. The alternative implementation selects a random item in the collection of matched transitions for a fairer load-balancing.

The following sub-sections will discuss scenarios of using our method to apply alternative semantics for event handling and transition selection thereby varying the kind of adaptation along the three dimensions discussed previously. Section 4.3, specifically, will discuss how to combine several variation points in one semantic module; for instance, how to use the tailored version of both `handleEvent` and `transitionSelection` in a concise and elegant manner.

4.2 Scope of Adaptations

Listing 6 shows the implementation of the first dimension presented in section 3.3.1.

Listing 6 illustrates domain type semantic adaptation. The module `dslPackage` (lines 5-9) is a Groovy closure, which consists of three parts: (a) the declaration of a DSML program (lines 5 to 9), (b) a piece of meta-program that tailors the semantics of the DSML (lines 17 to 19), and (c) a piece of code that starts the execution of the DSML program (line 26). Parts (a) and (c) have been explained in 3.2. The adaptation consists in changing the `transitionSelection` method of the default state meta-object in line 17. As explained earlier, all instances of class `State` are affected by

```
1  // this closure contains the DSML package
2  // (DSML program + adaptations)
3  def dslPackage = {
4    // the DSML program
5    fsm 'MyFsm', {
6      state 'S1', { ... }
7      ...
8      state 'SX', { ... }
9    }
10
11   // adaptation of the default semantics
12   // of the domain class State
13   // by replacing the implementation of
14   // the transitionSelection method
15   // of the default meta-object associated
16   // with the State class
17   State.metaClass.transitionSelection = { event ->
18     return this.transitions.findAll(event).last
19   }
20
21   // executing the DSML program
22   MyFsm.execute({'ok','error',...})
23 }
24 dslPackage.delegate =
25   new de.tud.statemachine.StateMachineDSML()
26 dslPackage();
```

Listing 6: Domain Type Semantic Adaptation

this kind of adaptation and will execute with the tailored transition selection semantics.

For sake of space, we cannot elaborate on a complete DSML program that performs a semantic adaptation at the level of a single domain object. The code is similar to that in listing 6, except that it is not the `metaClass` of class `State` (line 17) that is changed but the `metaClass` of a domain object. For example, we can change the meta-object of `S1` using `MyFsm.S1.metaClass.transitionSelection = {...}` .

4.3 Granularity of Adaptations

This section illustrates the second semantic dimension, presented in section 3.3.2. On the one extreme in this dimension, a semantic adaptation affects a single domain method; on the other extreme, a semantic adaptation may imply the construction of a completely new meta-object.

Listing 7 shows DSML code that is embedded similarly to listing 6. It focuses on the adaptation in lines 7 to 10. The only element that is changed is a domain method of a domain class.

On the contrary, listing 8 shows the creation of a semantic module and its use for tailoring the semantics of a domain class. Similarly to using classes to represent domain types, we use a new subclass for modularizing alternative semantics for a domain type. In the example, lines 2–9 define such a subclass called `TailoredState`. Subclassing a domain type to create a new meta-object allows leveraging two key Groovy features used in listing 8:

1. The possibility to attach new semantics to an existing domain class, using a registry mechanism (line 13).

2. The automatic creation of a meta-object for each new class (line 14).

A meta-object for the new subclass is automatically created and stored in the class variable `TailoredState.metaClass`. In listing 8 lines 13–14, we register this meta-object

```
fsm 'MyFsm', {
  ...
}

// we tailor only transition selection
// part of the semantics of States
State.metaClass.transitionSelection = {
  event ->
  return this.transitions.findAll(event).last
}
// executing the DSML program
MyFsm.execute({'ok','error',...})
```

Listing 7: Method-Level Adaptation

```
// we use classes for modularizing the semantics
class TailoredState extends State {
  def transitionSelection(event) {
    /* cf. variation point transitionSelection */
  }
  def handleEvent(event) {
    /* cf. variation point handleEvent */
  }
}

// we tailor the semantics of State in one unit
// for both event handling and transition selection
InvokerHelper.metaRegistry.setMetaClass(State,
  TailoredState.metaClass)
// executing the DSML program
MyFsm.execute({'ok','error',...})
```

Listing 8: Semantic Module Adaptation

also for the `State` class. As a consequence, the domain operation implementations of `TailoredState` will be used for `State` objects.

4.4 Moment of Adaptations

As shown figure 1, a pre-execution adaptation is simply a piece of code preceding the DSML program that changes the semantics of the language itself. While our method enables such adaptations, for sake of space, we cannot elaborate on them.

We now present a complete example of a semantic adaptation that occurs during the execution of a DSML program, i.e. a context-dependent adaptation. Consider now the self-adaptive DSML program in listing 9. This DSML program is a state machine representing a Web service composition for a travel booking process. The machine consists of two states, first booking a flight and second booking a hotel. In both states, a call to a Web service is made in the `on_entry` blocks. States transitions are triggered by the reception of Web service responses.

This program is self-adaptive since it is able to recover from failing synchronous partners, thanks to our FSM DSML which supports semantic adaptations: the `TravelPackage` program is able:

1. to handle the error event in the `BookingFlight` state (line 12);
2. to replace the failing partner whith an asynchronous one (line 14)
3. to adapt itself to the new webservice by changing the event reception semantics of the DSML only for state `BookingFlight` in order to listen to asynchronous events (lines 12–21).

```
def dslPackage = {
  // declaration of the DSML program
  fsm 'TravelPackage', {
    state 'BookingFlight', {
      on_entry {
        /* synchronous call to flight_webservice */ }

      // nominal mode
      when 'done', { enter 'BookingHotel'}

      // error recovery mode
      when 'error', {
        // change to an asynchronous webservice
        targetService = "another_flight_webservice"
        // and we have to change the semantics too
        this.metaClass.handleEvent = { event ->
          /* new asynchronous implementation */
        }
        // re-enter current state
        enter 'BookingFlight'
      } // end when
    } // end state

    state 'BookingHotel', {
      on_entry {
        /* synchronous call to hotel_webservice */ }
      }
    }
  }

  // executing the DSML program
  TravelPackage.execute({'ok','error','done',...})
}
dslPackage.delegate =
  new de.tud.statemachine.StateMachineDSML()
dslPackage();
```

Listing 9: Context-Dependent Adaptation

In such cases, the semantic adaptation code becomes part of the code of the DSML program and the adaptation logic is executed only when DSML execution reaches the lines 12–21.

In this section, we have presented an instantiation of our method as a proof of concept that has shown its real implementability. From the viewpoint of end-users of our method, i.e. DSML designers, this section is fully complementary to the conceptual presentation of our method in sections 2 and 3 and enables them to implement on their own a DSML that supports late semantic adaptations.

5. RELATED WORK

Domain-Specific Languages.

The implementation of DSMLs using "traditional, closed" compilers (e.g. [1]) does not allow semantic adaptations. In contrast, extensible compilers, such as Polyglot [24] or JastAdd [13, 8], target semantic adaptations of the form of extensions to Java language. Akesson et al. [33] address the implementation of extensible DSMLs. However, extensible compilers do not support all the semantic adaptation dimensions discussed in this paper. Only class-level adaptations are supported, in the sense that the adaptation granularity is the class as well as in the sense that all domain objects are executed under the same semantics. Furthermore, dynamic semantic adaptation that depends on application execution state is not supported. Last but not least, the application adaptation are very different as opposed to our approach, where – due to using the language embedding technology –

the same language is used for implementing an application and the semantics of the DSML.

Our approach follows the *domain virtual machine* pattern [9], i.e., it is a DSML interpreter realized by a set of domain classes implementing the domain semantics in their methods. Kermeta [23] is a language to implement DSML interpreters following the domain virtual machine pattern. However, our domain classes are embedded into a host language which allows to seamlessly integrate DSML programs and programmatic semantic adaptations. More importantly, our approach supports non-invasive, application-specific, and even execution context specific semantic adaptations.

Steele [28] proposes to build interpreters out of a set of building blocks called *pseudomonads*, in reference to Haskell monads [32]. Achieving a semantic adaptation can be done by composing interpreters. Comparing to our method, the DSML programmer has to understand not only the interpreter of the DSML but also the composition operator of pseudomonads.

Ramsey [26] described the implementation of the Lua scripting language as an embedded interpreter in Objective Caml. While Ramsey implements a general-purpose language (Lua) interpreter, our approach targets domain-specific interpreters in order to design them as extensible.

The initial method of embedding DSMLs by Hudak [15] does not consider the issue of semantic adaptations. Similar to our work, polymorphic embedding [14] enables several interpretations of a DSML program by employing a similar architecture for the DSML implementation that separates the language interface and the domain metamodel. However, polymorphic embedding does not support a meta-level architecture allowing DSML programs to change their semantics in a fine-grained and application context-specific manner during their execution.

Reflection and Meta-Object Protocols.

Meta-interfaces have been implemented for various languages, e.g., 3-KRS [21], CLOS [17], Smalltalk [10]. Meta-object protocols (MOPs) provide "interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation" [17]. MOPs are *open implementations* [16] of (object-oriented) general-purpose languages. Compile-time MOPs have been provided for popular compiled languages OpenC++ [4] and OpenJava [29]. MOPs have been adopted in dynamic scripting languages, such as Ruby [27] and Groovy [5]. Using the above MOPs for extending DSML semantics have not been addressed.

There are no methods for DSML implementation available, that derive a MOP for the implemented DSML. The approach proposed in this paper is generic for class-based languages. Other dynamic languages that come with a MOP can be used to provide a flexible DSML semantics as presented in this paper.

xPico [11] allows to extend the syntax and semantics even at runtime by reflectively manipulating the AST at well-defined adaptation points. The idea to use reflection and the targeted flexibility is similar to our approach. Although xPico allows syntactic variability, the semantic adaptation of xPico is limited, as explicit adaptation points must be defined to allow extensibility. The problem with the xPico approach is that it does not provide an adequate meta-interface and provides only access to the AST but not to domain abstractions. However when implementing a DSML for multiple users, it is impossible to envision every adaptation point at design-time of the DSML semantics. On the contrary, our approach permits unanticipated adaptation points, i.e., every domain class method is a *latent* adaptation point.

A domain-specific meta-object protocol for distributed environment, called diMOP, has been presented in [20]. This design-time MOP is used to specify behavioral characteristics, such as non-functional concerns, at the design level in extended UML diagrams. However, the focus of this paper is the language level and executable meta-objects. The idea of having a domain-specific meta-object protocol is an interesting one. The diMOP is only a MOP for one domain, while every DSML implemented following our approach exposes a domain-specific MOP.

6. SUMMARY AND FUTURE WORK

In this paper, we have presented a method for implementing DSMLs that support semantic adaptations that may be application-specific and may occur as late as during the execution of DSML programs. The proposal leverages meta-objects [16] in the context of domain-specific modeling languages. Also, we have elaborated on an instantiation of the method in the Groovy programming language in the context of state machines. Although not shown in this paper, our solution is applicable for DSMLs with more complex sets of domain concepts (and language constructs), such as workflow languages, aspect languages, and others [6, 7].

As usual for dynamic approaches, there is a trade-off between adaptability and statically checked correctness. Our approach supports a maximal adaptability and may suffer from possible correctness issues. For instance, DSML programmers who override part of the default DSML semantics might violate contracts and responsibilities that are implicit in the DSML design. These limitations will be addressed in future work. For instance, semantic adaptations may require adaptations in several domain classes and operations performed in concert. Our future work will also explore explicit contracts that can be checked at runtime to ensure the semantic consistency of adaptations.

7. REFERENCES

[1] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *Conference on Software Reuse*, pages 143–53, 1998.

[2] A. Charfi, T. Dinkelaker, and M. Mezini. A Plug-in Architecture for Self-Adaptive Web Service Compositions. In *Proceedings of the 2009 IEEE International Conference on Web Services*, pages 35–42. IEEE Computer Society, 2009.

[3] F. Chauvel and J.-M. Jézéquel. Code Generation from UML Models with Semantic Variations Points. In L. Briand and C. Williams, editors, *UML MoDELs*, volume 3713 of *LNCS*, pages 54–68, Montego Bay, Jamaica, October 2005. Springer Verlag.

[4] S. Chiba. A metaobject protocol for C++. In *OOPSLA*, pages 285–299. ACM Press New York, NY, USA, 1995.

[5] Codehaus. The Groovy Home Page. `http://groovy.codehaus.org/`.

[6] T. Dinkelaker. Versatile language semantics with reflective embedding. In *Proceedings of the 2009 OOPSLA Doctoral Symposium*, 2009.

[7] T. Dinkelaker, M. Eichberg, and M. Mezini. An architecture for composing embedded domain-specific languages. In *Proceedings of AOSD*, 2010.

[8] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *Proceedings of OOPSLA'2007*, 2007.

[9] J. Estublier, G. Vega, and A. D. Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *Proceedings of MODELS/UML*, 2005.

[10] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, Boston, MA, USA, 1983.

[11] S. Gonzalez, W. De Meuter, and V. Brussel. Domain-Specific Language Definition Through Reflective Extensible Language Kernels. In *Workshop on Reflectively Extensible Programming Languages and Systems (at GPCE)*, 2003.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass, 2000.

[13] G. Hedin and E. Magnusson. JastAddŮan aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[14] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *Generative Programming and Component Engineering (GPCE'08)*, pages 137–148. ACM New York, NY, USA, 2008.

[15] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[16] G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, 1996.

[17] G. Kiczales, J. d. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[18] D. König and A. Glover. *Groovy in Action*. Manning, 2007.

[19] T. Kosar, P. Martínez López, P. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and software technology*, 50(5):390–405, 2008.

[20] J. Lee, S. Min, and D. Bae. Aspect-Oriented Design (AOD) Technique for Developing Distributed Object-Oriented Systems over the Internet. In *International Computer Science Conference*. Springer, 1999.

[21] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[22] M. Mernik and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[23] P. A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005*, 2005.

[24] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[25] OMG. UML 2.0 superstructure. Technical report, Object Management Group, 2004.

[26] N. Ramsey. Ml module mania: A type-safe, separately compiled, extensible interpreter. *Electronic Notes in Theoretical Computer Science*, 148(2):181–209, 2006.

[27] Ruby programming language. `http://www.ruby-lang.org/`.

[28] G. L. Steele. Building interpreters by composing monads. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 472–492. ACM New York, NY, USA, 1994.

[29] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. *Reflection and Software Engineering*, 1826, 2000.

[30] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[31] J. van Gurp. *Variability in software systems: the key to software reuse*. PhD thesis, Blekinge Institute of Technology, 2000.

[32] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming (LFP'90)*, pages 61–78, 1990.

[33] J. Åkesson, T. Ekman, and G. Hedin. Development of a modelica compiler using jastadd. *Electronic Notes in Theoretical Computer Science*, 203(2):117 – 131, 2008. Workshop on Language Descriptions, Tools, and Applications (LDTA 2007).

A Case for Custom, Composable Composition Operators

Wilke Havinga, Christoph Bockisch, Lodewijk Bergmans
Software Engineering group – University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{w.havinga,c.m.bockisch,l.m.j.bergmans}@ewi.utwente.nl

ABSTRACT

Programming languages typically support a fixed set of composition operators, with fixed semantics. This may impose limits on software designers, in case a desired operator or semantics are not supported by a language, resulting in suboptimal quality characteristics of the designed software system. We demonstrate this using the well-known State design pattern, and propose the use of a composition infrastructure that allows the designer to define custom, composable composition operators. We demonstrate how this approach improves several quality factors of the State design pattern, such as reusability and modularity, while taking a reasonable amount of effort to define the necessary pattern-related code.

1. INTRODUCTION

One of the most important quality characteristics of source code is its modularity. Good modularity is achieved, when each distinct piece of behavior (also called concern) in a program is encapsulated in one or only a few modules; and when it is possible to extend and refine this behavior in a way that requires no changes to existing code. A high degree of modularity in source code, thus, favors its re-usability and maintainability.

The degree of modularity that can be achieved, is significantly influenced by the composition power of operators offered by a language to compose modules. Therefore, research in the field of programming languages is intensively concerned with providing new composition operators. Examples are method or function calls, aggregation, inheritance, mixin-composition, or aspect-oriented composition.

However, we have observed that in each programming language only a few of the known composition operators are at the developer's disposal as language features. This hinders the modularity of source code. Thus, the ability to modularize source code is limited by the choice of composition operators made by the language designers. In our research, we want to enable the developer to freely use and mix all existing—and future—composition operators.

For most of the composition operators, different variations exist, e.g., when inheriting from a class, either the parent (e.g., in Beta) or the child implementation (e.g., in Java) may have precedence. While approaches exist to support different variants of single operators simultaneously (e.g., Beta-style and Java-style inheritance in [12]), the developer is typically provided with very limited choice. That is, different concerns may be well modularizable in different composition styles; but if no language exists that supports all necessary styles, not all concerns can be optimally modularized.

To avoid this limitation, often domain-specific languages (DSL) are developed that provide composition operators tailored toward a specific program domain. However, developing a DSL only pays off, when it is used sufficiently often. If this approach, thus, is not feasible and a general-purpose language is used, often the missing composition operators are emulated by a specific programming style, e.g., in terms of design patterns [10], which encode interactions between (and thus compositions of) objects.

To emulate composition operators, design patterns typically require some pieces of code which are application-independent (possibly tailored with element names from the application program) but cannot be localized in one module; we refer to these code pieces as *boilerplate code*. Boilerplate code entails several disadvantages.

- Firstly, it *obfuscates the design*; instead of specifying the relation of two or more modules explicitly, this code defines their composition imperatively. Because this code is scattered over multiple participating modules, the design intention becomes even more implicit.

- Secondly, boilerplate code is *difficult to write*. While it is not very sophisticated, its correctness is not easily enforced; for example, consider the Visitor pattern, where each `Element` class must implement the method `void accept(Visitor visitor){ visitor.accept(this); }`[1]. Each class contains the same line, but it is not possible to factor it out into the superclass.

[1]In languages like Java with an overloading semantics for methods, the static type of the argument distinguishes between the `accept` methods for different `Element` types in a `Visitor`.

- Finally, while it is sometimes necessary to combine multiple composition operators, *not all required operators can be emulated* by design patterns. As an example, consider the expression problem [8], where the building blocks of the application are data types and operations on them. With an object-oriented language, the data types are easily extensible, but not the operations. The Visitor pattern emulates a functional composition style, which makes it easy to extend the operations, but in turn the data types cannot be easily extended. Different language-level solutions to this problem have been proposed that are all founded on combining multiple composition operators [5, 8].

As also others have noticed [7, 26, 13], we claim that quality characteristics of design pattern implementations can often be improved if the implementation language supports particular composition operators. However, while this decreases the complexity of programs using a supported pattern, providing such support by extending the syntax of a language will increase the language's complexity [19].

In our research, we are concerned with developing a composition infrastructure, where the developer can choose from different composition operators and from different variations thereof. However, we do not simply aim at providing a fixed set of composition operators to choose from; but we aim to provide an infrastructure in which composition operators can be user-defined. In addition, our approach allows to re-use and combine implementations of composition operators—thereby developing new composition operators—because they are first-class. This also makes our approach open for future developments in the research of composition operators.

In this paper we present our approach through the example of the State pattern and our prototypical *Co-op* language. We have chosen this pattern because it is suitable to demonstrate the interplay between different composition operators, namely forwarding and delegation semantics as well as aspect-oriented composition. By this example we show that customizable composition operators can lead to a re-usable implementation of design patterns as well as to improved modularity of source code.

2. COMPOSITION ISSUES DEMONSTRATED

In this section, we demonstrate the occurrence of issues caused by language limitations, based on a concrete example. For this purpose, we discuss the object-oriented "State" design pattern [10], which realizes a state machine.

Figure 1 shows a concrete instance of the State pattern that (partially) implements the TCP/IP protocol. Based on this figure, which is very similar to the example found in the Design Patterns book [10], we identify several issues.

First, the State pattern has to be instantiated and tailored to this specific application. In general, many patterns (including the State pattern) specify roles that have to be mapped to implementation-specific classes. However, parts of these classes represent common pattern-defined behavior, made specific for a particular instantiation of the pattern. This obfuscates the generality of the design pattern, decreases the

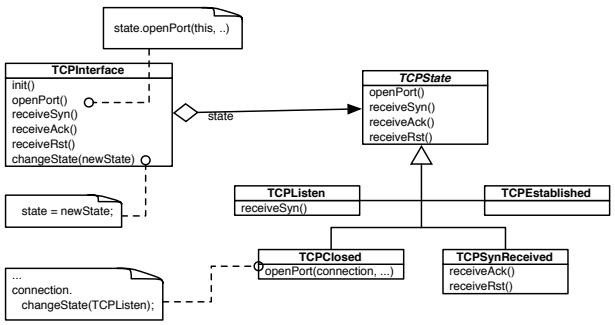

Figure 1: State pattern instantiation partially representing TCP/IP

separation between application-specific and pattern-generic code, and makes it harder to reuse common parts of the pattern implementation ("boilerplate" code).

Specifically in this example, the most important occurrence of boilerplate code is found in the methods defined in class `TCPInterface`. For each action (e.g., `openPort`, `receiveSyn`, etc.) supported by the state machine, this interface class has to define a method that forwards calls to the currently active state. As shown in figure 1, the forwarding method has to pass the `this` reference to the state object, such that it can, e.g., call `changeState` on the context object. Similarly, whenever a new action has to be added to the state machine, additional boilerplate code has to be added to both the State superclass `TCPState`, as well as `TCPInterface`. Both issues can be addressed more concisely in languages that support explicit delegation [22].

Second, pattern implementations may impose limitations on the way a particular concept is expressed. In the case of the State pattern implementation as shown in figure 1, the behavior associated with each state is modularized. However, state transitions are encoded as part of the actions, and thus become scattered over multiple State implementation classes.

As an alternative, the State pattern therefore also explicitly suggests that all state transitions can be kept in a single location, e.g., a transition table. This addresses the scattering of transition statements over the program, thus making it easier to, e.g., check whether an instantiation of the State pattern matches a corresponding state diagram, or to modify several transitions in one go. However, in many languages this alternative requires additional boiler-plate code, as is shown in figure 2.

In this alternative design, the constructor of `TCPInterface` constructs a table of state transitions. The methods in class `TCPInterface` each call the method `changeState(..)`. This method is parameterized by the action that is being executed, which is needed to look up the "next" state in the transition table. Similar to the code that "manually" forwards method calls to the current state object, the invocation of method `changeState`, passing the action, is thus replicated for each action supported by the state machine implementation.

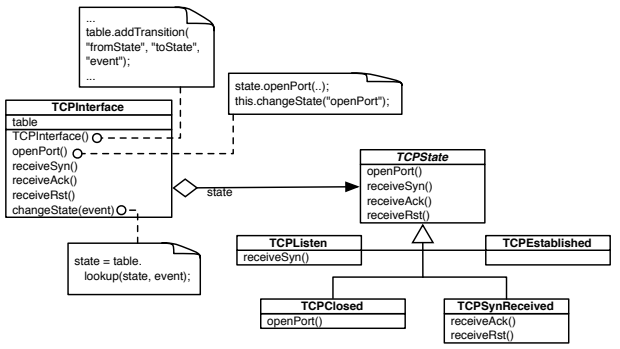

Figure 2: State pattern implementation using a transition table

One way to improve on the situation described above, is by addressing limitations in the underlying implementation language. In this case specifically, both implementations can benefit from a language that supports explicit delegation, whereas the table-based design can additionally benefit from a language that supports pointcut-advice constructs. We demonstrate this in detail in the next section; it should be noted, however, that this example is meant as a demonstrator for the usefulness of more flexible composition operator support in general, even if space limitations prevent us from discussing other examples here.

3. USING COMPOSABLE COMPOSITION OPERATORS

Our approach is based on a composition infrastructure, which supports composition primitives that allow programmers to design custom or domain-specific composition operators. This infrastructure is implemented as an object-based language called *Co-op*, which is discussed in detail elsewhere [14, 15].

Figure 3 shows a schematical overview of a *Co-op*-based design of the State pattern, applied to the TCP/IP example. As is the case with the original OO pattern, it supports both design alternatives discussed in the previous section (i.e., encoding transitions as part of the state implementations, or as a separate transition table). The lower half of the diagram contains the reusable, pattern-generic parts, which should fulfill two main tasks. First, it establishes and controls a delegation relation between a *context* object (as it is called in the original pattern description; in our example, class `TCPInterface` fulfills the role of *context*) and state implementation objects (instances of `TCPClosed`, `TCPListen`, etc.). Second, in case a table-based implementation is desired, it automatically ensures that the specified state transitions are executed at the appropriate moment, i.e., without adding any invocations to the application-specific state implementations (in the upper half of the diagram).

Below, we discuss each module described in the diagram in some detail, and show how this approach reduces the amount of boiler-plate code in the application-specific part.

Listing 1 shows the implementation of the generic, reusable parts of the State pattern. An instance of the State pattern can be created by constructing an instance of module

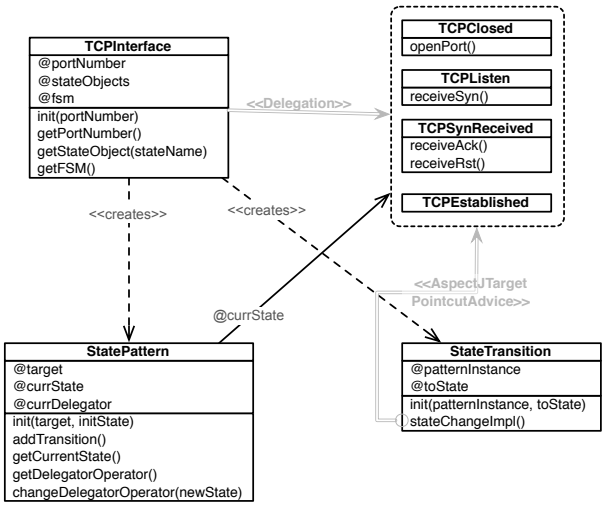

Figure 3: Design diagram of the *Co-op*-based State pattern

`StatePattern`. To facilitate the delegation from the context object to an object representing the current state, each state pattern instance keeps references to the `context` and `currentState` objects, and a reference to the delegation operator[2]. These instance variables of the pattern are defined on line 2. The constructor (lines 4–8) calls the operation that establishes the delegation relation (line 7). This operation, `changeDelegatorOperator(..)` (lines 10–18), which should be invoked whenever a state change is required, deactivates the delegation to the current state object (lines 12–13), and activates a new delegation relation from the context object to the new state object (lines 16–17).

We lack the space to discuss the internal details of the `Delegation` operator in detail, but a discussion of this exact operator can be found in prior work [14]. Here, it suffices to know that the constructor of the `Delegation` module establishes and activates delegation from the object referenced by the first argument (here: `@context`) to the object referenced by the second argument (here: `newState`). Effectively, this means that invocations on the context object are forwarded to the indicated state object, while the "this"-object still refers to the context object (i.e., `this`-calls are all directed to the context object).

Finally, lines 20–22 implement the state transition mechanism used in the table-based pattern implementation: by invoking `addTransition`, a pointcut-advice instance is constructed, which triggers `after` the specified `action` is invoked on the `fromState`. Whenever the pointcut triggers, as an advice the operation `stateChangeImpl` is invoked, as defined on lines 33–35, which changes the state to the selected `toState`. The module `StateTransition` stores references to the pattern instance, as well as the desired `toState`, so that these can be used by the advice[3].

[2]In *Co-op*, operators are themselves implemented as modules, and can be referenced as first-class objects. For a detailed explanation, see [14].

[3]Ideally, these could be supplied as advice parameters, mak-

```
module StatePattern {
 var context, currState, currDelegator;

 initWithContext:aContext initState:initState {
  context = aContext;
  currState = initState;
  this changeDelegatorOperator: initState;
 }

 changeDelegatorOperator:newState {
  // Deactivate the existing delegation (if any)
  (currDelegator isDefined) ifTrue:
   [currDelegator deactivate];

  // Active delegation to new state object
  currState = newState;
  currDelegator = Delegation newFrom: context to:
       newState;
 }

 addTransitionFrom:fromState action:action to:toState {
  AspectJTargetPointcutAdvice new: "after" matchTarget:
       fromState matchOperation:action aspectInstance: (
       StateTransition new:this to:toState) advMethod: "
       stateChangeImpl";
 }
 ... // trivial accessors not shown here
}

module StateTransition {
 var patternInstance, toState;

 init:aPatternInstance to:aToState {
  patternInstance = aPatternInstance;
  toState = aToState;
 }
 stateChangeImpl {
  patternInstance changeDelegatorOperator: toState;
 }
}
```

Listing 1: *Co-op*-based implementation of the State pattern

As mentioned in section 2, we found two types of functionality in the State pattern that can be made more reusable, while also removing the need for a lot of boilerplate code. First, by using delegation, it is no longer necessary to write manual forwarding operations in the context class (here: `TCPInterface`). Second, when using a table-based implementation, the pointcut-advice composition operator removes the need to manually invoke a method that decides about the next state. Note that although our approach *allows* the use of such a transition table, this is by no means obligatory; embedding transitions in action implementations works fine, as well. However, when transition tables are used, our approach removes the need for boilerplate associated with the original implementation.

Listing 2 shows how the State pattern implementation defined above as a custom, "pluggable" composition operator that can be used in any *Co-op* program, is applied to the TCP/IP example discussed in section 2.

```
module TCPInterface {
 var portNumber, fsm;

 init:aPortNumber {
  var closedState, listenState, synReceivedState,
       establishedState;

  portNumber = aPortNumber;
 closedState = TCPClosed new;
 listenState = TCPListen new;
 synReceivedState = TCPSynReceived new;
 establishedState = TCPEstablished new;

  fsm = StatePattern newWithContext: this initState:
       closedState;

  fsm addTransitionFrom: closedState action: "openPort"
       to: listenState;
  fsm addTransitionFrom: listenState action: "receiveSyn
       " to: synReceivedState;
  fsm addTransitionFrom: synReceivedState action: "
       receiveAck" to: establishedState;
  fsm addTransitionFrom: synReceivedState action: "
       receiveRst" to: listenState;
  // ...additional transitions not shown here
 }
 getPortNumber { return portNumber; }
}

module TCPClosed
{
 openPort {
  Console writeln: "TCPClosed: opening port: " with: (
       this getPortNumber);
 }
}

module TCPListen
{
 receiveSyn {
  Console writeln: "TCPListen: received SYN; sending SYN
       -ACK";
 }
}
// etc. for other TCP states not shown here
```

Listing 2: Application of the generic State-pattern implementation

In this listing, the constructor of module `TCPInterface`, found on lines 4–21, sets up the state machine: it creates an instance of each TCP state modeled in this example (lines 8–11), and initializes a State pattern instance (line 14), appointing itself as the context object, and setting `closedState` as the initial state object. In this example, we also used the pointcut-advice based transition mechanism, which is initialized in lines 16–19. Note that all the initialization code here is completely application-specific, and also, no boilerplate related to the internal "machinery" required by the pattern implementation is visible. Once the state machine is thus set up, the delegation and pointcut-advice operators automatically take care of effectuating the desired state machine behavior.

The remaining code in listing 2 shows the mock-up state implementations. Note that the state modules do not contain or need any references to the state pattern. Still, because of delegation, you can still use behavior of class `TCPInterface` by means of *this*-calls, such as `this getPortNumber` (line 27).

An example demonstrating how the complete state machine can be instantiated and executed is shown in listing 3. Note that in listing 3, no boilerplate code or references to the state pattern are necessary either.

```
module Main {
 main { var tcpserver;
  tcpserver = TCPInterface new: "80";
  tcpserver openPort; // Request port open
```

ing the module `StateTransition` superfluous, but our much simplified implementation of AspectJ-like pointcut-advice does not support this at present.

```
5     tcpserver receiveSyn; // Receive incoming conn.
6     //etc.
7 } }
```

Listing 3: Using the state machine implementation

When the state machine is initialized (line 5), calls will be delegated to the initial state, an instance of `TCPClosed`. Thus, when `openPort` is invoked (line 6), the call is delegated to the operation `openPort` in `TCPClosed`, as shown before. After this action has been executed, the pointcut-advice that executes the state transition to `listenState`, an instance of `TCPListen`, is automatically invoked, since it triggers *after* the invocation of `openPort` on the instance of `TCPClosed`. Thus the state machine implementation automatically delegates calls to the appropriate implementation, and automatically triggers state changes.

The complete example as well as a prototype *Co-op*-interpreter (a plain jar-file, no installation required) can be downloaded from the *Co-op* website [1].

4. RELATED WORK

The work in this paper is related to a large body of research on defining new languages that support novel composition techniques, especially in the domain of object-based and aspect languages. Many papers also present a (small) set of composition techniques that aim at unifying existing ones. However, *most* of such related research proposes a *fixed set of composition operators*, presented as part of a language, extension of a language, or an application framework. In contrast, our work focuses on a language that has no—or just one—built-in composition operators, but rather is a platform for constructing a wide range of user-defined composition operators.

To the best of our knowledge, there are no other languages that offer *dedicated support* for user-defined composition operators (that can be reused and combined), at least not within the domain of object-oriented and aspect-oriented languages. Please note that this excludes languages that offer generic extension mechanisms—such as macros in Lisp—or allow for the extension and modification of the program through metaprogramming; our work is particularly related to metaprogramming [6] and especially meta-object protocols [21]. As explained, e.g., in [20], the power of metaprogramming comes with more complexity and responsibility.

This means that the difficulty of language design—except for the concrete syntax—is now on the MOP designer. Indeed, our work might just as well have been presented as a novel design of a MOP, but for practical reasons we chose to use a concrete language, *Co-op*. We are not aware of any MOPs (or languages, or frameworks) that offer similar generic abstractions and structure as we presented in this paper. In particular, we do not know any MOPs that provide abstractions for defining new composition operators with similar variety, expressiveness and composability. For example, *Co-op* explicitly supports a variety of object-oriented as well as aspect-oriented composition operators.

Of the research that aims at providing frameworks for higher-level languages through reflection or meta object protocols, we just mention COLA [24], AspectS [18], MetaClassTalk [4]: please refer to [14] for a discussion of these. There are several frameworks that aim at offering a generic platform for OO and AOP language implementations. For such platforms, the designers have typically made efforts to find a small set of generic constructs that typically serve as a target 'language' for a compiler/code transformation. An important distinction with our work is that these platforms do not aim at, and hence do not support, the ability of creating user-defined composition operators within the same language.

We have used the example of a modular, reusable implementation of a design pattern to exemplify that a single fixed composition technique is insufficient, while at the same time demonstrating that a design pattern implementation can in fact be modeled as a composition operator that 'extends' the language.

In [3], Bosch argues that language support is needed for explicit representation of design patterns in programming languages. The LayOM language offers a number of common design patterns as built-in constructs. These can be extended by growing the language, which supports modular extension of the lexer, parser and code generators for a new pattern: in contrast to our approach, the extension is not specified in the programming language itself. Also in [17], techniques for explicit representation of design patterns are proposed that are based on extension of the language and, consequently, the compiler.

Rajan and Sullivan [25] argue that design patterns are a suitable test case for evaluating and comparing aspect languages, because (1) design patterns are standard, well-documented design structures, and (2) existing examples [13] of design pattern implementations in AOPLs are available. They base their evaluation of the EOS language on a comparison with the AspectJ implementations of patterns in [13], following the metrics that have been proposed by Garcia et al. in [11].

Several efforts have been made to represent design patterns as first-class entities. For example, in [9], the fragment model is introduced to represent design patterns and their components. The FACE approach [23] extends the OMT notation with pattern-specific entities. Similarly, [26] proposes a modeling notation for representing design patterns—specifically for the support of the design and integration of object-oriented frameworks. All of these approaches build on the assumption that a design pattern has roles, which must be filled in by entities that use the design pattern. These roles are called participants in [10].

Hanneman and Kiczales [13] shows how to implement the GoF design patterns using aspect-oriented modularization techniques; in several cases this enables the modularization of all pattern-generic code within a single module (aspect).

5. EVALUATION AND CONCLUSION

Support for flexible, user-definable composition operators can help to improve the modularity and reusability of design pattern implementations, as we have shown for the State design pattern specifically in this paper.

Although this paper shows only one example, the results can be generalized. As has been discussed by the example of the Visitor pattern in section 1, or by the example of other patterns in [16, 15, 2, 13], rich composition operators in the language provide a powerful way to solve problems which are typically only "worked-around" by means of Design Patterns, i.e., requiring boilerplate code in several locations.

In addition, our approach of using a composable composition infrastructure (called *Co-op*) allows the definition of new composition operators that reuse existing ones. We have shown this by expressing the State design pattern as a custom composition operator, which reuses two existing operators that implement explicit delegation and a basic pointcut-advice mechanism. Also as a result of this, we could define such a relatively complex and reusable operator in less than 50 lines of code.

6. REFERENCES

[1] Co-op homepage, `http://wwwhome.cs.utwente.nl/~havingaw/coop/`, 2008.

[2] C. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, 2009.

[3] J. Bosch. Design patterns as language constructs. *JOOP*, 11(2):18–32, 1998.

[4] N. Bouraqadi, A. Seriai, and G. Leblanc. Towards unified aspect-oriented programming. In *Proceedings of ESUG 2005 (13th international smalltalk conference)*, 2005.

[5] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, 2006.

[6] P. Cointe. Reflective languages and metalevel architectures. *ACM Computing Surveys*, 28-4, 1996.

[7] M. Dominus. Patterns are signs of weakness in programming languages, `http://blog.plover.com/prog/design-patterns.html`.

[8] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN international conference on Functional programming*, pages 94–104, New York, NY, USA, 1998. ACM.

[9] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, 1997.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[11] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. In P. Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 3–14. ACM Press, Mar. 2005.

[12] D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 116–129, New York, NY, USA, 2004. ACM.

[13] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[14] W. Havinga, L. Bergmans, and M. Aksit. A model for composable composition operators: Expressing object and aspect compositions with first-class operators. In *Proceedings of the 9th international conference on Aspect-Oriented Software Development*, Mar 2010.

[15] W. K. Havinga. *On the Design of Software Composition Mechanisms and the Analysis of Composition Conflicts*. PhD thesis, University of Twente, Enschede, June 2009.

[16] W. K. Havinga, L. M. J. Bergmans, and M. Akşit. Prototyping and composing aspect languages: using an aspect interpreter framework. In *Proceedings of 22nd European Conference on Object-Oriented Programming (ECOOP 2008), Paphos, Cyprus*, volume 5142/2008 of *Lecture Notes in Computer Science*, pages 180–206, Berlin, 2008. Springer Verlag.

[17] G. Hedin. Language Support for Design Patterns Using Attribute Extension. In *Proceedings of the Workshops on Object-Oriented Technology*, pages 137–140. Springer-Verlag London, UK, 1997.

[18] R. Hirschfeld. Aspect-oriented programming with AspectS. In M. Akşit and M. Mezini, editors, *Net.Object Days 2002*, Oct. 2002.

[19] R. Johnson. Design patterns and language design, `http://www.cincomsmalltalk.com/userblogs/ralph/blogView?entry=3335803396`.

[20] G. Kiczales. It's not metaprogramming. *Software Development Magazine*, (10), 2004.

[21] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[22] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[23] T. D. Meijler, S. Demeyer, and R. Engel. Making design patterns explicit in FACE: a frame work adaptive composition environment. In *ESEC '97/FSE-5: Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 94–110, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[24] I. Piumarta and A. Warth. Open, extensible object models. In *Self-Sustaining Systems*, volume 5146/2008 of *Lecture Notes in Computer Science*, pages 1–30. Springer, Springer Berlin/Heidelberg, 2008.

[25] H. Rajan and K. Sullivan. Design Patterns: A Canonical Test of Unified Aspect Model. Technical report, Iowa State University, 2005.

[26] D. Riehle and T. Gross. Role model based framework design and integration. *SIGPLAN Not.*, 33(10):117–133, 1998.

Generic IDE Support for Dispatch-Based Composition

Christoph Bockisch
Software Engineering group – University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands
c.m.bockisch@ewi.utwente.nl

Andreas Sewe
Software Technology group – Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany
sewe@st.informatik.tu-darmstadt.de

ABSTRACT

Programming-language research produces a significant number of new programming styles to improve the composability of programs. This increases re-usability as well as other quality characteristics. But although they offer interesting composition concepts, new programming languages are rarely used because IDE support, which developers are used to, is missing. Examples of such IDE support are the visualization of call hierarchies or interactive debugging. While some languages, e.g., AspectJ, eventually reach a more mature level with elaborate IDE integration, not all language designers are able to invest this much effort towards IDE integration. Furthermore, the IDE integration of AspectJ also has its limitations; when debugging, the developer is confronted with synthetic code with no exact correspondence in the source code. As a result, the developer needs to understand the transformations performed by the compiler. Finally, some information invariably gets lost during weaving, e.g., the ability to map code evaluating pointcut designators to their definition in the source code.

In this paper, we propose to implement generic IDE tools for programming languages that provide advanced dispatching mechanisms. Such languages, including predicate dispatching and pointcut-advice languages, can be mapped to our execution model, called ALIA. The same execution model can then drive debugging functionality as well as static IDE services.

1. INTRODUCTION

In order to improve the modularity of source code, research strives to define new composition mechanisms, often in terms of new languages. Many such languages provide composition mechanisms by allowing to influence the dispatch of, e.g., method calls, like in multiple dispatching [9] or predicate dispatching [14]. But other composition styles can be mapped to a dispatching-based execution model as well, as we have shown [5] for pointcut-advice languages [15], Composition Filters [12], and a DSL for composing objects following the Decorator design pattern.

Usually, advanced dispatching mechanisms are provided as an extension of an existing programming language, the so-called base language, and the semantics of the advanced program features are realized by transforming them to the base language's imperative code. We have shown [5] that the dispatching mechanisms of all these languages share concepts from several broad categories: selection of call sites based on syntactic properties, access to the runtime state in which they are executed, evaluation of functions over the runtime state to select from alternative meanings, declaration of meaning in terms of actions on the runtime state, and description of relationships between applicable actions. Each language uses some extension of each core concept and the concrete concepts used in different languages often overlap.

Similarly, the requirements for IDE support of such languages overlap. Different kinds of support for the development in the investigated languages are recurring, but have to be implemented from scratch for each language. As a result, the IDE support for new languages is typically limited, as more effort is spent on the design of the language and the implementation of compilers than on the language's IDE integration. In the following, we discuss a few examples of IDE support from which all investigated languages can benefit.

Among the investigated languages, the aspect-oriented ones support implicit invocation. It thus is desirable to let the IDE visualize the places in the code at which other code may be (implicitly) the target of dispatch. To this end, the IDE support for the AspectJ language, the AspectJ Development Tools (AJDT), provides different ways of visualizing such relations. However, even for languages with only explicit invocation, similar IDE support is present. The Eclipse Java Development Tools (JDT) allow, e.g., to search for all call sites of a method, or to show the possible targets of a call site. It should be noted that, while calls must be explicit in Java, they can be virtual and multiple implementations may be applicable. The potential targets depend on the inheritance hierarchy, which may be too complex for the developer to grasp in its entirety. IDE support is therefore essential. The same observation holds for predicate-dispatching languages.

All investigated languages can be compiled to pure Java bytecode and can run on a standard JVM. Therefore, the default debugger of the IDE can be used to debug programs written in those languages. What is debugged, however, is the program *after* the transformation. Consequently, the developer is facing large amounts of infrastructural code that has been inserted by the compiler to realize the semantics and will often end up stepping through code for which no source code exists, which makes it even harder to understand. Another difficulty is that the Java debugger assumes that all code of a class was compiled from a single source file, but with new composition mechanisms this assumption may no longer hold: one class may be composed of multiple source files. Compilers merge all files into one; thus, the mapping from target code to the source code is lost and cannot be used by the debugger anymore.

We have provided an architecture for implementing advanced-dispatching languages in a way that they can share the implementation of overlapping concepts [5]; it is called the *Advanced-dispatching Language-Implementation Architecture* (ALIA) and consists of a lan-

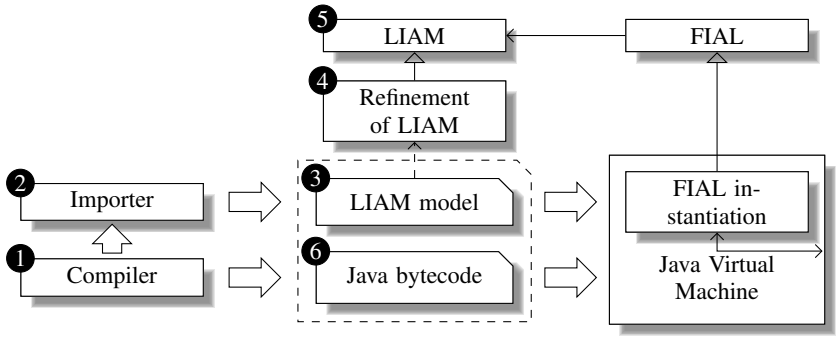

Figure 1: Overview of the application life cycle in ALIA4J-based language implementations.

guage-independent meta-model of advanced dispatching concepts and any number of execution environments that process models conforming to this single meta-model. For languages extending Java we have implemented this architecture, called ALIA for Java (ALIA4J), which furthermore provides a framework factoring out shared components of such execution environments.

In this position paper, we will discuss how ALIA's meta-model, more specifically its implementation in ALIA4J, and the framework for execution environments can be used to provide a generic infrastructure for IDE support of advanced-dispatching languages.

2. THE ALIA ARCHITECTURE FOR JAVA

In ALIA4J, the meta-model stipulated by ALIA is embodied in the *Language-Independent Advanced-dispatching Meta-model* (LIAM). LIAM hereby acts as the form of intermediate representation for advanced dispatching in programs. The actual intermediate representation, in turn, is a model conforming this meta-model (the so-called *LIAM model*). Code of the program not using advanced dispatching mechanisms is represented in its conventional Java bytecode form. The *Framework for Implementing Advanced-dispatching Languages* (FIAL) implements common components and work flows required to implement execution environments based on a JVM for executing LIAM models. A brief overview, of the approach can be found in [7][1].

Figure 1 shows an overview of the ALIA4J approach. Concretely, the flow of compiling and executing applications in this approach is shown. The compiler ❶ starts processing the source code; a dedicated importer component ❷ adapts the compiler's output to a model for the advanced dispatch declarations in the program ❸ based on the refined subclasses ❹ of the LIAM meta-entities ❺. Furthermore, the compiler produces an intermediate representation of those parts of the program that are expressible in the base language ❻ alone.

The nine meta-entities of LIAM capture the core concepts underlying the various dispatching mechanisms, but at a finer granularity than the concrete concepts found in high-level languages; one concrete concept often maps to a combination of LIAM's core concepts. Figure 2 shows the meta-entities in LIAM, which are implemented as abstract classes. Attachment, specialization, and predicate are an exception to this rule, i.e., they are concrete classes, as they provide logical groupings of entities of the meta-model and cannot be refined. The meta-entities are discussed in detail in [5, Chapter 3.2][2].

In short, an attachment corresponds to a unit of dispatch description. In terms of aspect-orientation (AO), this is a pointcut-advice

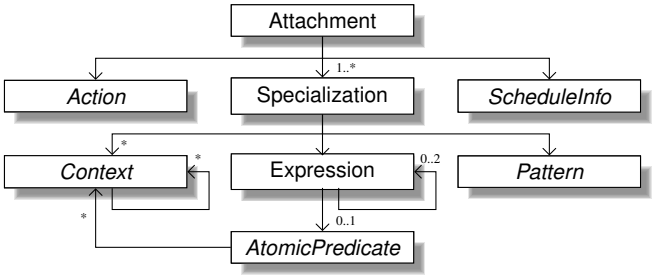

Figure 2: Entities of the Language-Independent Advanced-dispatching Meta-Model (LIAM) as UML class diagram.

pair, in terms of predicate dispatching, an attachment corresponds to a predicate method. Action specifies an action to which the dispatch may lead (e.g., an advice or the predicate-method body). Specialization defines static and dynamic properties affecting dispatch: patterns specify syntactic properties of call sites which are affected by the declared dispatch; predicate and atomic predicate entities model dynamic properties a dispatch depends on (dynamic pointcut designators in AO terminology). Context entities model access to values in the context of a dispatch, like the calling object or argument values. Finally, the schedule information models constraints between multiple actions applicable at the same generic-function call. This includes the order of their execution, as well as relations like mutual exclusion.

At runtime, FIAL derives a dispatch model for each dispatch site in the program from all attachments that have been defined. Thereby, FIAL solves the constraints specified as schedule information and derives a single dispatch function per call site from the predicates of all specializations. This function is represented as a binary decision diagram (BDD) [8], where the inner nodes are the atomic predicates used in the predicate definitions and the leaf nodes are labeled with the actions to be executed. For each possible result of dispatch, the BDD has one leaf node. Figure 3 shows an example of such a dispatch model with the atomic predicates x_1 and x_2 and the actions y_1 and y_2. For a detailed explanation of this model, we refer the reader to [18].

The dispatch model is defined in such a way that an execution strategy can immediately be derived from it. The default execution strategy requires that each concrete entity implementation provides a Java method implementing its semantics. It is possible to override the default strategy and implement an optimization strategy on a per atomic predicate basis, in a modular way. These strategies are extensively discussed in [5].

The approach allows to implement new concrete concepts modularly by refining the abstract class of a meta-entity. We have already shown how to map the languages AspectJ, JAsCo, Compose*, Cae-

[1]Some details presented in [7] are outdated, but it may nevertheless act as an introduction to the basic concepts.

[2]There, some meta-entities are named differently, but the structure of the meta-entities is the same. Therefore, the interested reader will be able to map the discussion to the new names.

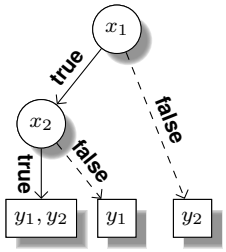

Figure 3: A dispatch function's evaluation strategy.

sarJ, and a simple domain-specific language to our meta-model [5]. By now, we have also developed mappings for the languages MultiJava [10], JPred [16], and ConSpec [4], which are, however, still unpublished work.

Important for the present paper is the fact that all concrete concepts participating in a dispatch are expressed in a declarative and fine-grained model. This model can thus be used to derive information relevant to the different services of an IDE. Furthermore, the model stays first-class during the execution of a program and can therefore easily support dynamic features of an IDE, e.g., debugging, profiling, or testing.

3. ALIA4J-BASED IDE SUPPORT

So far, we have implemented a limited IDE integration for the ALIA4J mapping of AspectJ language. However, we aim at making this support more general and support other languages, too. Moreover, we aim at filling the gaps in our IDE support.

3.1 Cross References for AspectJ

For the AspectJ language, we have implemented a nearly complete integration with our architecture. All necessary LIAM entities are implemented and we have developed an automatic importer component which allows to execute AspectJ programs on an FIAL-based execution environment while developing it in the standard AJDT. The benefit of this integration is that some FIAL-based execution environments perform sophisticated dynamic optimizations which make AspectJ programs execute faster than the product of the standard compiler [6].

Because we bypass the weaving phase of the AspectJ compiler in this approach, pointcuts are not evaluated at compile time anymore. Thus, the compiler also cannot determine the crosscutting structure of the aspects which would normally be used by the IDE to show, e.g., in the "Cross References" view, or which would be used to facilitate navigation between advised join point shadows and their advice, as depicted in Figure 4. To restore the accustomed functionality, we have developed an extension to the AJDT that provides the crosscutting structure when compiling AspectJ applications for execution on a FIAL-based execution environment, i.e., without compile-time weaving. This comprises an instantiation of the FIAL framework which is, however, not integrated in a Java Virtual Machine like a full-fledged FIAL-based execution environment. Instead of providing FIAL with dynamic information about generic-function calls, it provides static approximations of call sites. Our framework then evaluates all patterns in the LIAM models of the AspectJ project and builds the dispatch model for each call site. Afterwards, for dispatch models which are not trivial, i.e., where no advice is attached, the links are established in AJDT's abstract structure model.

To support this work, the LIAM meta-entities are extended to also store the location in the source code where they are defined. This is similar to the debug information present in Java bytecode. This debug information facilitates recovery of the file name and line number

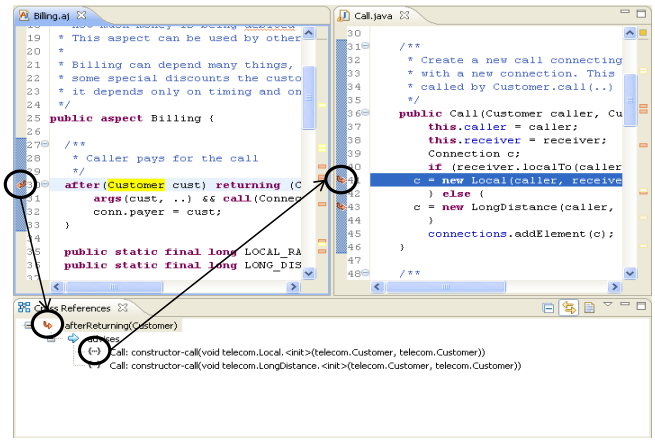

Figure 4: Linking of pointcut-advice and advised locations in AJDT.

whose compilation has lead to a bytecode instruction, respectively in the case of LIAM to a model entity. The builder uses this information to establish links between source locations, as is stipulated by the AJDT abstract structure model.

In the current version, input is hard-wired to the AspectJ compiler's output. But since our architecture already provides a plug-in mechanism to provide input in different formats, a straight-forward extension is to use this mechanism. Then, the same support can be provided for any language that can be mapped to our approach.

While the above AJDT extension shows the feasibility of building static tool support based on our ALIA approach, we do not aim to extend the AJDT in our future research work. Instead, we will re-implement similar support, including an AJDT-like structure model and related views, by directly extending the Java Development Tools (JDT). This is necessary because the AJDT and the structure model are hard-linked to the AspectJ compiler, a dependency that we would rather avoid. Furthermore, we have already outlined that there are commonalities between the cross references view and, e.g., the call hierarchy of methods explicitly called. Since both concepts, explicit and implicit calls, are unified in our architecture, we like to provide IDE support for both in the same way. The developer will benefit from such unified tools, because he will see all contributors to a call at the same time.

3.2 Debugging Support

The debugging support we envision will be based on the availability of our declarative dispatch model at runtime. For example, this makes it possible to visualize the dispatch model for a call at a breakpoint. The dispatch model is complete in the sense that it specifies on which runtime values the dispatch depends and which predicates are evaluated on these values. While the model, naturally, only specifies the role of values that are used (e.g., "the first argument value"), in a debugger, also the value can be shown. This is already done by modern debuggers, e.g., in the "Variables" view of the Eclipse debugger. In contrast to general-purpose debuggers, our debugger for advanced dispatch will only show values relevant for the dispatch, and associate them to their role names.

Another contribution that results from using the ALIA approach to enable debugging is that dispatch declarations defined in different languages can be combined. Since all dispatch declarations (pointcut-advice, multi-methods, etc.) are mapped to the same meta-model, i.e., to LIAM, the actions resulting from these declarations can be

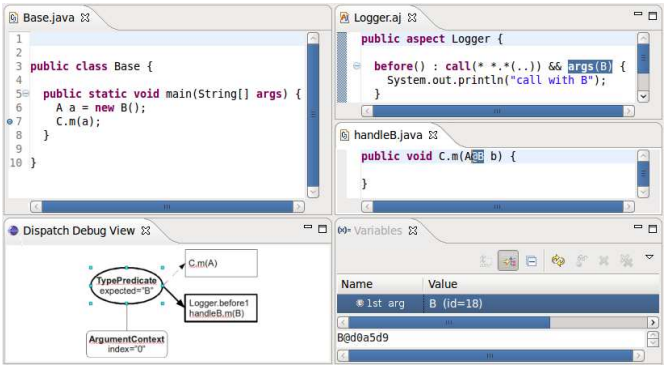

Figure 5: An idea of the GUI for generic debugging support for multiple advanced-dispatching languages.

executed alongside. That means that, e.g., calls to multi-methods can be advised.

Figure 5 illustrates the envisioned visualization in debugger. At the top of the figure, three editors are shown. The editor at the left-hand side shows Java code calling the method C.m(A) in line 7; at this line, a breakpoint is set. At the right-hand side, the top editor shows an AspectJ pointcut-advice and the bottom editor shows a multi-method defined in MultiJava. Both dispatch declarations define a dynamic constraint on the first argument: Only when this is of type B, the advice is to be executed, respectively, the multi-method applies. In this case, the multi-method overrides the Java method definition.

The bottom part of Figure 5 shows a possible visualization of the (simplified) dispatch model for the call at the breakpoint. The dispatch function is simple and only contains one atomic predicate, which tests the type of a context value, in the example that of the first argument. The bold elements show the path which the evaluation actually has followed. The bold solid arrow emerging from the predicate indicates that it has been satisfied, therefore, the actions in the bold box are to be executed as the dispatch's result, i.e., the actions Logger.before1 and handleB.m(B). If the predicate was satisfied, the action C.m(A) would be executed.

In a graphical debugger as proposed here, the user can select and introspect entities that participate in the dispatch at which the virtual machine is currently suspended. In the figure, the TypePredicate is selected. The selection in the editors showing the AspectJ and MultiJava code highlights the code which has led to this predicate in the dispatch function. The "Variables" view shows the runtime values on which the current selection depends, i.e., the first argument value. As can be seen, this is an instance of B and therefore, the predicate is satisfied.

As outlined above, the entities in the dispatch model can be linked to multiple source locations. The result of single atomic predicates in the dispatch function can be presented, which explains the result of the dispatch. Potentially, it will be advantageous not to completely evaluate the dispatch function and let the developer view the result afterward, but to allow a step-wise evaluation of the dispatch function. We will investigate both approaches.

3.3 Additional Ideas

The AJDT provides the developer with more detailed information than just "these advice apply to this join-point shadow". It already includes additional information by specifying whether the join-point shadow is always affected by an advice or only sometimes because there is a dynamic pointcut designator in the matching pointcut. Also, when showing the applicable advice, the AJDT orders them according to their precedence.

Nevertheless, we envision to increase the provided information in several ways. First, it is interesting to specify not only *that an advice is conditional* but also, *what the condition is*. Next, presenting a sequential list of advice is too limited because some languages support more complex relations between advice at a join-point shadow. AspectJ, e.g., already provides "around" advice which can be nested; thus, a tree would be more suitable to present this information. Other languages allow to define more complex relationships between advice at a shared join-point shadow. Examples are mutual exclusion or conditional execution in Compose*, or overriding in JPred and MultiJava.

The dispatch model, explained in some detail in the previous subsection, can also be made available before runtime. A visualization of the cross references can, thus, take all information in the dispatch model into account. This includes the exact specification of the condition under which an action is applicable at a call, dispatch declarations sharing the call site and relationships (order, execution constraints, etc.) among them.

Since the implementation of our architecture, i.e., FIAL and LIAM, is very modular, it is also easily possible to make part of their implementation interactive. A possible use is making pattern matching interactive in order to debug patterns. The AJDT shows the developer in which places pointcuts match, but in some cases, developers of pointcut-advice may wonder *why* a specific pointcut (respectively the pattern used in a pointcut) does or does not match. Since the definition of specializations (the equivalent to pointcuts in AspectJ) and the call sites are available first-class in FIAL, it is possible to perform the evaluation, e.g., for a specific call site, and show the developer the different steps in the evaluation. This is similar to the debugging support for dispatch functions, but can be performed before runtime.

4. RELATED WORK AND FUTURE WORK

Eaddy et al. [13] have identified several requirements for debugging aspect-oriented programs. They support source-level debugging by deferring the weaving to runtime, as in our approach. It is thus possible to view the definition of pointcut-advice that have lead to the execution of a specific statement. In contrast to our approach, the dispatch function is not represented in a structured declarative way, but only by the imperative code resulting from the pointcut-advice definitions. Thus, the dynamic program state that has lead to executing or not executing an advice is more difficult to determine for the developer. Furthermore, the original definition of an aspect (or dispatch declaration) is not presented. Therefore, constraints among advice sharing this join-point shadow are not easily visible, and, thus, cannot be easily debugged.

Pothier et al. [17] discuss a retrospective debugging approach for aspect-oriented programs. They record a complete execution trace that can be inspected after the execution. While this is not the kind of debugging that we will support, we will nevertheless take inspiration from their work in order to present AO-specific visualization of debugging information.

De Borger at al. [11] define an architecture for implementing debuggers for aspect-oriented languages. This architecture is based on a structurally reflective model of aspect definitions. For each aspect that is active during the program's execution, its structure can be queried by means of this model. It is possible to determine the executions of advice, which are *caused* by a pointcut, including executions in the past and in the future. Their model is meant to be an API used by a debugger front-end and offers some infrastructure required by debuggers, e.g., to enable aspect-specific breakpoints.

Our underlying model is more fine-grained and provides more information: constraints among aspects like precedence are not available through the reflective API. Nevertheless, we plan to investigate whether their work can be used as an interface for our approach. It

may be possible that our back-end, i.e., a FIAL-based execution environment, can be used as an implementation of their API. Should we follow this path, we aim to contribute additional functionality to the API which can be provided by means of our back-end. Similarly, the IDE integration of debugging that we envision, may be implementable with their API as back-end.

The IDE Meta-tooling Platform (IMP) [2] is an Eclipse project aiming at providing meta-implementations of typical IDE tools. Examples are a re-usable infrastructure for syntax highlighting, refactoring support, semantic or static analyses, execution and debugging. Their focus is on providing an infrastructure for the IDE integration and the graphical user interface, but not on providing an infrastructure for the runtime part of actual debugger implementations. Nevertheless, we will consider to integrate our work with this project. Potentially, the LIAM meta-model can act as re-usable abstract syntax tree for dispatch declarations in the IMP. We hope to be able to re-use components for the more static IDE support like the visualization of implicit and explicit calls.

There are other Eclipse projects into which we may integrate our envisioned work. The first option is the Dynamic Languages Toolkit (DLTK) [1] which is a collection of frameworks to minimize the effort of developing IDEs for dynamic languages. The second option is the Textual Modeling Framework (Xtext) [3] which is a framework for generating full-fledged Eclipse text editors from grammars for domain-specific languages, including an abstract source code model.

5. CONCLUSION

In the suggested research work, we aim at providing a generic implementation of IDE support, most importantly containing debugging support, for advanced-dispatching programming languages. We will build this support on the FIAL framework and the LIAM meta-model (part of the ALIA architecture for Java), which provide a first-class, declarative model of all dispatches in a program. We have mapped the aspect-oriented languages AspectJ, CaesarJ, Compose*, JAsCo, the predicate-dispatching languages JPred and MultiJava, and other languages to this model. All mapped languages will thus be able to directly benefit from the IDE support we aim to provide.

The IDE support will primarily consist of a navigable visualization of explicit as well as implicit calls (the former are used in predicate dispatching, the latter in pointcut-advice languages), and of debugging support. Both kinds of IDE integration will be driven by the declarative, first-class dispatch model available in ALIA. Since ALIA facilitates the execution of dispatch declarations written in different languages, all such dispatch declarations can be executed in one program run alongside; similarly, the debugging support we envision will be able to debug all such declarations at the same time. It will facilitate to jump to the source code defining the dispatch declaration, and it will to show all execution steps leading to a specific dispatching result. We will investigate similar support for reasoning about the evaluation of patterns used in pointcut-advice, respectively for the composition of actions applicable at the same call site.

Providing such IDE support that will work "out of the box" will increase the acceptance of new programming languages which offer sophisticated composition mechanisms by means of dispatch declarations. The envisioned IDE support will make the effects of applying advanced composition mechanisms to a program more obvious to developers, which will help them to learn such new mechanisms. ALIA's ability to execute programs written in different languages with different composition primitives and the resulting IDE support, will give developers the free choice of combining different languages and benefit from all their features. We would also like to note that many composition mechanisms which do not obviously map to a dispatching problem can still be handled by our architecture. For example, we successfully mapped AspectJ's inter-type member declarations to our approach; in fact, the example used in section Section 3.2 uses the open classes feature of MultiJava, which is equivalent to inter-type member declarations.

6. ACKNOWLEDGMENTS

We would like to thank all contributors to the ALIA4J project. In particular, our thanks go to Jannik Jochem, who developed the integration of ALIA4J with the AJDT.

7. REFERENCES

[1] The Dynamic Language Toolkit. `http://eclipse.org/dltk/`, 2010.

[2] The IDE Meta-tooling Platform. `http://eclipse.org/imp/`, 2010.

[3] The Textual Modeling Framework. `http://eclipse.org/xtext/`, 2010.

[4] I. Aktug and K. Naliuka. ConSpec: A formal language for policy specification. In *Proceedings of REM*. Elsevier Science Publishers B. V., 2008.

[5] C. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, 2009.

[6] C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini. Efficient control flow quantification. In *Proceedings of OOPSLA*. ACM Press, 2006.

[7] C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *Proceedings of VMIL*, New York, NY, USA, 2007. ACM.

[8] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35, 1986.

[9] C. Chambers. Object-oriented multi-methods in cecil. In *Proceedings of ECOOP*. Springer Verlag, 1992.

[10] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. In *Proceedings of OOPSLA*. ACM, 1999.

[11] W. De Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *Proceedings of AOSD*. ACM, 2009.

[12] A. de Roo, M. Hendriks, W. Havinga, P. Dürr, and L. Bergmans. Compose*: a language- and platform-independent aspect compiler for composition filters. In *Proceedings of WASDeTT*, 2008.

[13] M. Eaddy, A. V. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Software Composition*. Springer, 2007.

[14] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP*. Springer Verlag, 1998.

[15] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP*. Springer Verlag, 2003.

[16] T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for Java. *ACM Transactions on Programming Languages and Systems*, 31(2), 2009.

[17] G. Pothier and E. Tanter. Extending omniscient debugging to support aspect-oriented programming. In *In Proceedings of SAC*. ACM, 2008.

[18] A. Sewe, C. Bockisch, and M. Mezini. Redundancy-free residual dispatch. In *Proceedings of FOAL*. ACM, 2008.

Aspect-Based Variability Model for Cross-Organizational Features in Service Networks

Stefan Walraven, Bert Lagaisse, Eddy Truyen & Wouter Joosen
DistriNet, Dept. of Computer Science
K.U.Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

ABSTRACT

Different clients have different needs, therefore adaptability and variability are crucial properties for service compositions to fit those varying requirements. This is hard to achieve in a cross-organizational context where services are implemented and deployed by different organizations (e.g. companies, administrative domains, ...): a feature, for example security, cannot be condensed into a single module that is applicable to all the different services. This paper proposes an aspect-based variability model for representing cross-organizational features in service networks such as systems of systems or service supply chains. We argue that cross-organizational features should be managed in a multi-layered architecture, distinguishing between policy and mechanism. Such a multi-layered architecture is completely lacking in AOSD currently. Based on this tenet, we first describe a technology-independent feature ontology that is well-defined for a domain or a specific service network and map it to an aspect-based feature implementation.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.2.11 [**Software Engineering**]: Software Architectures—*Service-oriented architecture (SOA)*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Documentation, Management

Keywords

AOSD, Variability modelling, Service engineering, Feature-oriented

1. INTRODUCTION

Recent trends in service engineering aim to combine the benefits of feature-based and service-based approaches [5, 14, 1, 19]. This combination increases the reusability of services and gives service consumers the opportunity to select different variants of a service. In addition, a service provider can provide fine-grained customization capabilities for a service, without having to create new services for each customization.

A feature is a distinctive mark, quality, or characteristic of a software system or systems in a domain [12]. Features define both common facets of the domain as well as differences between related systems in the domain. They make each system in a domain different from others. Features are also used to define the domain in terms of the mandatory, optional, or alternative characteristics of these related systems. Aspect-oriented software development (AOSD) [9] often has been put forward as a possible solution to enable modularization and composition of features [20, 17, 15].

However, services are mostly used in a service composition consisting of services from different organizations. In such a cross-organizational context, a feature cannot be condensed into a single feature module any more. The reason is that service implementations are black boxes, implemented and deployed by different organizations, and only the interface descriptions are available [1]. But this doesn't exclude the need to share semantically compatible features between those different services. A typical example of a cross-organizational crosscutting feature is security. When implementing an access control concern in an application, for instance, security actions need to be performed for every interaction between application components. However in a cross-organizational application, it is difficult to defend that a single module, for instance an aspect, should encapsulate the implementation of the internal security mechanisms of the organization involved as well as the global security policy governing how security must be addressed in the overall interaction between organizations. The latter security policy belongs to a level of abstraction above the internal security mechanism.

Therefore this paper proposes an aspect-based variability model for representing cross-organizational features in service networks such as systems of systems or service supply chains. We argue that cross-organizational features should be managed in a multi-layered architecture. Such a multi-layered architecture is completely lacking in AOSD currently.

The remainder of this paper is structured as follows. In section 2, we further illustrate and motivate the need for a

variability model for cross-organizational features in AOSD. Section 3 elaborates the overall approach and presents the application of the approach to an example. We discuss related work in section 4 and conclude in section 5.

2. MOTIVATION & ILLUSTRATION

In this section we further motivate and illustrate the importance of an aspect-based variability model for cross-organizational features in service networks.

We present an example in the e-finance domain (see Fig. 1). A bank offers a stock trading service to inspect, buy and store stock quotes. To be able to provide this service, it cooperates with the stock market, which in turn cooperates with a settlement company. So the stock trading service is a composition of the services provided by these three companies. Each participant can take up two roles in a composition: service consumer (client) and service provider (server). For example the bank company is a server for the bank customers, but consumes the `QuotesOrderService` of the stock market.

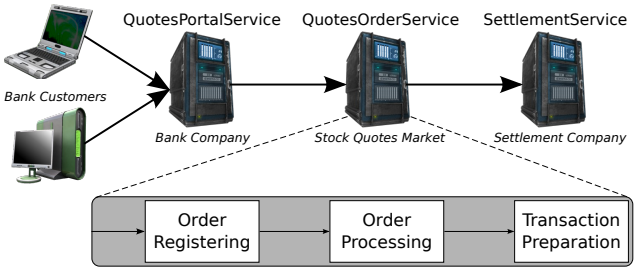

Figure 1: Illustration of the stock trading service composition.

During a typical session, a client inspects stock quote data, inspects the stored stock quotes in his custody account and potentially buys or sells some stocks. Clients can issue a stock order by using the web service portal facility of their bank. The bank service acquires the client's order and forwards it to the stock market. Processing the order request in the stock market consists of three sequential functional steps. Firstly, the client order is registered in the stock market by the `OrderRegistration` unit and then forwarded to the `OrderProcessing` unit. Secondly, at regular time intervals, the `OrderProcessing` unit searches for matches between buying and selling offers. If two orders match, they are forwarded to the `TransactionPreparation` unit, which delegates the actual trade of goods to the settlement company.

Since different clients have different needs, this service composition can be customized with respect to agreed features such as prioritized processing, billing, stepwise feedback, logging, non-repudiation, transaction support, secure communication, authentication, authorization and secure server-side storage. Choosing different features results in differents variants. If selected, the stepwise feedback feature, for instance, informs the client about the progress made in processing its requests, at the level of the individual services as well as the different sequential units within a service. Several alternatives are available for the stepwise feedback feature, such as feedback by email or by mobile text messages. Similarly, a client can select prioritized processing. By having this feature injected, the client's requests are prioritized over other requests. However, the prioritizing feature requires the billing feature: prioritizing requests comes at an expense.

Figure 2 presents the stock trading service composition including the prioritized processing feature. We see that the stepwise feedback feature affects both the `QuotesPortalService`, to retrieve customer account information, and the `QuotesOrderService`, to perform the prioritizing. A more trivial case is the secure communication feature: encryption and decryption operations should be performed at both sides of the connection. This clearly illustrates that a single feature, often consisting of a client and server functionality part, can affect multiple services in a service composition.

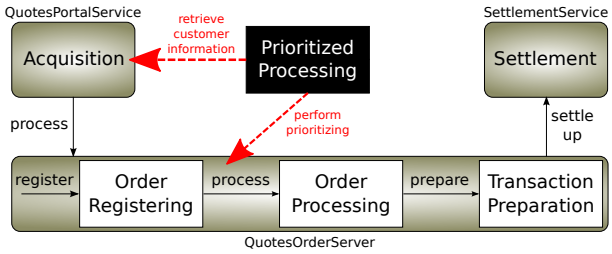

Figure 2: Services affected by the stepwise feedback feature.

However, each company in a cross-organizational service network has its own IT administration and trust domain, and will not allow external parties to add or update feature implementations. The services provided by the different partners are black boxes, loosely-coupled and independently maintained by the company's own administrators. This black-box scenario hinders the feature modularization and composition in a cross-organizational context [1]. Therefore a feature cannot be condensed into a single module any more. Cross-organizational features need to be split up in client-side and server-side aspects, independently implemented with possibly different AO-technologies. However, a uniform high-level representation of those features is crucial to be able to share them in a particular application domain or service network.

3. APPROACH

In this section we present our approach to achieve a multi-layered architecture for the uniform representation of cross-organizational features in AOSD. A multi-layered architecture, distinguishing between policy and mechanism, is a core tenet of the body of research on cross-organizational coordination architectures. We shortly review the state-of-the-art in this field. Subsequently, based on this tenet, we propose that aspect-based variability is first described at the level of a technology-independent feature ontology that is well-defined for a domain or a specific service network. Each organization implements this feature ontology independently using an AOP technology of its choice. The mapping between the independent feature ontology and the aspect-based implementation is then specified as part of the second layer of the model. Finally, we show how this feature

ontology is used for cross-organizational service customization.

3.1 Cross-organizational Coordination Architectures

Our approach is inspired by the design principles of cross-organizational design. In the field of cross-organizational coordination architectures, a layered system architecture is a core principle of the reference model [31]. This reference model distinguishes between (i) the type of agreements that are established, (ii) the language for describing the agreements, and (iii) the middleware for establishing and executing the agreements.

The language for describing how the interactions between two or more independent services are to be done is further refined into a conceptual and a computational model [31]:

1. A *conceptual model* provides the modeling concepts to describe the regulations at a sufficient high-level of abstraction that is independent from the organizations internal processes and data.
2. A *computational model* offers behavioral concepts that are mappable to implementable actions in the underlying software system that can be enforced upon contracted services.

The conceptual model of the language should be as independent as possible from the computational model to enable that different organizations can implement the same agreement differently depending on their choice of implementation platform, while adhering to the terms of the agreement.

When multiple independent organizations interact with each other, they have to integrate their business processes in order to be able to operate, gain added-value and survive in a market. To enable this, a certain agreement must be complied by all participating organizations in the business relationship. We think that a common feature ontology can be part of this agreement. Therefore, it is plausible to assume that services in a service network can share a common feature ontology.

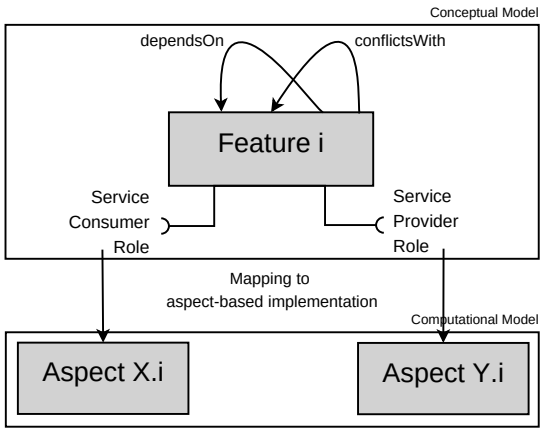

Figure 3: Aspect-based variability model.

3.2 High-level Feature Ontology

The conceptual model in our approach for specifying cross-organizational features consists of a feature ontology. Similarly to the conceptual model of the cross-organizational coordination architectures, this feature ontology should be high-level and independent from the aspect-based implementation to enable organizations in service networks to implement cross-organizational features using an AOP technology of their choice (see Fig. 3). In order to be successful, the feature ontology must have a clear scope on which particular application domain or area it applies, for example, a specific market such as financial services or an individual (long-running) business relationship between multiple organizations.

The specification of such a common feature ontology is divided into a base level and one or more application-specific levels. The base ontology is a framework and vocabulary for specifying application-specific ontologies. An application-specific ontology contains a catalog of features that can be used within a certain cross-organizational service network. Application-specific ontologies are hierarchically structured: the application-specific ontology of a specific service composition imports and extends the ontology of the application domain.

A feature ontology can be seen as a *high-level, technology-independent agreement* between the parties involved (typically a service consumer and service provider). This agreement prescribes the intended behavior of the feature and clearly sets out the roles that different parties involved have to play, as depicted in Fig. 3. These roles are described by a name (e.g. Service Consumer) and a set of responsibilities. These responsibilities specify constraints on behavior (the specification of an algorithm to be used) and interfaces (message types and operations that are required or provided). Further, composition rules can be specified that prescribe which features depend on other features and which features can't be executed during the same request due to feature interference.

Listing 1: Example of high-level features.

```
feature PrioritizedProcessing {
 dependsOn: Billing;
 role ServiceConsumer {
  responsibility retrieveCustomerAccount {
        provides: CustomerAccount;
  }
 }
 role ServiceProvider {
  responsibility performPrioritizing {
        requires: CustomerAccount;
        provides: AccountableItem;
  }
 }
}
```

For example, the `PrioritizedProcessing` feature from Fig. 2 needs two roles: a service consumer who retrieves customer account information, and a service provider, responsible for performing the prioritizing. The service provider role requires a `CustomerAccount` attribute, which will be provided by the service consumer role. After the prioritizing, the ser-

vice provider role will provide a `AccountableItem` attribute that will be used by the `Billing` feature. The feature description is presented in Listing 1. It also defines a composition rule that prescribes that `PrioritizedProcessing` requires the `Billing` feature.

3.3 Mapping to Aspect-based Implementation

The mapping between the high-level feature ontology and the aspect-based implementations is specified on the level of the service platform, hiding the implementation details for external parties. The use of AOSD [9] enables a clean separation of concerns, in which the core functionality of a service is separated from any feature behavior. Therefore they are implemented separately from each other as composite entities containing a set of aspect-components, providing the behavior of the features (so called advice). This advising behavior can be dynamically composed on all the components of a service – at client-side and at server-side.

By capturing the semantics of the features in a high-level feature ontology, the different features can be implemented independently by each of the service providers using their favorite service platform and AO-composition technology. Hence, the different services in the network may have their own optimized aspect-based implementations of the different features, and the most appropriate feature implementation in each service may depend on environmental circumstances. This decentralized feature management allows a variety of service platforms using different AO-composition technologies to be interconnected. In addition, the implementation of the different features and the software composition strategy are open for adaptation by each of the local administrators. However, the feature implementations have to satisfy certain constraints, enforced by the feature ontology.

Each *feature implementation mapping* within a specific organization is described by means of a declarative specification that specifies: (i) the feature and role that is implemented, (ii) the aspect-component that implements the particular role, and (iii) optionally an AO-composition for weaving the aspect-component into the internal processes and data of the organization (see Listing 2).

Listing 2: Example of a feature implementation mapping.

```
featureImplementationMapping PPImpl {
 implements: PrioritizedProcessing;
 role: ServiceConsumer;
 ao-component: PPAOComponent;
 ao-composition {
   ...
 }
}
```

3.4 Using the Feature Ontology for Cross-Organizational Service Customization

The Web Services Description Language (WSDL) is an XML-based language that provides a model for describing web services. The web service is defined by an interface, describing the operations that can be performed and the message types that are required/provided by these operations. The

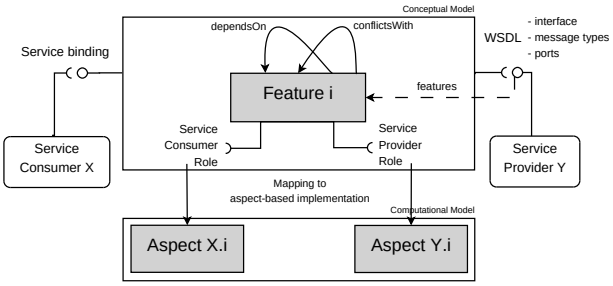

Figure 4: Using the Variability Model for Web Services.

WSDL also defines services as collections of network endpoints, or ports. A port is nothing more than the address or connection point to the web service (typically a http URL). To be able to use our feature ontology, the WSDL should be extended with the set of available features (see Fig. 4).

The variability model is accessible to the clients of the service application and allows them to select a desired set of features. Configuration of features across the service network happens through instantiation of *service bindings*. A service binding is a declarative specification, specifying the web service location, the selected port and which features are desired (see Listing 3).

Listing 3: Example of a service binding.

```
servicebinding {
 URI: http://www.stocktradingexample.be;
 port: StockTradingServiceSoapEndpoint;
 features: PrioritizedProcessing, Billing;
}
```

4. RELATED WORK

We first discuss the work in the context of cross-organizational service provisioning. Next we discuss the related research in the domain of service composition.

Cross-organizational coordination architectures. A multi-layered architecture, distinguishing between policy and mechanism, is a core principle of the body of research on cross-organizational coordination architectures. Firstly, agreements must be represented digitally by means of a language that offers the necessary concepts for describing and enforcing agreements. Second, coordination middleware must be developed in order to establish agreements dynamically, and to enforce the agreements or detect violations against it.

The current state-of-the-art on cross-organizational coordination architectures in the general area of SOA consists of policy-based and contract-based frameworks. Contract frameworks (such as BCL [21], [11], [26], GlueQoS [32], T-BPEL [29] and SLAng [28, 27]) focus mostly on negotiation, enactment and monitoring, while policy-based architectures (e.g. Ponder [7] and LGI [22]) focus exclusively on enforcement. These coordination architectures establish agreements dynamically between two or more organizations, but fail to support the coordination of system-wide customizations of service compositions. Our approach deals with this by providing an aspect-based variability model for

cross-organizational features, managed in a multi-layered architecture.

Service composition. Previous research focussed already on automated composition of web services into composite web services [10, 6, 13]. For this purpose, matchmaking algorithms search for matching web services based on their input/output, the interaction protocol and functional behavior, using a forward or backward chaining algorithm and a discovery service. The matchmaking process can be either centralized (i.e. planning a complete composition at once), or decentralized, allowing each web service in the composition to decide individually which web services to select in providing the required services for processing the request. This functional matchmaking process is originally based upon WSDL information in the UDDI directory to select the appropriate services. In more recent work, the matchmaking process is based upon QoS properties of the different web services [32, 33, 34, 16, 4, 8]. Here, non-functional properties such as security, reliability and performance are used by the matchmaking algorithm to select the most appropriate service. For example, in [33], Zheng et al. propose a quality-driven approach to select component services during the execution of a composite service. For this purpose, they define a web service quality model based upon five non-functional properties and a global quality-driven selection algorithm formulating these properties as a linear optimization problem. In this approach, every service is assumed to have one particular QoS profile, described in the quality model. [18] presents an heuristic algorithm for composing services to achieve global QoS requirements in dynamic service environments.

A common denominator in this research domain is the usage of ontologies [3] to store semantic information about web services to automate the matchmaking of services in a web service composition based upon functional and non-functional properties [25, 16, 30]. In our approach, we use ontologies and semantic information to describe features as first class entities rather than describing web services with their properties. In this way, the information about the features is web service independent. Thanks to this ontology, automated reasoning can be done about the customization of the orchestration on a per-request basis, without considering the actual web service composition.

The GlueQoS middleware-based approach of Wohlstadter et al. [32] manages dynamically changing QoS requirements of web services by delaying QoS commitments of the services. Each service describes its QoS preferences, and a middleware-based resolution mechanism searches for a satisfiable set of QoS features to inter-operate for services that encounter each other for the first time. Similar to our approach, GlueQoS uses a fixed ontology for classifying features and describing their interactions and possible interference. However, their selection of features is fully decentralized and on a per-collaboration basis (optimally suited for a highly dynamic web service composition), but lacking support for client-specific customization and consistent processing throughout cross-organizational service compositions, as our approach does.

Finally, our approach does not pretend to replace existing WS-standards such as WS-Coordination [24], WS-Policy [2] and WS-Security [23], but we intend to offer a complementary approach for consistent customization of features in orchestrations. For example, in our approach we use a per-request tagging solution to achieve coordination between the client and the different web services. In case more complex coordination schemes are needed (e.g. if coordination messages don't follow the message flow), our approach can be combined with WS-Coordination. This coordination specification was originally defined for coordinating transaction protocols, but is extensible for all kinds of coordination protocols in a web service environment.

5. CONCLUSION AND FUTURE WORK

In this paper, we proposed an aspect-based variability model for representing cross-organizational features in service networks. Our approach consists of a multi-layered architecture, mapping a technology-independent feature ontology onto an aspect-based implementation.

The approach supports maintaining the compatibility of feature implementations across a service network of independent organizations. The common feature ontology can also be leveraged to support client-specific customization of cross-organizational features across such service networks. As only limited tests have been performed, further validation and evaluation of our approach are necessary.

6. REFERENCES

[1] Apel, S., Kaestner, C., and Lengauer, C. Research challenges in the tension between features and services. In *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments* (New York, NY, USA, 2008), ACM, pp. 53–58.

[2] BEA Systems, IBM, Microsoft Corporation, SAP AG, Sonic Software, and VeriSign. Web Services Policy Framework (WS-Policy). `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-polfram/ws-policy-2006-03-01.pdf`, March 2006.

[3] Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web. *Scientific American 284*, 5 (2001), 34–43.

[4] Bilgin, A. S. A DAML-based repository for qos-aware semantic web service selection. In *IEEE International Conference on Web Services (ICWS 2004)* (2004), IEEE Computer Society, pp. 368–375.

[5] Cohen, S., and Krut, R., Eds. *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines* (May 2008), Carnegie Mellon University - Software Engineering Institute.

[6] Constantinescu, I., Faltings, B., and Binder, W. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS 2004)* (2004), pp. 506–513.

[7] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. The ponder policy specification language. In *Policies for Distributed Systems and Networks* (2001), Springer, pp. 18–38.

[8] Felfernig, A., Friedrich, G., Jannach, D., and Zanker, M. Semantic configuration web services in

the cawicoms project. In *ISWC '02: First International Semantic Web Conference on The Semantic Web* (2002), Springer, pp. 192–205.

[9] Filman, R. E., Elrad, T., Clarke, S., and Akşit, M. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2004.

[10] Foster, H., Uchitel, S., Magee, J., and Kramer, J. Compatibility Verification for Web Service Choreography. In *IEEE International Conference on Web Services (ICWS 2004)* (2004), IEEE, pp. 738–741.

[11] Hoffner, Y., Field, S., Grefen, P., and Ludwig, H. Contract-driven creation and operation of virtual enterprises. *Computer Networks 37*, 2 (2001), 111–136. Electronic Business Systems.

[12] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. 21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[13] Lassila, O., and Dixit, S. Interleaving discovery and composition for simple workflows. In *Semantic Web Services, 2004 AAAI Spring Symposium Series* (2004).

[14] Lee, J., Muthig, D., and Naab, M. An Approach for Developing Service Oriented Product Lines. In *SPLC '08: 12th International Software Product Line Conference* (Sept. 2008), pp. 275–284.

[15] Lee, K., Kang, K. C., Kim, M., and Park, S. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *Software Product Line Conference, 2006 10th International* (0-0 2006), pp. 10–112.

[16] Lee, Y., Patel, C., Chun, S. A., and Geller, J. Towards intelligent Web services for automating medical service composition. In *IEEE International Conference on Web Services (ICWS 2004)* (2004), pp. 384–394.

[17] Loughran, N., and Rashid, A. Framed Aspects: Supporting Variability and Configurability for AOP. In *Software Reuse: Methods, Techniques and Tools* (2004), Springer, pp. 127–140.

[18] Mabrouk, N. B., Beauche, S., Kuznetsova, E., Georgantas, N., and Issarny, V. QoS-aware service composition in dynamic service oriented environments. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (New York, NY, USA, 2009), Springer-Verlag New York, Inc., pp. 123–142.

[19] Medeiros, F. M., de Almeida, E. S., and de Lemos Meira, S. R. Towards an Approach for Service-Oriented Product Line Architectures. In *Proceedings of the Third Workshop on Service-Oriented Architectures and Software Product Lines (SOAPL)* (2009), S. Cohen and R. Krut, Eds., pp. 151–164.

[20] Mezini, M., and Ostermann, K. Variability management with feature-oriented programming and aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* (New York, NY, USA, 2004), ACM, pp. 127–136.

[21] Milosevic, Z., Linington, P. F., Gibson, S., Kulkarni, S., and Cole, J. Inter-Organisational Collaborations Supported by E-Contracts. In *Building the E-Service Society* (2004), Springer, pp. 413–429.

[22] Minsky, N. H., and Ungureanu, V. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol. 9*, 3 (2000), 273–305.

[23] OASIS Web Services Security (WSS) TC. Web Services Security (WS-Security). `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`, February 2006.

[24] OASIS Web Services Transaction (WS-TX) TC. Web Services Coordination (WS-Coordination). `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os.pdf`, February 2009.

[25] Paolucci, M., Kawamura, T., Payne, T. R., and Sycara, K. Semantic matching of web services capabilities. In *ISWC '02: First International Semantic Web Conference on The Semantic Web* (2002), Springer, pp. 333–347.

[26] Shrivastava, S. Tapas final report. Tech. rep., Technical Report Project deliverable D20, 2005.

[27] Skene, J., and Emmerich, W. Engineering Runtime Requirements-Monitoring Systems Using MDA Technologies. In *Trustworthy Global Computing (TGC)* (2005), vol. 3705, Springer, pp. 319–333.

[28] Skene, J., Lamanna, D. D., and Emmerich, W. Precise Service Level Agreements. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 179–188.

[29] Tai, S., Mikalsen, T., Wohlstadter, E., Desai, N., and Rouvellou, I. Transaction policies for service-oriented computing. *Data & Knowledge Engineering 51*, 1 (2004), 59–79.

[30] Trastour, D., Bartolini, C., and Gonzalez-castillo, J. A Semantic Web Approach to Service Description for Matchmaking of Services. In *In Proceedings of the International Semantic Web Working Symposium (SWWS)* (2001).

[31] Truyen, E., and Joosen, W. A reference model for cross-organizational coordination architectures. In *International Conference on Enterprise Distributed Object Computing Workshops* (2008), IEEE, pp. 252–263.

[32] Wohlstadter, E., Tai, S., Mikalsen, T., Rouvellou, I., and Devanbu, P. GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 189–199.

[33] Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., and Sheng, Q. Z. Quality driven web services composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web* (New York, NY, USA, 2003), ACM, pp. 411–421.

[34] Zeng, L., Benatallah, B., Ngu, A. H. H., Dumas, M., Kalagnanam, J., and Chang, H. QoS-aware middleware for Web services composition. *IEEE Transactions on Software Engineering 30* (2004), 311–327.