A Model for Composable Composition Operators

Expressing object and aspect compositions with first-class operators

Wilke Havinga Lodewijk Bergmans Mehmet Aksit

Software Engineering group – University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{w.havinga,l.m.j.bergmans,m.aksit}@ewi.utwente.nl

Abstract

A considerable amount of research, especially within the OO and AOSD communities, has focused on understanding the potential and limitations of various composition techniques. This has led to a large amount of proposals for alternative composition techniques, including many variations of message dispatch, inheritance, and aspect mechanisms. This paper makes the case that there is no single perfect composition technique that suits every situation, since different techniques incur different trade-offs. The proper composition technique to use depends on the particular design problem and its requirements (e.g. w.r.t. adaptability, reusability, understandability, robustness, etc. of the various elements of the design). However, most programming languages limit the available composition techniques to a very few. To address this, we propose a novel composition model. The model provides *dedicated abstractions* that can be used to express a *wide variation of object composition techniques* ("composition operators"). Examples include various forms of inheritance, delegation, and aspects. The proposed model *unifies* objects (with encapsulated state and a message interface) and composition operators; composition operators are specified as first-class citizens. Multiple composition operators can be combined within the same application, and composition operators can even be used to compose new composition operators from existing ones. This opens new possibilities for developing domain-specific composition operators, taxonomies of composition operators, and for reuse and refinement of composition operators. To validate and experiment with the proposed model, we have designed and implemented a simple language, that we also use in this paper to show concrete examples.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Modules and Interfaces, Object-oriented design methods; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Object-oriented programming; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages, Extensible languages

AOSD 2010 March 15–19, 2010, Rennes, France

1. Introduction

The history of programming languages shows a continuous search for new—presumably better—composition techniques. The typical aim of such techniques is to find better ways for structuring increasingly complex software systems into modules that can be developed and reused independently.

Composition operators are language mechanisms that let programmers compose behavior and/or data, defined as separate entities, by means of a *composition specification*. An example of a composition operator is *function application* (viz., calling a function or method). This operator allows the invocation of functionality that is defined separately (as a function definition), by means of a call statement (fulfilling the role of *composition specification*). Other examples of composition operators include inheritance (in many different styles), delegation, pointcut-advice mechanisms, composition filters, mixins, traits, etc.

Most languages adopt a fixed set of composition operators, typically with explicit notations and predefined semantics. In case a language does not provide a composition operator with the desired compositional behavior, programmers may need to write workarounds in their applications; by adding glue code, or by using macros, libraries, frameworks or language extensions. However, typically, such workarounds are not integrated with the language, and the resulting abstractions suffer from lack of comprehensibility, adaptability and reusability.

The availability of only a limited set of composition operators causes additional issues; most existing languages have a bias towards one kind of decomposition of software systems[1], which also imposes constraints on the viability of particular evolution scenarios, or in other words, the *extensibility* of software [13]. Thus, each composition operator (and hence, language) has a bias that makes some types of evolution scenarios easier to accommodate, or less error-prone, than others. Such trade-offs are inherent to the choice of particular composition operators – there exists no single composition operator that is able to address all kinds of evolution scenarios equally well, while still providing meaningful higher-level abstractions.

To work towards addressing the issues identified above, we present a composition infrastructure that (a) supports the definition of a range of composition mechanisms, (b) allows composition mechanisms to be expressed in terms of first-class entities, enabling the construction of new composition mechanisms from existing ones, (c) supports the use of multiple composition mechanisms within the same program, while (d) supporting a variety of aspect- as well as object-based composition mechanisms.

Our approach has been implemented and is presented in this paper in terms of a small language, called "*Co-op*". In this language,

[1] This is similar to the "tyranny of the dominant decomposition" [39].

composition operators can be constructed using several "primitive" elements, such as selectors, bindings, actions and constraints, which can be used to define composition operators based on implicit invocation. These primitive elements are expressed in terms of first-class elements (objects), so that they can be freely composed. We use these primitive elements to express several composition mechanisms, including different styles of inheritance, e.g. as found in Smalltalk [17] or Beta [28], as well as aspects.

This paper is structured as follows: the next section discusses the issue of trade-offs between composition operators in more detail. Section 3 discusses our approach, presenting our composition model. In section 4, we show example applications of our model, defining several composition operators and applying those to an example case. The next section discusses how our composition operators can be used to create new composition operators by composing existing ones. Section 6 discusses several important design decisions, and is followed by a discussion of related work, in section 7, and an evaluation and conclusion, in section 8.

2. Motivation

2.1 Background

In this paper, we argue that there is no single "best" composition technique; instead, different composition techniques may be the most suitable in different application contexts. Each composition technique offers a particular trade-off between various characteristics, such as flexibility, ease of understanding, ability to share behavior or state, robustness against programmer mistakes, and-soforth[2]. In addition, different application contexts require different characteristics from the composition technique to be employed. As a result, software engineers require multiple composition techniques in their toolbox.

This is by itself not a new observation, and several approaches have been proposed that aim to address this issue at least partially (Section 7 discusses related work in detail):

- *offer a variety of languages*: for example one of the philosophies of the .NET platform is to offer software engineers a choice of programming languages within the same platform, including interoperability of objects among languages. The latter is achieved by committing to a standard object model including (fixed) inheritance semantics.
- *domain-specific languages*: is a variation to the previous item, but emphasizes that different application domains require tailored abstractions and ways to express these abstractions. Accordingly, domain-specific abstractions may also require domain-specific composition techniques.
- *meta-object protocols*: A metaobject protocol (MOP) is an interpreter of the semantics of a program that is open and extensible [40]. A specific MOP implementation offers a framework that fixes certain core parts of the language, and allows for extension and refinement of other parts, essentially by refining the implementation of the interpreter. This allows a range of -related- interpreters to be constructed using the MOP. The ease and modularity of doing these extensions depends completely on the design of the MOP.

In this section, we will introduce a simple example that illustrates why multiple, different, composition techniques may be needed within a single application. Figure 1 provides an overview of the example, which simulates a simple office or workflow environment. It contains an is-a hierarchy of several person objects; at the most general level an object type `Person` is defined, which offers basic functionality generic to persons. A more specific object type `Employee` defines that all employees have a (unique) ID, which must conform to certain rules checked by method `validID()`. Employees also have a method `performTask()` that enacts specific tasks.

Types `Secretary` and `Staff` are both special cases of `Employee`, where `Secretary` manages an encapsulated agenda, offering several methods for scheduling appointments. Each secretary object has its own instance of the agenda. `Staff` objects have a `jobDescription`, which is a list of tasks they are allowed to perform (by the `performTask()` method). All these objects also provide an `asString()` method that returns a string representation of the object. Finally, `LogOfficeTasks` monitors all tasks that are performed by all employees in the system, this can be used to check progress, to enforce certain workflows, or to implement billing.

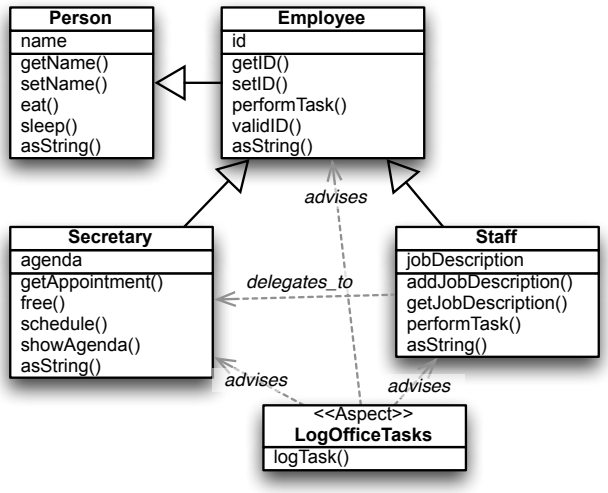

Figure 1. Example application that illustrates various composition techniques.

In the context of this simple application, we will discuss a number of alternative compositions to apply, some of the trade-offs that are involved:

2.2 Inheritance

The example application involves a number of *is-a* relations; `Employee` *is-a* `Person`, `Secretary` *is-a* `Employee` and `Staff` *is-a* `Employee`. These *is-a* relations can be represented by the well-known object-oriented *inheritance* composition operator. There are many different proposals for inheritance semantics (see e.g. the overview in [37]); here we will discuss two alternatives; *Smalltalk-style* inheritance and *Beta-style* inheritance.

In languages that support an inheritance mechanism similar to Smalltalk [17] or Java, subclasses can override methods defined in their superclass, and decide whether the original behavior (in the superclass) is invoked, through a message call to "super". The advantage of this type of inheritance is that it supports unanticipated extensions; a class need not be prepared for possible extensions in the future. A disadvantage is that it is impossible to restrict subclasses from completely redefining existing behavior as implemented in the superclass. This makes it very easy to define subclasses that (accidentally) break properties that where (previously) guaranteed by their superclass.

In the example, `Employee` defines method `validID()`: this method determines whether an ID is valid, according to certain

[2] We consider "best of breed" composition techniques only, and further ignore that perhaps for particular techniques, one composition technique only present improvements over another.

wellformedness rules. A subclass, such as `Staff`, can override this method and thereby refine the rules, e.g. because the ID of all `Staff` persons must have a certain format. However, this may also, inadvertently, break the rules defined in the superclass; and there is nothing that the original implementation of the method can do to prevent this[3].

In the *Beta* language [28], superclasses have control over the execution of methods that are overridden in subclasses. This is achieved by inserting calls to *inner* in those locations where a method may be extended by additional behavior, as (optionally) supplied by a subclass. For example, the implementation of the `isValid()` method in `Employee` could make an `isValid()` call to `inner`, with the certainty that its local checks are still always executed. The advantage is that a superclass can more easily guarantee certain properties (e.g., invariants), as it has control over the invocation of any sub-behavior. On the other hand, this approach makes it impossible to completely replace ("override") existing behavior by means of subclassing, thus severely limiting the potential directions in which a class can evolve. In addition, the desired extension points must be predicted correctly.

This shows that both Smalltalk-style and Beta-style inheritance may be the best choice in certain situations, but neither is the best in all possible situations; the preferred solution is that a designer can deliberately choose which style of inheritance to use in which part of a design.

2.3 Delegation

In our example, staff members have a shared agenda, managed by an instance of `Secretary`; so staff instances share/reuse the state (instance variables) of `Secretary`, but also the behavior (methods) for accessing the agenda; methods `getAppointment()`, `free()`, `schedule()`, and `showAgenda()` must be available on the interface of `Staff` as well. In other words, instances of `Staff` "delegate" part of the functionality to a `Secretary` instance.

We will consider three alternative composition techniques to compose the behavior of `Secretary` and `Staff`: inheritance, message passing and delegation. First, implementing this behavior using inheritance (where e.g. `Staff` would inherit from `Secretary`) is inappropriate, since this would cause each instance of `Staff` to have a copy of the shared agenda. Implementing this composition by a message passing relationship between `Staff` instances and a `Secretary` instance (to which they must keep a reference in that case) suffers from the "self problem" [27]: For example the `schedule()` method of `Secretary` needs to add information about the employee with whom the appointment is made, or rather, on whose behalf the method is executed. In the case of a message send, the "self" or "this" object context changes to the object to which the call is routed, that is, the `Secretary` instance; it is impossible in this case to refer to the original receiver of the message. Although it is possible to work around this by passing the original interface object as an extra call parameter, this is an unsatisfactory solution. For example, this workaround necessitates changes to the interface of affected methods, which may be undesirable, for instance, if this changes the interface of a class that is part of an existing library, or if this change of interface has a cascading effect within the inheritance tree.

In contrast, in languages that support explicit delegation, the "self" context does not change when delegating an operation, and still refers to the original receiver of the request; in this case an instance of `Staff`.

Again we see that composition through inheritance is better in some situations (e.g. when reusing and refining behavior, but not state), and composition through delegation in other situations (when reusing both behavior and state).

[3] Although e.g. Java has an additional keyword `final` that completely forbids overriding of a specific method.

2.4 Aspects

As a third example of the trade-offs between composition operators, we take a look at the monitoring behavior defined for our example. Monitoring is a *crosscutting* behavior; it is a single concern that affects a number of places in the code; in this case all the locations where employees perform tasks. This can be implemented by inserting monitoring code in all relevant locations, perhaps just consisting of a single library call, including some code that passes the relevant context. Aspect-oriented composition is an alternative implementation technique, which has the advantage that it modularizes crosscutting concerns, in this case the monitoring code, which makes it much easier to maintain the monitoring behavior, understand which locations in the system are monitored, and add or remove new monitoring locations. The latter may even occur implicitly when the base code evolves.

In our example, monitoring of tasks can easily be captured by an aspect, that acts upon the execution of the `performTask()` method, and can observe the context of the join point, including the task that is passed as an argument. As a result, monitoring will be implemented in a separate module, where all changes to the monitoring process can be localized.

Possible disadvantages of using aspects are that they are applied implicitly; when observing the source code of a single module, it may not be obvious which aspects are applied, in which order, and how they interact. Also, if the crosscutting behavior is different at each join point (for example if the message to be logged at each join point is really semantically different, and these differences cannot be factored out in terms of the context), then it may not make much sense to modularize that into a single aspect.

2.5 Contributions of this paper

Clearly, the examples we presented in this section are just a small subset of all possible design considerations, trade-offs and alternative composition that can be made in the design of software systems. The key message we want to convey is that *by committing to a single, or a fixed set of, composition technique(s), it will be impossible to fulfill all (quality) requirements* in many systems. The costs of such deficiencies may vary from close to nothing in simple, non-critical applications, to substantial for critical applications where for example modularization, robustness or adaptability are important requirements.

This paper makes the following contributions, therewith addressing this problem:

1. It presents a novel model that supports composition operators as user-defined, modular and reusable first-class abstractions. We are not aware of any language or MOP that has a similar model (design) and characteristics;
2. In particular, our model has strong support for composition of composition operators; either the combination of multiple, different operators in a single application, or the ability to construct new composition operators from existing ones, through the application of other composition operators.
3. Our proposed model unifies and supports both object-oriented and aspect-oriented composition techniques.
4. It illustrates the feasibility of the model by presenting a simple, experimental, object-based language as an instantiation.

3. Composition Model

In this section, we propose our core composition model. This model can be seen as a generalization of prior work, related to the mod-

eling and composition of (domain-specific) aspect languages [22]. In this paper, we employ a very similar set of concepts to model diverse composition operators. These composition operators can then be applied to any kind of object-based model. In this paper we will demonstrate this through an object-based programming language called *Co-op*, which we designed for the purpose of experimenting with a language that does not supply built-in composition operators of its own. We will (briefly) explain and use *Co-op* in the following section to illustrate the definition and usage of composition operators in detail[4]. We consider the general model of composition operators we propose and its capabilities as a main contribution of this paper, rather than the *Co-op* language itself, which serves mainly as a vehicle for experimentation and illustration of the composition operators model.

The core composition model we propose consists of the following elements:

- *Events*, which may be published (generated) during the execution of a program,
- *Event Selectors*, queries that can be matched against published events based on properties of the event, as well as other (reflective) information about the program that can be reached through the context of an event,
- *Action Selectors*, which select an operation to be invoked, as well as the intended target object,
- *Bindings*, which bind event selectors to action selectors, and in addition specify the binding of values between the "incoming" event and the invoked behavior, or the "outgoing" event (in effect, this achieves sharing of values between contexts),
- *Constraints*, which can be used to restrict or determine the ordering of the execution in the case that multiple selectors (and hence, bindings) match the same event.

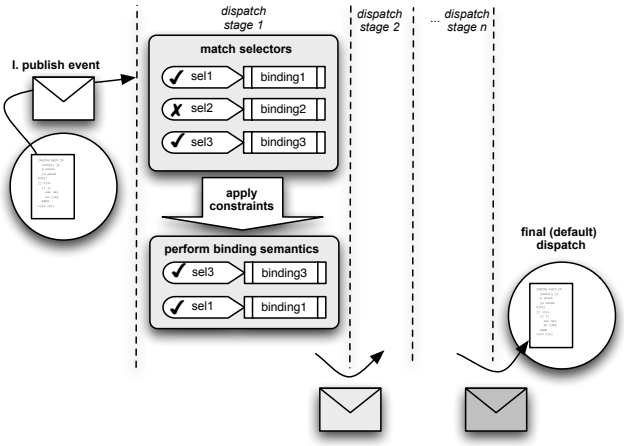

Figure 2. An overview of the event dispatching process

These concepts can be used together to define how events are eventually bound to concrete operations. Figure 2 schematically illustrates this. We briefly list the flow of events in a number of steps:

- On the left hand side we see that execution of operations may lead to publishing of an event.
- It is determined which of all (active) event selectors match with the event. This potentially enables a set of bindings that refer to those selectors.
- Applying the constraints that have been defined between the various bindings may further reduce the applicable bindings (e.g. because bindings may exclude each other), and determines a (partial) ordering among the bindings.
- The resulting set of bindings is evaluated; this consists of binding of values from the received event to the resulting event, and the evaluation of the associated action selectors.
- This process may repeat itself multiple times (i.e. there can be multiple dispatch stages), as long as there are matching bindings.
- Actual execution of (base-level) operations will occur when a "default binding" is executed; this is discussed below.

We have defined a denotational semantics of this model as applied to the *Co-op* language, including a precise definition of the complete selector evaluation process that we informally described above. However, there is insufficient space to include it in this paper. For those interested, we refer to [21, Appendix A].

We apply this model in the context of a simple object-based programming language, which defines *modules* that may encapsulate *instance variables* and *operations*. Modules are essentially classes, but have no built-in composition mechanisms such as inheritance. *Operations* may declare parameters and local variables. Their implementation specification consists of a list of statements, of which 3 kinds exist: *assignments*, *event generations*, and *returns*. Apart from built-in modules such as `Boolean`, `String`, `List`, and `Dictionary`, the language also supports closures, so that common control flow mechanisms can be implemented without extending the syntax of the language (cf. Smalltalk).

Listing 1 shows the definition of module `Person`, as discussed in section 2. We expect that, for anyone familiar with Smalltalk or Java, the syntax of this code will be understandable. For example, the statement on line 13 generates an event (or message) that specifies variable `p` as its intended target, `setName` as a selector, and a string-literal parameter "John Smith". An important difference, as compared to Smalltalk, is that in the absence of composition operators, such *event generations* do not have any observable effect, as of yet. The effect of the expression `Person new`, on line 12, is to create a new instance of module `Person`, and to invoke the operation `init` of that module.

```
1  module Person{
2    var name;
3
4    init { name = ""; }
5    getName { return name; }
6    setName:newName { name = newName; }
7    asString { return "Person with name: " cat: name; }
8  }
9
10 module Main {
11   main {
12     var p; p = Person new;
13     p setName: "John Smith";
14     Console writeln: (p asString);
15   }
16 }
```

Listing 1. module `Person` and usage example in *Co-op*

Specifically, an event has the following properties, which may be used by event selectors as matching criteria:

sender: the object context from which the event originates.

target: the intended receiver object of the event, which is explicitly specified (but can be modified by composition operators).

[4] We believe that showing examples of this model as a pure MOP would require showing a lot of MOP code, that would be harder to understand.

selector: a selector indicates the name of the operation intended to be invoked (which can also be modified by composition operators).

local type: the type (module) from which the event originates; i.e. the module that defines the operation that generated the event.

lookup type: this indicates the module in which to look for the operation implementation (defaulting to the type of the target object, but this too can be changed by composition operators).

call annotations: these annotate calls with additional semantics, which can be interpreted by composition operators, and may be used to indicate that an event should be interpreted in an irregular way. For example, in our model, composition operator features such as "super" or "inner" calls (in respectively Smalltalk-style and Beta-style inheritance) are essentially modeled as "this"-calls with an annotation that instructs the inheritance operator to perform operation lookups differently.

To make the language of practical use, we define a "default" composition operator in terms of the composition model concepts discussed above. This composition operator defines an *event selector* that matches *any* event within the program, regardless of its properties. Its *action selector* selects the operation (if it exists) with the message selector specified by the event (e.g. `setName`), in the module corresponding to the *lookupType* of the intended target object. In the event specified on line 13 (Listing 1), variable `p` is the intended target, which has the type `Person`. The *event selector* is bound to the *action selector* by means of a binding that in addition defines the pseudo-variable `this` as equal to the target object of the event. This pseudo-variable is then available within the called context, and can be accessed as a normal variable.

It is important to note that constraints can be applied to the default operator, just like they can be applied to any other operators. The default composition operator is therefore not a "fixed" mechanism that cannot be overridden.

In the case that no other operator has modified the event properties or imposed constraints, the default operator thus has the—hopefully unsurprising—effect of dispatching the "call" to the indicated target object, invoking the operation indicated by the message selector.

In the following sections, we define several additional composition operators, and show how these can be applied to the example introduced in section 2. *Co-op* represents the concepts used to construct composition operators as first-class objects within the program, i.e., they can be used as parameters, returned as values of operations, assigned to (instance) variables etc., like any other object. This also means that composition operators can be composed with other composition operators, and thus reused and refined. Thus, composition operators are not built into the language, but can be expressed and composed using primitives supported by the language. Since there is insufficient space in this paper to explain the *Co-op* language in detail, we refer to [21] for an extended discussion of the language.

Figure 3 shows the composition infrastructure schematically. In the middle left, an object (a module instance) publishes an event. This event is evaluated by the active set of selector bindings. These bindings can be defined using *Co-op* modules. Finally, the evaluation of the event typically leads to the invocation of one (or more) operations, on an instance of the target object. How the evaluation works is discussed in more detail in the next subsection.

4. Definition and usage of composition operators

In this section, we illustrate the definition of new composition operators and the combination of these within a single application by following the example from section 2.

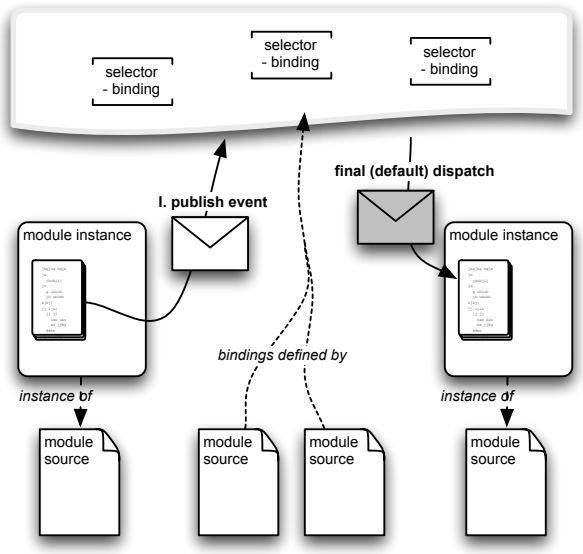

Figure 3. Overview: Composition in *Co-op*

4.1 Inheritance

As described in section 2, it often makes sense to model "is-a" relationships using inheritance. In this section, we explain the implementation of a single-inheritance mechanism. We have additionally implemented several other variants of inheritance (documented in [21], and available on the website [2]), including multiple inheritance and Beta-style inheritance [28]. There, also support for "super" calls is included, which we do not show here.

We define the intended behavior of inheritance as follows: given an inheritance relation where module `X` extends module `Y`, events of which the lookup type is `X` should in principle lead to execution of the indicated operation within module `X`. (This corresponds to the behavior implemented by the default call binding.) However, if (and only if) module `X` does not define an operation with the message selector as indicated by the event, the event is instead re-evaluated while its *lookup type* is changed to `Y`, thus "forwarding" the message to the "parent" type. Listing 2 shows the implementation of such an inheritance mechanism using *Co-op*.

To implement the above in terms of the concepts used to construct composition operators, listing 2 defines an *(event) selector*, `callsToChild`, that matches when the *lookup type* of an event matches the `child` type of an inheritance relation (lines 5–6). Note that the *(event) selector* is defined as a normal object; it is initialized with a closure (the part between brackets, line 6), which is parameterized by an *event* (the closure parameter is notated as `|evt|`, and follows the opening bracket of the closure definition). Once registered with the *Co-op* interpreter, the selector will be evaluated against every event that occurs within the program. Similarly, lines 7–8 define an *(action) selector*, `sendToParent`, that yields a reference to the selected operation in the `parent` class. Lines 9–11 bind these together, and also define the value the pseudo-variable "this" will have in case the binding matches and its selected operation is invoked.

This design causes two different "bindings" to match the event: the `default` binding, which matches all events, as well as the just described `inheritanceBinding`. Since this would lead to the execution of two operations when both the `parent` and `child` type implement an operation with the specified message selector, we add a constraint between the default binding and the inheritance binding, on line 13–14 (in our implementation, the default binding

is globally accessible). The constraint specifies that if the default binding can successfully invoke the operation (which is the case only if the `child` type defines an operation with the name specified as the message selector of the event), we want to skip the invocation of the binding to the `parent` type. This way, operation definitions in the `child` type effectively override those in the `parent` type, while operations not defined in the `child` type are forwarded to the `parent` type, as intended by the inheritance mechanism.

```
1  module SingleInheritance { ...
2    inherits:child from:parent {
3     var inheritanceBinding, callsToChild, sendToSuper;
4
5     callsToChild = Selector new:
6       [|evt| (evt lookupType) isEqual: child];
7     sendToParent = Selector new:
8       [|evt| OperationRef new:parent withSelector:(evt selector)];
9     inheritanceBinding = Binding new: callsToChild
10      toSelector: sendToParent withContext:
11      [|evt| (Dictionary new) at: "this" put: (evt target)];
12
13    (SkipConstraint new: defaultCallBinding
14                    skip:inheritanceBinding) activate;
15    inheritanceBinding activate;
16   }
17 }
```

Listing 2. Defining a single-inheritance operator

On lines 14 and 15, both the constraint and the binding are "activated", that is, registered with the *Co-op* interpreter. From this point on, the binding and constraint are "active" within the program, and will react to events.

Listing 3 shows an example that uses the inheritance operator, reflecting the inheritance structure shown in Figure 1. The first 4 lines define the inheritance structure. Line 6 creates an instance of `Employee`. Line 7 generates an event with selector `performTask`, with the just created `Employee` object as its target. Since the module `Employee` indeed defines an operation `performTask`, the default binding matches and is able to successfully invoke this operation. Since the constraint between default- and inheritance binding prevents the inheritance binding from carrying out its job in case the default binding already handled it, it does not further attempt to invoke `performTask` on module `Person`.

```
1 inh = SingleInheritance new;
2 inh inherits: "Employee" from: "Person";
3 inh inherits: "Staff" from: "Employee";
4 inh inherits: "Secretary" from: "Employee";
5
6 e = Employee new;
7 e performTask: "write spec";
8 e setName: "John Smith";
```

Listing 3. An example using single inheritance

On line 8 however, an event with selector `setName` is generated. Figure 4 explains the control flow in this case.

First, all event selectors are evaluated. Both the event selectors belonging to the inheritance binding and the default binding match this particular event. However, the default binding cannot successfully invoke the operation "setName" within the current lookup type (`Employee`), since the module does not contain such an operation. Thus, since the default binding is unsuccessful, the inheritance action selector is evaluated. This selector "rewrites" the lookupType of the event to `Person`, and then re-attempts to dispatch the event, again matching it to all selectors, etc. Because of this "multi-stage" dispatch, it is possible to correctly deal with multiple levels of inheritance. After the event has been rewritten, the default binding is able to invoke the operation `setName` on the current lookupType (`Person`). The default binding thus implements the final dispatch stage of the evaluation.

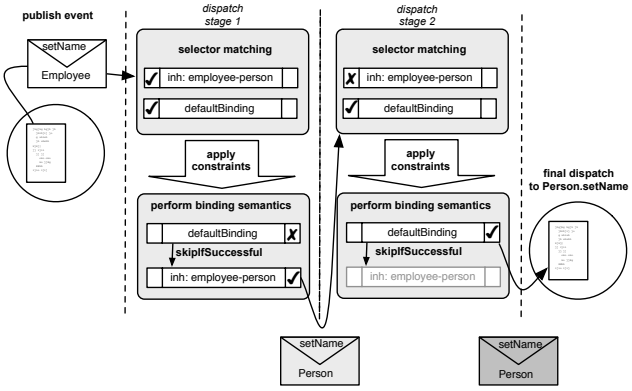

Figure 4. Example of message dispatch; single inheritance

4.2 Delegation

In this section, we implement a delegation mechanism. The intention, as discussed in section 2, is to delegate "calls" to objects of type `Staff` to their respective `Secretary`. In practice, this means that calls to `Staff` will be delegated to a connected object of type `Secretary`, in case the module `Staff` does not implement the invoked operation itself. In addition however (and this part cannot be implemented in languages that do not explicitly support delegation), even when the call is forwarded, the pseudo-variable "this" keeps referring to the original `Staff`-object, so that any "this"-calls within `Secretary` will be dispatched to `Staff` first, and are again only handled by `Secretary` if module `Staff` does not implement that operation itself.

```
1  staff = Staff new; staff setName: "John";
2  secr = Secretary new;
3  Delegation newFrom: staff to:secr;
4  staff schedule: "today" description: "meeting"; //handled by
       secretary
5
6  module Secretary {
7    var agenda; ...
8
9    schedule:time description:appointment {
10     Console writeln: ("Appointment with " cat: (this getName));
11     agenda at: time put: appointment;
12     ...
13 }
```

Listing 4. An example using delegation

Listing 4 shows an example. Lines 1 and 2 create objects of type `Staff` and `Secretary`. Line 3 establishes a delegation-relation between them—we discuss the implementation of module `Delegation` below. Now, on line 4, we make a call to operation `schedule`, even though module `Staff` does not implement such an operation (see Figure 1). Because of the delegation mechanism however, the call is forwarded to the `Secretary` object, of which lines 6–13 show a part of the implementation. Within operation `schedule`, references to variable "this" still refer to the original staff object. Thus, this example prints "Making an appointment for John". Without delegation, the "this" object would refer to the secretary, and thus print the name of the secretary instead.

So far, to implement inheritance, it was only required to rewrite the lookup type of an event. To implement delegation, two steps are necessary: (1) we redirect invocations to particular objects to a completely different object altogether, and (2) even so, we want the pseudo-variable "this" to still refer to the original object.

```
1 module Delegation {
2   var delegationBinding;
3
4   initFrom:from to:to {
5     var overrideConstraint, callsToFrom, sendToForward;
```

```
6
7     callsToFrom = Selector new:
8       [ |event| ((event target) isSameObject: from) and:
9                 ((event lookupType) isEqual: (from type))];
10    sendToForward = Selector new:
11      [ |event| OperationRef newInModule: (to type)
12        withSelector: (event selector)
13        withAnnotation: (event callAnnotation) withTarget: to];
14    delegationBinding = Binding newFromSelector: callsToFrom
15      toSelector: sendToForward
16      withContext: [(Dictionary new) at: "this" put: from];
17
18    overrideConstraint = SkipConstraint new: defaultCallBinding
          skip: delegationBinding;
19    delegationBinding activate;
20    overrideConstraint activate;
21  }
22
23  getBinding { return delegationBinding; }
24 }
```

Listing 5. Definition of a delegation operator in *Co-op*

Listing 5 shows how this can be implemented in *Co-op*. The event selector on line 7–9 matches events of which the intended target object equals `from`, which is a parameter of the delegation instance (e.g., in listing 4, variable `staff` is used as a value for this parameter). The action selector on line 10–13 rewrites the event, such that the lookup type is set to the type of parameter `to`, the selector is unchanged, the call annotation (if any) is unchanged, and the target object is changed to the value of parameter `to`. When creating the binding, on line 14–16, we instruct the binding to set the pseudo-variable `this` to the original target object `from`, rather than having it set to the (now modified!) event target. Finally, line 18 specifies a constraint, such that calls are only delegated when the original receiver object does not implement a desired behavior (such as agenda functionality) itself.

4.3 Aspects

In addition to common object-oriented composition operators, the query-based approach to matching events can also be employed to implement aspects. Listing 6 shows the definition of a general-purpose pointcut-advice mechanism, that supports *before* and *after* advice (similar to AspectJ). Instances of this aspect module are parameterized by an advice type (*before* or *after*), a module- and operation-pattern to match (only very simple "patterns" are supported: "`*`" matches everything, otherwise a concrete module or operation-name is expected), the advice module and operation to be executed, and an object on which the advice should be invoked, so that advices can also share state ("aspect state").

Pattern evaluation is implemented on lines 7–13; if an event matches the specified patterns, both sub-"pointcuts" will match. In line 15–16, these definitions are combined into a single *event selector* (cf. pointcut). The advice to be invoked is specified on line 17–22. Lines 24–25 define a constraint that orders the execution of the aspect relative to the default binding, executing the aspect either before or after the operation invoked by the default binding. Note that in this case, the execution of the original invocation is *not* skipped if the execution of the aspect is successful, but rather, the invocations are only *ordered* with respect to each other.

```
1  module AspectJLikePointcutAdvice {
2    init:advType matchClass:matchClass matchSel:matchSel
3                aspectInstance:aspectInstance adviceSel:adviceSel {
4      var binding, constraint, pointcut, advice;
5      var classMatchExpr, selMatchExpr;
6
7      classMatchExpr = [|event| (matchClass isEqual: "*")
8        ifTrue: [true]
9        ifFalse: [matchClass isEqual: (event lookupType)]];
10
11     selMatchExpr = [|event| (matchSel isEqual: "*")
12       ifTrue: [true]
13       ifFalse: [matchSel isEqual: (event selector)]];
14
15     pointcut= Selector new: [|evt| (classMatchExpr execute: evt)
16                               and: (selMatchExpr execute: evt)];
17     advice = Selector new: [OperationRef newInModule:
18      (aspectInstance type) withSelector: adviceSel
19      withAnnotation: "" withInstance: aspectInstance];
20     binding = Binding newFromSelector: pointcut
21      toSelector: advice withContext:
22      [|evt|(Dictionary new) at: "this" put: (evt target)];
23
24     (advType isEqual: "before") ifTrue:[constraint =
           PreConstraint new: binding before: defaultCallBinding];
25     (advType isEqual: "after") ifTrue: [constraint =
           PreConstraint new: defaultCallBinding before: binding];
26     constraint activate;
27     binding activate;
28   }
29 }
```

Listing 6. Definition of a pointcut-advice mechanism in *Co-op*

Listing 7 shows an example using the above composition operator. When initialized, `LogOfficeTasks` creates an instance of the pointcut-advice operator (line 5–7) that *before* the execution of operation `performTask()` in any module in the system, invokes the operation `logTask` in module `LogOfficeTasks`, using the `LogOfficeTasks` instance itself (`this`) as the advice context. From that point on, whenever the operation `performTask()` is invoked, the advice in `logTask()` is invoked before the actual execution of `performTask()`. In this example, the advice keeps track of progress (line 14), thus demonstrating the sharing of state between advice executions.

```
1  module LogOfficeTasks {
2    var progress;
3    init {
4      progress = "";
5      AspectJLikePointcutAdvice new: "before"
6        classMatch: (this classToLog) opMatch: (this optoLog)
7        aspectInstance: this advice: "logTask";
8    }
9    classToLog { return "*"; }
10   opToLog { return "performTask"; }
11
12   logTask {
13     Console writeln: "Log: performing task, current progress: ";
14     progress = progress cat: "+";
15     Console writeln: progress;
16   }
17 }
```

Listing 7. An example using the pointcut-advice mechanism

In addition, note that the task-logging aspect can be extended just like other modules, for example to override the operations `classToLog` and `operationToLog`, which (in a sense) define the "pointcuts" of the logging aspect. This way, it is possible to combine the use of several composition operators.

5. Composition of Composition Operators

Our composition model enables "composition" at different levels, which we distinguish here for the sake of clarity:

- In section 4, we constructed new composition operators that support expressing various object-oriented as well as aspect-oriented compositions.
- Multiple kinds of composition operators can be used (mixed) in the same program. For example, the *delegation* operator was demonstrated in an example that also involves an *inheritance* operator.
- Since the concepts used to define composition operators are modeled as first-class entities (objects) within the program, composition operators can also themselves be composed of (or,

can reuse parts of) other composition operators. We exemplify this below.

To illustrate the relations between regular modules and composition operators, figure 5 shows the composition relations in the example that we have demonstrated in this paper.

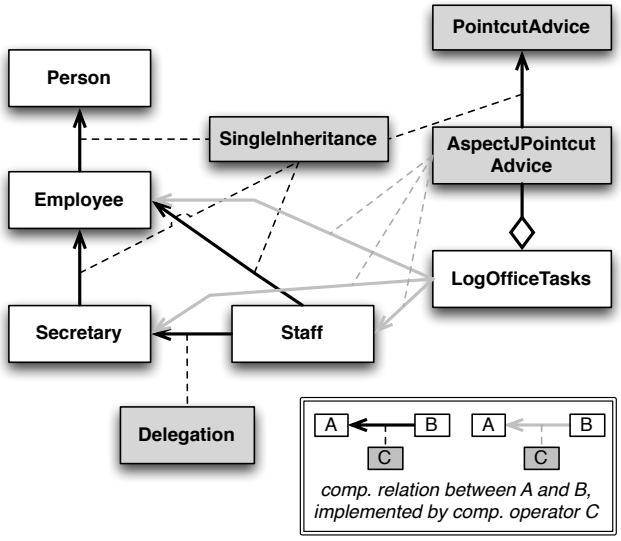

Figure 5. An overview of the composition relations discussed in the example in this paper

In particular, note how the two composition operators `PointcutAdvice` and `AspectJPointcutAdvice` are composed through the use of another composition operator, `SingleInheritance`. We briefly discuss this alternative implementation of pointcut-advice, which is shown in listing 8.

```
1  module PointcutAdvice {
2    init(advKind, aspectInstance, adviceMethod) {
3       var binding, constraint, advice;
4
5       advice = Selector new:
6         [OperationRef newInModule (aspectInstance type)
              withSelector: adviceMethod
7           withAnnotation: "" withInstance: aspectInstance];
8       binding = Binding
9         newFromSelector: (Selector new: (this getPointcut))
10        toSelector: advice withContext:
11         [|evt| (Dictionary new) at: "this", put: (evt target)];
12      // Remainder equal to listing 6, lines 24-27 ...
13   }
14 }
15
16 module AspectJPointcutAdvice {
17   var classMatchExpr, operMatchExpr;
18
19   initType:advType matchClass:matchClass matchOper:matchOper
20      aspin:aspectInstance, method:adviceMethod) {
21     classMatchExpr=[|event| (matchClass isEqual:"*")
22       ifTrue: [true]
23       ifFalse: [matchClass isEqual:(event lookupType)]];
24
25     operMatchExpr = [|event| (matchOper isEqual:"*")
26         ifTrue:[true]
27         ifFalse:[matchOper isEqual:(event selector)] ];
28
29     this@super
30       initType:advType aspin:aspectInstance method:adviceMethod;
31   }
32
33   getPointcut {
34     return [|event| (classMatchExpr execute:event)
35               and: (operMatchExpr execute:event)];
36   }
37 }
```

Listing 8. Pointcut-Advice operator with improved modularity

Note that the implementation shown above behaves in exactly the same way as the one defined in listing 6; we therefore do not explain it line by line. The important difference is the improved modularization: listing 8 splits the implementation of pointcut-advice into two modules: `PointcutAdvice` is an *abstract class* that implements the execution of an advice within the desired aspect instance, *before* or *after* an operation invocation. However, it does not itself define an implementation of operation `getPointcut()`, which has to be implemented by subclasses that may thus implement different styles of pointcut expressions. Module `AspectJPointcutAdvice` embodies such an extension, and implements a version of operation `getPointcut()` that evaluates an event against class- and operation matching patterns, in a way that is similar to (a subset of) AspectJ. Through inheritance, it depends on its (abstract) "parent" class, `PointcutAdvice`, to define the behavior that is not related to pointcut evaluation.

This example demonstrates that it is possible to use existing composition operators (such as inheritance) while defining new composition operators (in this case, a pointcut-advice mechanism).

6. Discussion

6.1 Intended behavior of compositions

We note that our approach cannot *automatically* guarantee that compositions that involve multiple composition operators exhibit the desired behavior. It is either the responsibility of the programmer who designs and implements a new composition operator, or of the application programmer that combines multiple compositions, to ensure that it works correctly if combined with other composition operators—if such use is intended.

Our model does however *facilitate* the implementation of constraints between multiple composition operators, by supporting declarative constraint specifications. For example, the definition of an aspect oriented composition operator in section 4.3 specifies a *partial ordering constraint* between the execution of aspect-related behavior and the default binding. In other words, a `PreConstraint` ensures that the action specified by one binding must be executed *at some point* before the action specified by the other, but not necessarily *immediately* before. If other composition operators (e.g., inheritance) match the same event, in addition to the aspect-oriented operator, it may be the case that both composition operators specify constraints in relation to the default binding. However, unless explicit constraints are added directly between these two composition operators, the "precedence" between these composition operator is undefined, if they both match the same event.

When allowing the combined use of multiple composition operators, the necessary constraints between composition operators depend on domain knowledge about the defined composition specifications. For this reason, such constraints cannot in general be derived automatically. As an example, should inheritance be resolved before evaluating aspects, or vice versa? This needs to be decided by the designers of the respective composition operators, or even the application itself [5].

In some cases, composition operators implement inherently conflicting notions of composition, and can therefore never be composed in a meaningful way. For example, when adding a single module to both a Beta-like inheritance hierarchy as well as a

[5] Note that this example corresponds roughly to *call* respectively *execution* join points in AspectJ; there it is left to the application programmer to decide [5].

Smalltalk-like inheritance hierarchy, the results can never be correct, since both operators have an inherently incompatible notion of inheritance (unless adopting an approach such as in [18]). Still, as long as each module occurs in at most one of the hierarchies, even these composition operators can both be used in the same program.

6.2 Reasoning about correctness and optimizations

Since our approach is currently based on an interpreter and dynamic typing, it is hard to implement any kind of static reasoning on the resulting compositions. Since it was primarily our intention to allow the definition of composition mechanisms with maximum flexibility, we have designed the language in a way that does not impose many constraints. For example, currently the only fixed part of an event selector, is the fact that they take an event as input, and return a boolean result as output. Since functions (operations, closures) are already supported by the language, we did not have to add any special language syntax to describe the selectors and other parts of the base composition mechanism.

It would however be possible to define a more declarative selector-language, to which static reasoning or partial evaluation could be applied more easily. In addition, partial evaluation of selectors could reduce the amount of event generation statements that can potentially match each selector, thus improving both the possibilities for optimization and reasoning about the program.

Performance has deliberately been excluded as a consideration in the design of the language and composition mechanism. The prototype implementation will exhibit slow performance for larger examples. One of the main reasons is that every event is to be considered at least once by all selectors, which all need to be evaluated for each event. We do believe that the performance for such a language can be substantially improved, by applying many known techniques for optimization and efficient implementation of dynamic languages [11], as well as models for implementing aspect-based mechanisms [7]. Suggestions already mentioned above are partial evaluation, as well as adding restrictions to the freedom that the current interpreted version allows (e.g., by defining a more structured selector language).

6.3 Implementation of *Co-op*

The *Co-op* language as well as the examples discussed in this paper, are implemented and available for download [2]. The current prototype is implemented on top of the Java Aspect Metamodel Interpreter (JAMI), which was presented in prior work [22]. More details about the *Co-op* language, its syntax, semantics and implementation can also be found in [21, Chapter 5 and Appendix A].

One important design consideration is that the operation implementations of built-in modules (for example those representing selectors, bindings, constraints, events, strings and booleans) are implemented natively, and thus their internal actions are not seen as "events". This is necessary, as these modules need to interface with the system's implementation in JAMI. In addition, since we are constructing composition operators within the language itself, there has to be a set of "lowest level" actions, which are not themselves interpreted by composition operators. For the same reason, the "default binding" that deals with the actual call dispatch is implemented as a primitive operation, i.e. its "internal" events (which include "calls" on the event-representation object) are not seen by any custom-defined selectors. The implementation of each composition mechanism, which may itself generate events, must eventually reduce to this "lowest"-level binding. While designing composition operators, an important consideration is that its selector- and binding-implementations should not generate events that will be matched by the implementation of that mechanism itself, or the implementation of a mechanism on which it depends. Otherwise, the process of event matching itself generates new events, which recursively get processed by event matching, ad infinitum.

7. 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 composition techniques. 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; default—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 macro's in Lisp – or allow for the extension and modification of the program through metaprogramming—as we will discuss below.

Of the research that aims at unifying composition techniques, we first discuss a few that relate particularly to the example composition operators we have shown in this paper:

- In [9], mixin inheritance is presented as a generalization of both regular ('Smalltalk-style') inheritance and Beta-style inheritance, as well as CLOS-style mixins. But the mixin mechanism itself is a fixed composition operator, and cannot be used to define new composition operators. The definition of a *Co-op* composition operator that implements mixin inheritance is part of our future work; it is our belief that this will not pose substantial technical problems.
- We have demonstrated that we can define and use multiple composition operators, including the use of respectively Smalltalk/-Java style *super* and Beta-style *inner*, in parallel. In [18], it has been shown how a *specific* method dispatching technique, implemented in the language *MzScheme*, allows the usage of both inheritance styles simultaneously.
- Compositional Modularity [4] is an inheritance model that supports a wide range of compositions on modules (which correspond to self-referential namespaces in this model). This is achieved by modeling the compositions as a set of operations on the modules. Compared to our approach, there are limitations to its expressiveness, due to the fixed set of primitive operations. The compositional modularity model has been applied to a variety of 'base' artifacts.
- Expressive, tailorable, message dispatching is a key component of our approach; the work on *predicate dispatch*[15, 30] is closely related, key distinctions are our focus on first class composition operators and implicit invocation model that supports also aspects.
- *Classpects*[35] unify aspect- and object-oriented programming. The language *Eos-U* implements the *classpect* construct, which can be considered as a combination of aspect-behavior and object behavior in a single abstraction mechanism. Eos-U offers the concept of bindings, which have roughly the same structure as the bindings in *Co-op*: binding advice to join points. However, there is no mechanism for expressing ordering constraints beyond declaration order. Regardless of these similarities, Eos-U is distinct from *Co-op* by offering only a fixed set of composition operators and abstractions.

- Composition filters (or interface predicates) in the Sina language [6] define a single language mechanism that can be used –among other things– to express various data abstraction mechanisms such as different forms of inheritance (single, multiple, conditional), delegation, and aspects. *filter modules* are abstractions of several filter expressions, but not an independent composition operator. The introduction of new filter types can be used to add additional composition behavior to a system, but all within the same framework of composition filters, not as new, independent composition operators.

There are other approaches that allow for the construction of user-defined composition operators. In particular, our work relates to metaprogramming [12] and especially meta-object protocols[25]. Depending on the programming language/environment, metaprogramming offers the programmer the full power to modify the behavior of programs. This includes the ability to write custom compositions. As explained e.g. in [24], the power of metaprogramming comes with more complexity and responsibility. In particular, it may be extremely hard to define multiple application-specific compositions in such a way that they work together without interference (i.e. such that they are composable).

Meta-object protocols (MOPs) aim at addressing this by providing a framework –albeit at the metalevel– with more structure and constraints, so that e.g. composition operators can be defined within a well-defined structure. 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 mechanisms. In *Co-op*, composition mechanisms are constructed using first-class, composable elements, which can be reused to define or compose new composition mechanisms. In addition, the resulting composition operators are also first-class entities, which means they can be composed, reused and extended as well.

It may well be possible to implement our approach as a metaobject protocol on top of, e.g., CLOS. However, the core contribution of our approach is not the *Co-op* language itself, but rather the model of composition that is enabled by the elements presented in section 3, e.g., selectors, bindings, constraints. These elements provide explicit support for the expression of composition mechanisms that are based on the notion of *implicit invocation*, such as aspects. In addition, our approach supports the expression of explicit, declarative constraint specifications to address dependencies between composition mechanisms. In a CLOS-based implementation, we would still have to provide all the novel abstractions and infrastructure that we have presented in this paper.

Of the research that aims at providing frameworks for higher-level languages through reflection or meta object protocols, we briefly discuss the following:

- [34] describe an "open, extensible object model" which shares some of our goals, as expressed by: *Raising the implementation [of the language] to the programmers' level lets them design and control their own implementation mechanisms in which to express concise solutions and free the original language designer from ever having to say "I'm sorry"*. Another important goal of this work is to come up with the smallest possible language implementation that is programmer-accessible, and allows for bootstrapping more complex object models. As a result, this work differs from our proposal that it aims at the most simple mechanisms, essentially based on allowing programmers to redefine method lookup in arbitrary ways. In contrast, we provide a model that offers a specifically designed structure (or: more detailed meta protocol), including selectors, constraints and bindings. In addition, our model is class-based, rather than prototype-based, and handles and integrates both aspect-oriented and object-oriented models.
- AspectS [23] is framework for supporting aspect-programming in Smalltalk. It extends the Smalltalk MOP with features that enable aspect programming. As such, it does not extend the language itself. For instance, it uses Smalltalk itself as the pointcut language, similar to our use of the 'base' language for defining selectors.
- MetaClassTalk [8] aims at 'unified aspect-oriented programming'. It exploits a combination of mixin-based inheritance and reflection to achieve this. Its aspects consist of (a) a set of mixins, (b) a pre-weaving script, (c) a post-weaving script. In this approach, every programmer is a meta-programmer, with a lot of control—and responsibility to write correct meta-programs. MetaClassTalk also involves 'weave-time' code; which is a disadvantage from the point of view of abstraction, but does have potential benefits with respect to performance optimization and static reasoning.

In [29], Masuhara and Kiczales propose the *Aspect Sand Box*, an interpreter framework to model aspect mechanisms. Using this framework, the effects of aspects are defined in terms of weaving semantics. The weaving process is modeled by extending or modifying the interpreter of a base language that models a single-inheritance object-oriented language (which can be seen as a core subset of Java). This approach differs from ours in that it is a flexible way to define composition operators, but as a new, fixed language, and not expressed or extensible within the language itself. The approach by Kojarski and Lorenz in [26], is different from ours in a similar way, even though it aims at supporting multiple aspect composition operators as part of Domain Specific Aspect Languages that can be combined in a single application.

Finally, we mention several frameworks that aim at offering a generic platform for OO and AOP language implementations (e.g. [3], [1], [10], [20], [38], and [14]). For such platforms, the designers have also 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.

8. Evaluation and Conclusion

In this paper, we presented the *Co-op* composition model and language. The main goal of *Co-op* is that it enables the creation—and usage—of first-class composition operators for expressing a wide variety of composition techniques. Examples that we have demonstrated in this paper are single inheritance, delegation and pointcut-advice, although we have also implemented various other operators (not included in this paper), including multiple inheritance, beta-style inheritance, support for super- and inner-calls, and a domain specific composition for observations. The implementation of these compositions can be found in [2]. In addition, we have defined a detailed denotational semantics of the *Co-op* language, which is included in [21].

This paper makes the following contributions:

1. Using a case study that discusses alternative inheritance semantics, delegation and aspects, we argue that languages with fixed

composition semantics cannot adequately express designs without sacrificing some desired design properties.

2. We present a novel composition model that supports composition operators as user-defined, modular and reusable first-class abstractions. We are not aware of any language or MOP that has a similar model (design) and characteristics.
3. The paper illustrates the instantiation of the model within a simple, experimental, object-based language (*Co-op*); this language has been implemented and tested for a number of (common) composition operators.
4. Our proposed model unifies and supports both object-oriented and aspect-oriented composition techniques; the paper illustrates this by showing how to express inheritance, delegation and aspects.
5. In particular, our model supports composition of composition operators; both the combination of multiple, different operators in a single application, and the ability to construct new composition operators from existing ones.

We believe the notion of user-defined, first-class composition operators, brings us closer to the following goal, as expressed by Guy Steele in his "growing a language" talk [19]: *"a language design can no longer be a thing. It must be a pattern – a pattern for growth - a pattern for growing the pattern for defining the patterns that programmers can use for their real work and their main goal."* In this case the pattern is a means to grow (by composition) user-defined composition operators, which express particular patterns of interaction among modules.

8.1 Design Considerations and Lessons learned

To achieve a better understanding how *Co-op* achieves the features we just presented, we will now discuss the key elements in the design of the *Co-op* model, and discuss how they contribute to the capabilities of *Co-op*. Although these design elements cannot be seen in isolation, we believe that the discussion below is partially generalizable to other composition techniques. In fact, many observations have been derived from the experiences in the design of object- and aspect-oriented languages.

- *Implicit invocations*: to be able to offer a generalized mechanism for both object-based and aspect-based compositions, the core design of the model needed to fully decouple message sends (i.e. event generation) from an eventual method execution. For this reason we adopt the notion of implicit invocations [32]; there exists a dynamic, one-to-many relationship between a message send and the possible operation executions or metalevel actions.
- Use of *queries* (the 'selectors' in our model) for selecting events and actions: the advantage of specifying a query instead of a fixed, 'hard-coded' identifier to refer to program elements, is that a query allows for much more conceptual/semantic relations (see e.g. [31]), rather than accidental and inflexible identifier-based connections.
- Use of *reflective information* (event reflection, program introspection and state information—through object interfaces) within queries: this is also one of the lessons from aspect languages; the use of context information—through a generic interface— is a powerful means to pass on context information between modules without making the modules dependent on each other (i.e. improving the composability). This context information may be available only at run-time.
- Concept of *bindings* for (a) associating queries with actions, and (b) associating data variables in different contexts: the first is a common technique in AOP languages (see e.g. Eos-U, as discussed in section 7).
- Use of *constraints* for composing bindings: this turned out to be a crucial issue in achieving composability; the ability to express constraints, including dynamic constraints, at a fine-grained, per-binding level. A topic of future work is whether it is important to be able to apply constraints to particular groups of bindings, which may even be selected by queries
- Using *metalevel actions* to manipulate events: composing systems through meta-level manipulation of messages between the system components has been proven before to be a successful technique [6, 16]. It has two attractive properties: it allows to reason about the system without breaking the encapsulation of the individual components, and it is relatively easy to offer a generic, shared abstraction of messages, which avoids application-specific dependencies.
- Representing composition operators, bindings, queries (selectors) and constraints as *first-class* citizens: this allows the explicit definition, manipulation and reuse of these elements. For example, this avoids the necessity of dedicated language constructs or keywords for referring to these elements.
- *Multi-stage dispatch*: an important feature, because it helps to realize transitive composition relations, such as exemplified by inheritance in section 4.1, where the method lookup for a single message send may involve multiple bindings. Multi-stage dispatch also promotes the composability of composition operators, as it allows for the application of multiple composition operators for a single event.
- *Dispatch to multiple operations*: this means that a single event may yield multiple actions and even multiple operation executions. This allows for example expressing before/after advices in an AOP style, and also less common examples of multiple dispatch, such as multi-cast semantics. Note that both multi-stage dispatch and dispatch to multiple operations require proper ordering constraints.

8.2 Future Work

There are still many possible issues to explore. An important issue that needs to be further investigated, is to what extent it is necessary to specify constraints among (bindings of) different composition operators, as this bears the theoretical risk that for every new composition operator, all possible interactions with existing composition operators needs to be investigated and specified. However, to date, we have not experienced such a need. As an additional future work, there are still many composition techniques that we would like to experiment with, and demonstrate that they can be expressed using *Co-op*. Examples include: traits [36], mixins [9], composition filters [6], and so forth. Along these lines, we are also interested to implement a composition framework as proposed in [33], which outlines a number of core operations from which a wide range of OO compositions can be constructed: it would be interesting to be able to express the core operations as separate composition operators, and use those to compose new, higher-level, composition operators.

References

[1] Java Aspect Metamodel Interpreter - `http://jami.sf.net/`, 2007.

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

[3] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible aspectj compiler. *Transactions on Aspect-Oriented Software Development I*, 3880/2006:293 – 334, February 2006.

[4] G. Banavar and G. Lindstrom. An application framework for module composition tools. In *In ECOOP '96, number 1098 in Lecture Notes in Computer Science*, pages 91–113. Springer Verlag, 1996.

[5] O. Barzilay, Y. A. Feldman, S. Tyszberowicz, and A. Yehudai. Call and execution semantics in AspectJ. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 19–24, Mar. 2004.

[6] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

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

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

[9] G. Bracha and W. Cook. Mixin-based inheritance. In *Conf. Object-Oriented Programming: Systems, Languages, and Applications; European Conf. Object-Oriented Programming*, pages 303–311. ACM, 1990.

[10] J. Brichau, M. Mezini, J. Noyé, W. Havinga, L. Bergmans, V. Gasiunas, C. Bockisch, J. Fabry, and T. D'Hondt. An Initial Metamodel for Aspect-Oriented Programming Languages. Technical Report AOSD-Europe Deliverable D39, Vrije Universiteit Brussel, 27 February 2006 2006.

[11] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, 1989.

[12] P. Cointe. Reflective languages and metalevel architectures. *ACM Comput. Surv.*, page 151, 1996.

[13] B. C. d. S. Oliveira. Modular Visitor Components. In *Proceedings of the 23th European Conference on Object-Oriented Programming (ECOOP 2009)*, 2009.

[14] R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, New York, NY, USA, 2008. ACM.

[15] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. *Lecture Notes in Computer Science*, 1445:186–211, 1998.

[16] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Comm. ACM*, 45(1):116–122, Jan. 2002.

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

[18] 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.

[19] J. Guy L. Steele. Growing a language. *Higher Order Symbol. Comput.*, 12(3):221–236, 1999.

[20] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP 2007: Proceedings of the 21th European Conference on Object-Oriented Programming*, pages 501–524, 2007.

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

[22] 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.

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

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

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

[26] S. Kojarski and D. H. Lorenz. Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. *SIGPLAN Notices*, 42(10):515–534, 2007.

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

[28] O. L. Madsen, B. Mø-Pedersen, and K. Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[29] H. Masuhara and G. Kiczales. Modular crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743 of *lncs*, pages 2–28, Berlin, July 2003. Springer-Verlag.

[30] T. Millstein. Practical predicate dispatch. *ACM SIGPLAN Notices*, 39(10):345–364, 2004.

[31] I. Nagy, L. Bergmans, W. Havinga, and M. Aksit. Utilizing design information in aspect-oriented programming. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[32] D. Notkin, D. Garlan, W. G. Griswold, and K. J. Sullivan. Adding implicit invocation to languages: Three approaches. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, London, UK, 1993. Springer-Verlag.

[33] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proc. OOPSLA '01 Conf. Object Oriented Programming Systems Languages and Applications*, pages 283–299. ACM Press, 2001.

[34] I. Piumarta and A. Warth. Open, extensible object models. In *Self-Sustaining Systems*, page 30. Springer, 2008.

[35] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, 2005. ACM Press.

[36] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. *Lecture notes in computer science*, pages 248–274, 2003.

[37] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[38] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, Sept. 2005.

[39] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. 21st Int'l Conf. Software Engineering (ICSE'1999)*, pages 107 – 119. IEEE Computer Society Press, May 1999.

[40] wikipedia. Metaobject, October 2009.